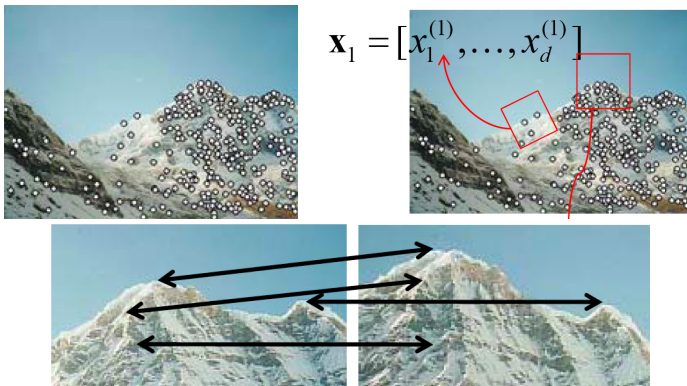


# Image Features: Local Descriptors

# Local Features

- **Detection:** Identify the interest points.
- **Description:** Extract a feature descriptor around each interest point.
- **Matching:** Determine correspondence between descriptors in two views.



[Source: K. Grauman]



# The Ideal Feature Descriptor

- **Repeatable:** Invariant to rotation, scale, photometric variations
- **Distinctive:** We will need to match it to lots of images/objects!
- **Compact:** Should capture rich information yet not be too high-dimensional (otherwise matching will be slow)
- **Efficient:** We would like to compute it (close-to) real-time

# Invariances



[Source: T. Tuytelaars]

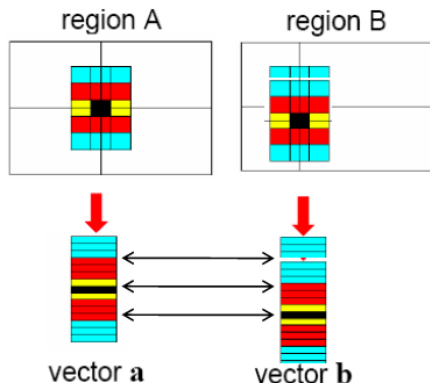
# Invariances



[Source: T. Tuytelaars]

# What If We Just Took Pixels?

- The simplest way is to write down the list of intensities to form a feature vector, and normalize them (i.e., mean 0, variance 1).
- Why normalization?
- But this is very sensitive to even small shifts, rotations and any affine transformation.



# Tones Of Better Options

- SIFT
- PCA-SIFT
- GLOH
- HOG
- SURF
- DAISY
- LBP
- Shape Contexts
- Color Histograms

# Tones Of Better Options

- **SIFT**    **TODAY**
- PCA-SIFT
- GLOH
- HOG
- SURF
- DAISY
- LBP
- Shape Contexts
- Color Histograms

# SIFT Descriptor [Lowe 2004]

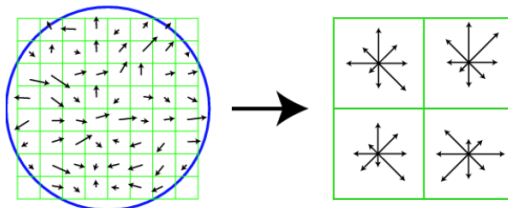
- SIFT stands for Scale Invariant Feature Transform
- Invented by David Lowe, who also did DoG scale invariant interest points
- Actually in the same paper, which you should read:

David G. Lowe

*Distinctive image features from scale-invariant  
keypoints*

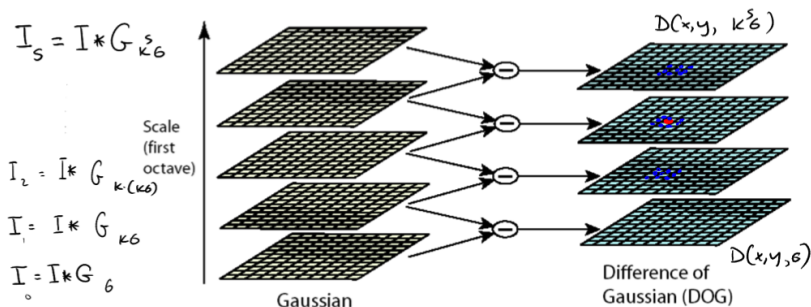
International Journal of Computer Vision, 2004

Paper: <http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf>



# SIFT Descriptor

- 1 Our scale invariant interest point detector gives scale  $\rho$  for each keypoint

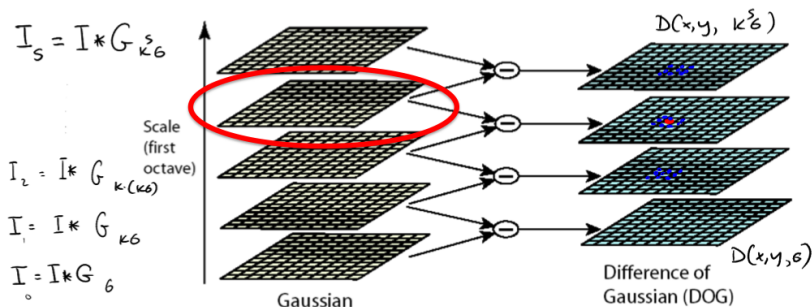


[Adopted from: F. Flores-Mangas]



# SIFT Descriptor

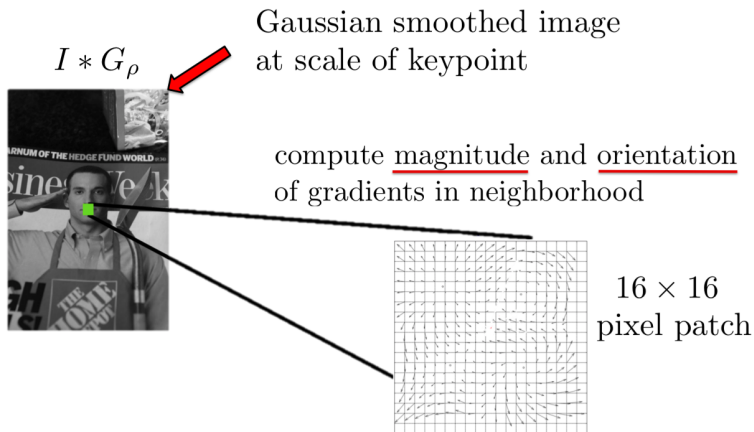
- ② For each keypoint, we take the Gaussian-blurred image at corresponding scale  $\rho$



[Adopted from: F. Flores-Mangas]

# SIFT Descriptor

- 3 Compute the gradient magnitude and orientation in neighborhood of each keypoint



[Adopted from: F. Flores-Mangas]

- 3 Compute the gradient magnitude and orientation in neighborhood of each keypoint

magnitude of gradient:

$$|\nabla I(x, y)| = \sqrt{\left(\frac{\partial(I(x, y) * G_\rho)}{\partial x}\right)^2 + \left(\frac{\partial(I(x, y) * G_\rho)}{\partial y}\right)^2}$$

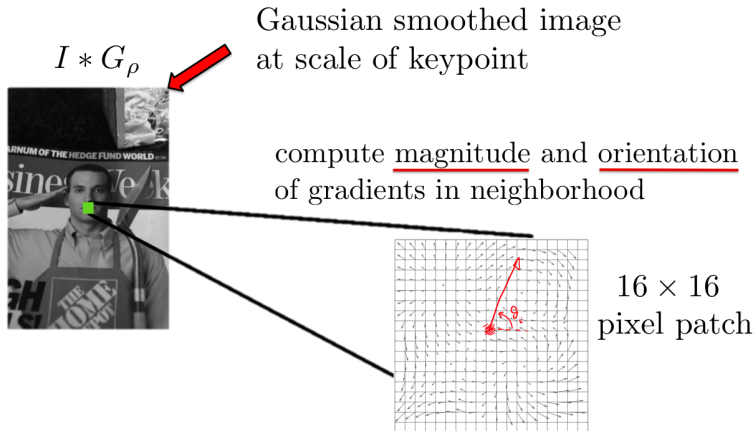
gradient orientation:

$$\theta(x, y) = \arctan\left(\frac{\partial I * G_\rho}{\partial y} / \frac{\partial I * G_\rho}{\partial x}\right)$$

(in case you forgot ;))

# SIFT Descriptor

- 4 Compute dominant orientation of each keypoint. How?

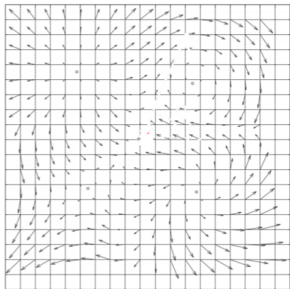


[Adopted from: F. Flores-Mangas]

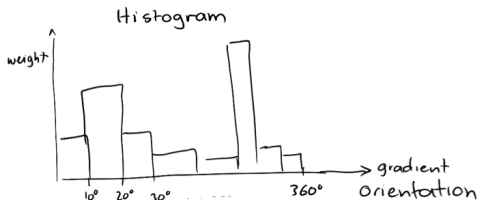
# SIFT Descriptor: Computing Dominant Orientation

- Compute a histogram of gradient orientations, each bin covers  $10^\circ$

$16 \times 16$



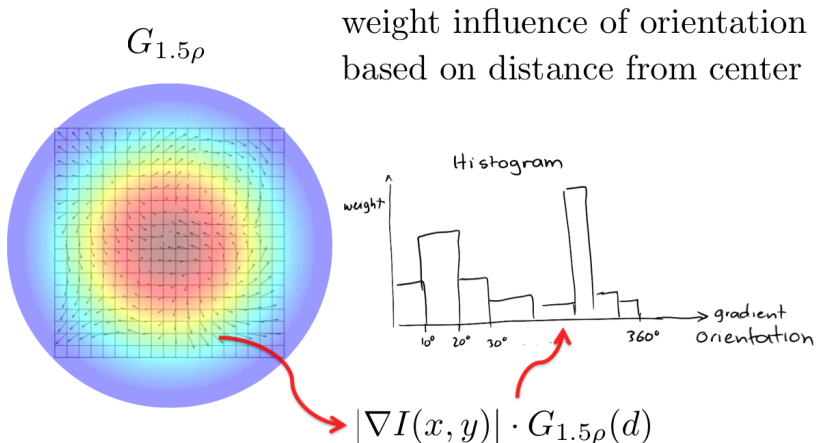
compute histograms of orientations  
by orientation increments of  $10^\circ$



[Adopted from: F. Flores-Mangas]

# SIFT Descriptor: Computing Dominant Orientation

- Compute a histogram of gradient orientations, each bin covers  $10^\circ$
- Orientations closer to the keypoint center should contribute more

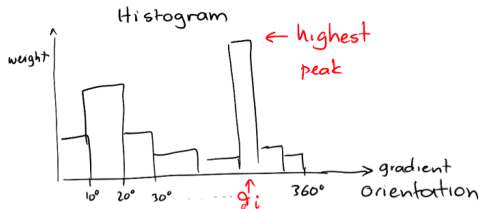
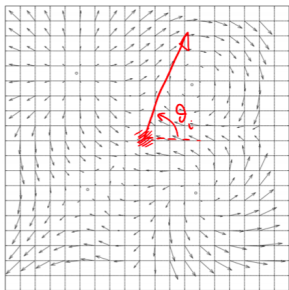


[Adopted from: F. Flores-Mangas]

# SIFT Descriptor: Computing Dominant Orientation

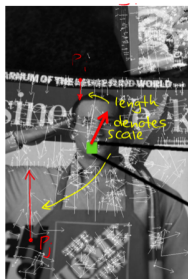
- Compute a histogram of gradient orientations, each bin covers  $10^\circ$
- Orientations closer to the keypoint center should contribute more
- Orientation giving the peak in the histogram is the keypoint's orientation

$16 \times 16$

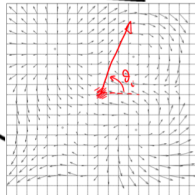


# SIFT Descriptor

## 4 Compute dominant orientation



compute magnitude and orientation  
of gradients in neighborhood



$16 \times 16$   
pixel patch

[Adopted from: F. Flores-Mangas]

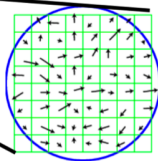


# SIFT Descriptor

- 5 Compute a 128 dimensional descriptor:  $4 \times 4$  grid, each cell is a histogram of 8 orientation bins relative to dominant orientation



compute descriptor, relative  
to dominant orientation



128 dim  
descriptor

each descriptor has:

$P_i = (x_i, y_i, \rho_i, \vartheta_i)$  and  $f_i \dots$  128 dim vector

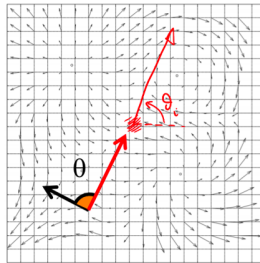
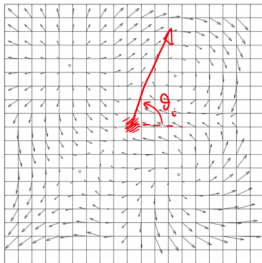
location      scale      orientation      feature vector

[Adopted from: F. Flores-Mangas]

# SIFT Descriptor: Computing the Feature Vector

- Compute the orientations **relative** to the **dominant orientation**

$16 \times 16$  patch  
centered in  $(x_i, y_i)$

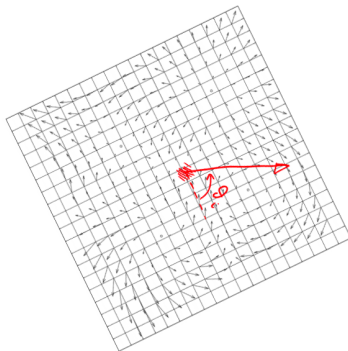
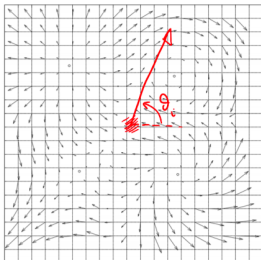


[Adopted from: F. Flores-Mangas]

# SIFT Descriptor: Computing the Feature Vector

- Compute the orientations **relative** to the **dominant orientation**

$16 \times 16$  patch  
centered in  $(x_i, y_i)$

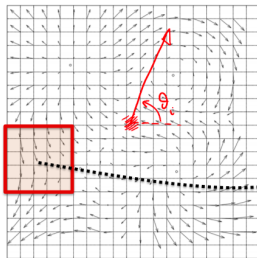


[Adopted from: F. Flores-Mangas]

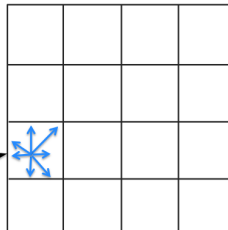
# SIFT Descriptor: Computing the Feature Vector

- Compute the orientations **relative** to the **dominant orientation**
- Form a  $4 \times 4$  grid. For each grid cell compute a histogram of orientations for 8 orientation bins spaced apart by  $45^\circ$

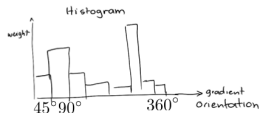
$16 \times 16$  patch  
centered in  $(x_i, y_i)$



SIFT descriptor



compute histogram of orientations  
this time 8 bins spaced by  $45^\circ$

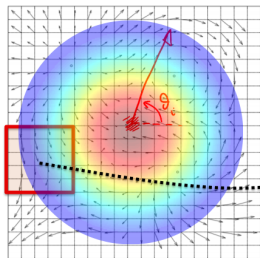


[Adopted from: F. Flores-Mangas]

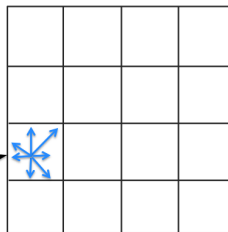
# SIFT Descriptor: Computing the Feature Vector

- Compute the orientations **relative** to the **dominant orientation**
- Form a  $4 \times 4$  grid. For each grid cell compute a histogram of orientations for 8 orientation bins spaced apart by  $45^\circ$

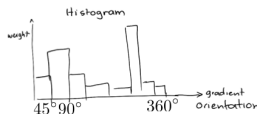
$16 \times 16$  patch  
centered in  $(x_i, y_i)$



SIFT descriptor



again weigh contributions  
this time:  $|\nabla I(x, y)| \cdot G_{0.5\rho}$

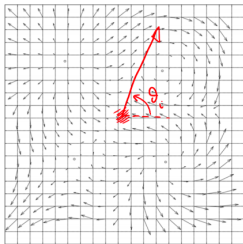


[Adopted from: F. Flores-Mangas]

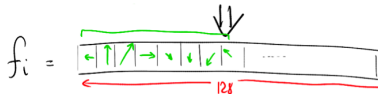
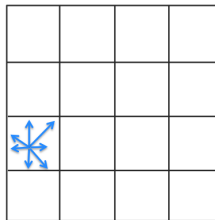
# SIFT Descriptor: Computing the Feature Vector

- Compute the orientations **relative** to the **dominant orientation**
- Form a  $4 \times 4$  grid. For each grid cell compute a histogram of orientations for 8 orientation bins spaced apart by  $45^\circ$
- Form the 128 dimensional feature vector

$16 \times 16$  patch  
centered in  $(x_i, y_i)$



SIFT descriptor



[Adopted from: F. Flores-Mangas]

# SIFT Descriptor: Post-processing

- The resulting 128 non-negative values form a **raw version** of the SIFT descriptor vector.
- To reduce the **effects of contrast or gain** (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length:  $f_i = f_i / ||f_i||$

# SIFT Descriptor: Post-processing

- The resulting 128 non-negative values form a **raw version** of the SIFT descriptor vector.
- To reduce the **effects of contrast or gain** (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length:  $f_i = f_i / ||f_i||$
- To further make the descriptor robust to other **photometric variations**, values are clipped to 0.2 and the resulting vector is once again renormalized to unit length.



# SIFT Descriptor: Post-processing

- The resulting 128 non-negative values form a **raw version** of the SIFT descriptor vector.
- To reduce the **effects of contrast or gain** (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length:  $f_i = f_i / ||f_i||$
- To further make the descriptor robust to other **photometric variations**, values are clipped to 0.2 and the resulting vector is once again renormalized to unit length.
- Great engineering effort!

# SIFT Descriptor: Post-processing

- The resulting 128 non-negative values form a **raw version** of the SIFT descriptor vector.
- To reduce the **effects of contrast or gain** (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length:  $f_i = f_i / ||f_i||$
- To further make the descriptor robust to other **photometric variations**, values are clipped to 0.2 and the resulting vector is once again renormalized to unit length.
- Great engineering effort!
- What is SIFT invariant to?

# SIFT Descriptor: Post-processing

- The resulting 128 non-negative values form a **raw version** of the SIFT descriptor vector.
- To reduce the **effects of contrast or gain** (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length:  $f_i = f_i / ||f_i||$
- To further make the descriptor robust to other **photometric variations**, values are clipped to 0.2 and the resulting vector is once again renormalized to unit length.
- Great engineering effort!
- What is SIFT invariant to?

# Properties of SIFT

Invariant to:

- Scale
- Rotation

Partially invariant to:

- Illumination changes (sometimes even day vs. night)
- Camera viewpoint (up to about 60 degrees of out-of-plane rotation)
- Occlusion, clutter (why?)

Also important:

- Fast and efficient – can run in real time
- Lots of code available

# Examples

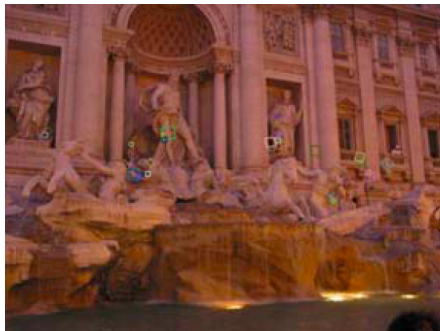


Figure: Matching in day / night under viewpoint change

[Source: S. Seitz]

# Example

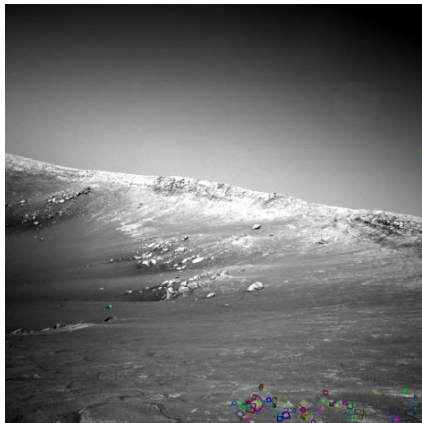


Figure: NASA Mars Rover images with SIFT feature matches

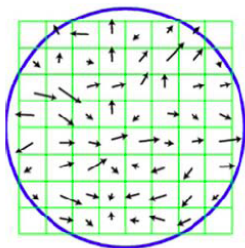
[Source: N. Snavely]

- The dimensionality of SIFT is pretty high, i.e., 128D for each keypoint
- Reduce the dimensionality using linear dimensionality reduction
- In this case, principal component analysis (PCA)
- Use 10D or so descriptor

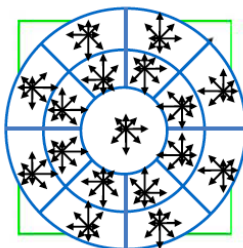
[Source: R. Urtasun]

# Gradient location-orientation histogram (GLOH)

- Developed by Mikolajczyk and Schmid (2005): variant of SIFT that uses a log-polar binning structure instead of the four quadrants.
- The spatial bins are 11, and 15, with eight angular bins (except for the central region), for a total of 17 spatial bins and 16 orientation bins.
- The 272D histogram is then projected onto a 128D descriptor using PCA trained on a large database.



(a) image gradients



(b) keypoint descriptor

[Source: R. Szeliski]

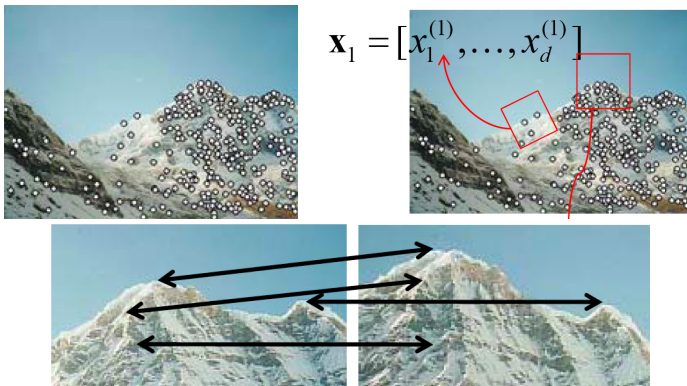


# Other Descriptors

- SURF
- DAISY
- LBP
- HOG
- Shape Contexts
- Color Histograms

# Local Features

- **Detection:** Identify the interest points.
- **Description:** Extract feature descriptor around each interest point.
- **Matching:** Determine correspondence between descriptors in two views.



[Source: K. Grauman]

# Image Features: Matching the Local Descriptors

# Matching the Local Descriptors

Once we have extracted keypoints and their descriptors, we want to match the features between pairs of images.

- Ideally a match is a correspondence between a local part of the object on one image to the same local part of the object in another image
- How should we compute a match?



Figure: Images from K. Grauman

# Matching the Local Descriptors

Once we have extracted keypoints and their descriptors, we want to match the features between pairs of images.

- Ideally a match is a correspondence between a local part of the object on one image to the same local part of the object in another image
- How should we compute a match?

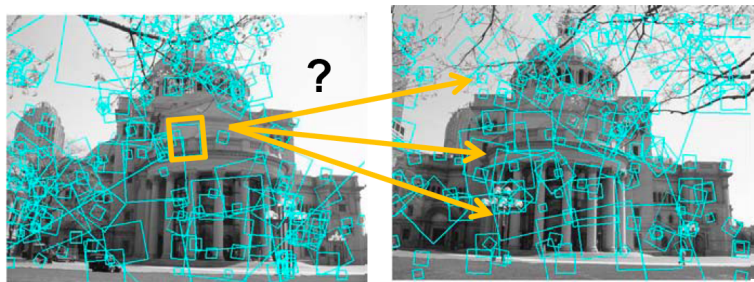
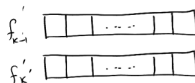
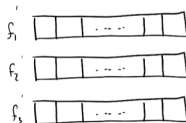
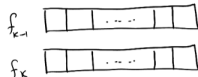
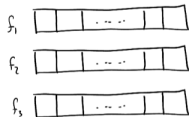
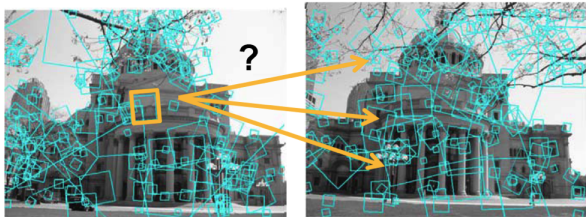


Figure: Images from K. Grauman

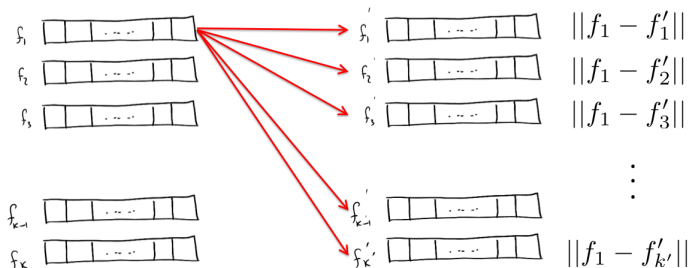
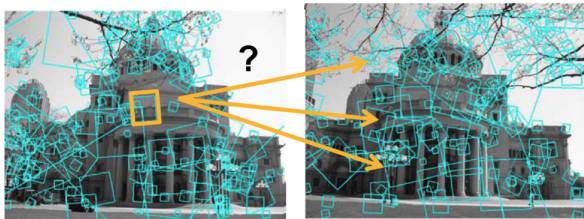
# Matching the Local Descriptors

- Simple: **Compare them all**, compute Euclidean distance



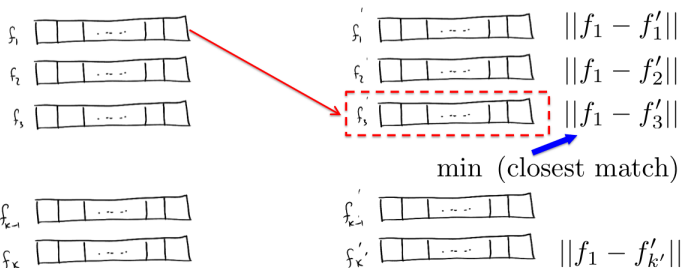
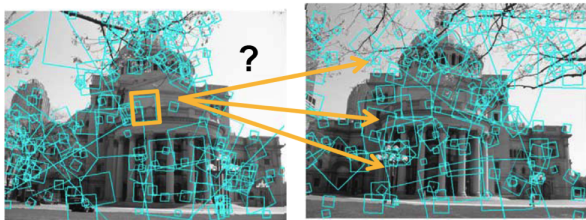
# Matching the Local Descriptors

- Simple: **Compare them all**, compute Euclidean distance



# Matching the Local Descriptors

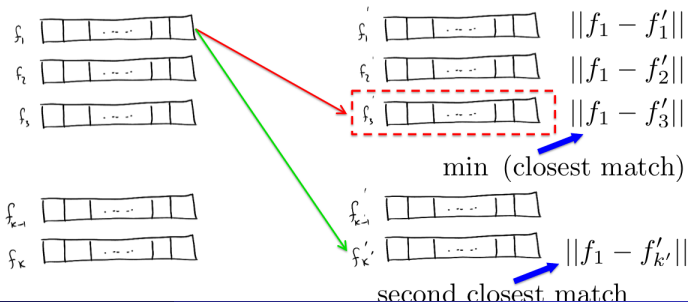
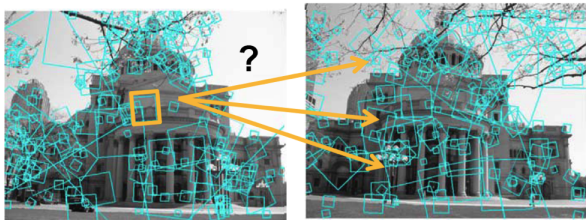
- Find closest match (min distance). How do we know if match is **reliable**?





# Matching the Local Descriptors

- Find also the second closest match. Match reliable if first distance “much” smaller than second distance

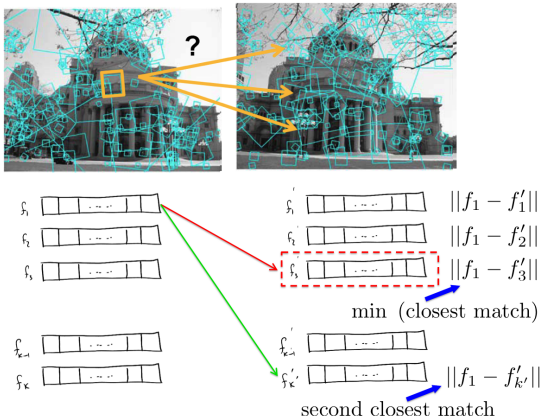


# Matching the Local Descriptors

- Compute the ratio:

$$\phi_i = \frac{\|f_i - f_i^*\|}{\|f_i - f_i^{**}\|}$$

where  $f_i^*$  is the closest and  $f_i^{**}$  second closest match to  $f_i$ .



# Which Threshold to Use?

- Setting the threshold too high results in too many false positives, i.e., incorrect matches being returned.
- Setting the threshold too low results in too many false negatives, i.e., too many correct matches being missed

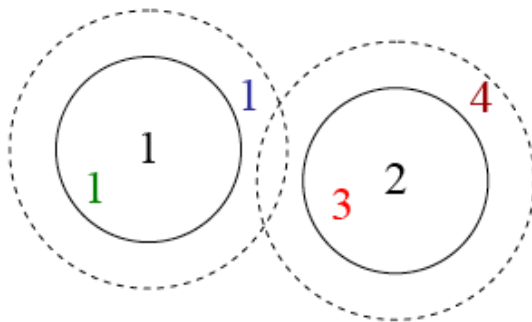


Figure: Images from R. Szeliski

# Which Threshold to Use?

- Threshold ratio of nearest to 2nd nearest descriptor
- Typically:  $\phi_i < 0.8$

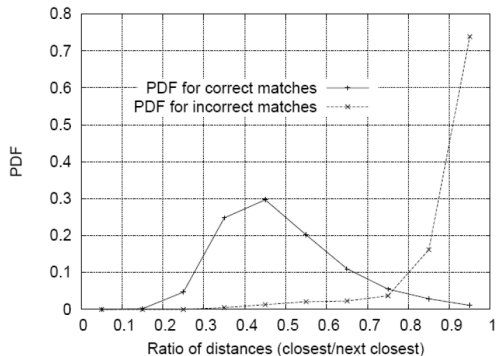


Figure: Images from D. Lowe

[Source: K. Grauman]

# Applications of Local Invariant Features

- Wide baseline stereo
- Motion tracking
- Panorama stitching
- Mobile robot navigation
- 3D reconstruction
- Recognition
- Retrieval

[Source: K. Grauman]

# Wide Baseline Stereo



[Source: T. Tuytelaars]

# Recognizing the Same Object



Schmid and Mohr 1997



Sivic and Zisserman, 2003



Rothganger et al. 2003



Lowe 2002

[Source: K. Grauman]

# Motion Tracking



Figure: Images from J. Pilet



# Now What

- Now we know how to extract scale and rotation invariant features
- We even know how to match features across images
- Can we use this to find Waldo in an even more sneaky scenario?

# Now What

- Now we know how to extract scale and rotation invariant features
- We even know how to match features across images
- Can we use this to find Waldo in an even more sneaky scenario?



template

Waldo on the road

# Now What

- Now we know how to extract scale and rotation invariant features
- We even know how to match features across images
- Can we use this to find Waldo in an even more sneaky scenario?



template

He comes closer... We know how to solve this

# Now What

- Now we know how to extract scale and rotation invariant features
- We even know how to match features across images
- Can we use this to find Waldo in an even more sneaky scenario?

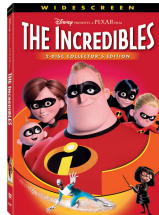


template

Someone takes a (weird) picture of him!

# Find My DVD!

- More interesting: If we have DVD covers (e.g., from Amazon), can we match them to DVDs in real scenes?



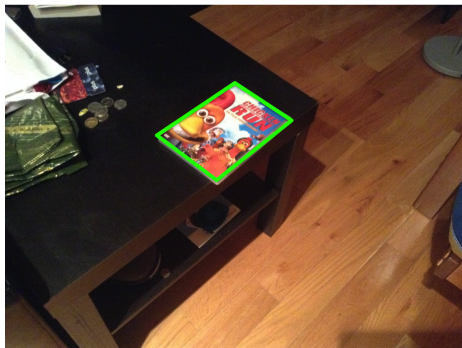
# Matching Planar Objects In New Viewpoints

# What Kind of Transformation Happened To My DVD?



# What Kind of Transformation Happened To My DVD?

- Rectangle goes to a parallelogram (almost but not really, but let's believe that for now)





# All 2D Linear Transformations

Linear transformations are combinations of

- Scale,
- Rotation
- Shear
- Mirror

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

[Source: N. Snavely]

# All 2D Linear Transformations

Properties of linear transformations:

- Origin maps to origin
- Lines map to lines

# All 2D Linear Transformations

Properties of linear transformations:

- Origin maps to origin
- Lines map to lines
- Parallel lines remain parallel

# All 2D Linear Transformations

Properties of linear transformations:

- Origin maps to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved

# All 2D Linear Transformations

Properties of linear transformations:

- Origin maps to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved
- Closed under composition

# All 2D Linear Transformations

Properties of linear transformations:

- Origin maps to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved
- Closed under composition

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{bmatrix} i & j \\ k & l \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

# All 2D Linear Transformations

Properties of linear transformations:

- Origin maps to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved
- Closed under composition

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{bmatrix} i & j \\ k & l \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

What about the translation?

[Source: N. Snavely]

# All 2D Linear Transformations

Properties of linear transformations:

- Origin maps to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved
- Closed under composition

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{bmatrix} i & j \\ k & l \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

What about the translation?

[Source: N. Snavely]



# Affine Transformations

Affine transformations are combinations of

- Linear transformations, and
- Translations

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

same as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b & e \\ c & d & f \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

# Affine Transformations

Affine transformations are combinations of

- Linear transformations, and
- Translations

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

same as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b & e \\ c & d & f \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

# Affine Transformations

Affine transformations are combinations of

- Linear transformations, and
- Translations

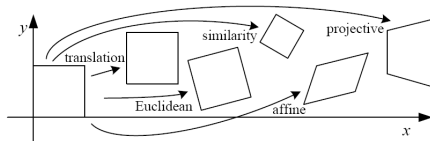
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b & e \\ c & d & f \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$






Properties of affine transformations:

- Origin does not necessarily map to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved
- Closed under composition
- Rectangles go to parallelograms

[Source: N. Snavely]

# 2D Image Transformations

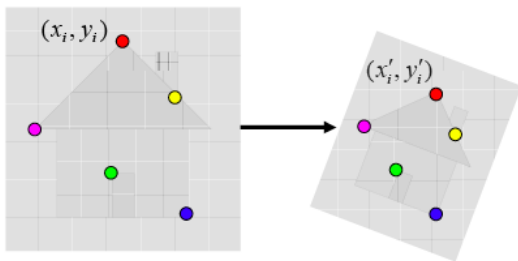


Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} I &   & t \end{bmatrix}_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} R &   & t \end{bmatrix}_{2 \times 3}$	3	lengths	
similarity	$\begin{bmatrix} sR &   & t \end{bmatrix}_{2 \times 3}$	4	angles	
affine	$\begin{bmatrix} A \end{bmatrix}_{2 \times 3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{H} \end{bmatrix}_{3 \times 3}$	8	straight lines	

- These transformations are a nested set of groups
- Closed under composition and inverse is a member

# What Transformation Happened to My DVD?

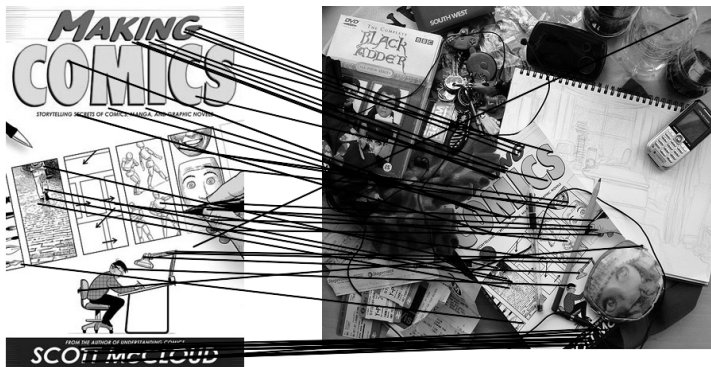
- Affine transformation approximates viewpoint changes for roughly **planar objects** and roughly **orthographic cameras** (more about these later in class)
- DVD went affine!



# Computing the (Affine) Transformation

Given a set of matches between images  $I$  and  $J$

- How can we compute the affine transformation  $A$  from  $I$  to  $J$ ?
- Find transform  $A$  that best agrees with the matches

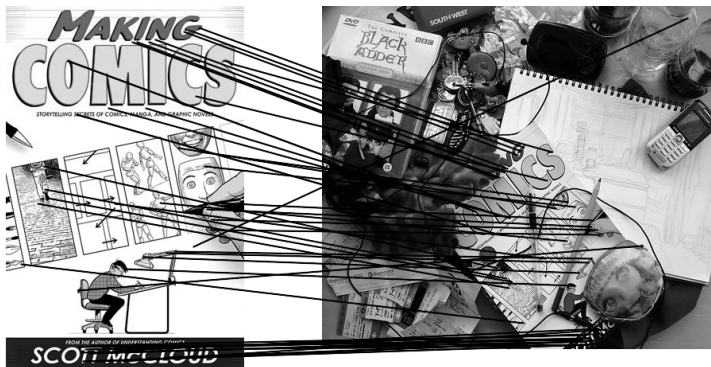


[Source: N. Snavely]

# Computing the (Affine) Transformation

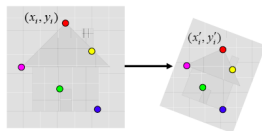
Given a set of matches between images  $I$  and  $J$

- How can we compute the affine transformation  $A$  from  $I$  to  $J$ ?
- Find transform  $A$  that best agrees with the matches



[Source: N. Snavely]

# Computing the Affine Transformation

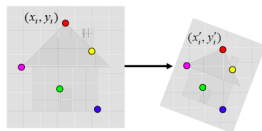


- Let  $(x_i, y_i)$  be a point on the reference (model) image, and  $(x'_i, y'_i)$  its match in the test image
- An affine transformation  $A$  maps  $(x_i, y_i)$  to  $(x'_i, y'_i)$ :

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \begin{bmatrix} a & b & e \\ c & d & f \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$



# Computing the Affine Transformation



- Let  $(x_i, y_i)$  be a point on the reference (model) image, and  $(x'_i, y'_i)$  its match in the test image
- An affine transformation  $A$  maps  $(x_i, y_i)$  to  $(x'_i, y'_i)$ :

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \begin{bmatrix} a & b & e \\ c & d & f \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

- We can rewrite this into a simple linear system:

$$\begin{bmatrix} x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} x'_i \\ y'_i \end{bmatrix}$$

# Computing the Affine Transformation

- But we have many matches:

$$\underbrace{\begin{bmatrix} \vdots \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ \vdots \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix}}_{\mathbf{P}'}$$

- For each match we have two more equations

# Computing the Affine Transformation

- But we have many matches:

$$\underbrace{\begin{bmatrix} \vdots & & & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ \vdots & & & & & \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix}}_{\mathbf{P}'}$$

- For each match we have two more equations
- How many matches do we need to compute  $A$ ?

# Computing the Affine Transformation

- But we have many matches:

$$\underbrace{\begin{bmatrix} \vdots & & & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ \vdots & & & & & \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix}}_{\mathbf{P}'}$$

- For each match we have two more equations
- How many matches do we need to compute  $A$ ?

# Computing the Affine Transformation

- But we have many matches:

$$\underbrace{\begin{bmatrix} \vdots & & & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ \vdots & & & & & \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix}}_{\mathbf{P}'}$$

- For each match we have two more equations
- How many matches do we need to compute  $\mathbf{A}$ ?
- 6 parameters  $\rightarrow$  3 matches
- But the more, the better (more reliable)
- How do we compute  $\mathbf{A}$ ?

# Computing the Affine Transformation

$$\underbrace{\begin{bmatrix} \vdots \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ \vdots \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix}}_{\mathbf{P}'}$$

- If we have 3 matches, then computing  $A$  is really easy:

$$\mathbf{a} = \mathbf{P}^{-1}\mathbf{P}'$$

# Computing the Affine Transformation

$$\underbrace{\begin{bmatrix} \vdots & & & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ \vdots & & & & & \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix}}_{\mathbf{P}'}$$

- If we have 3 matches, then computing  $A$  is really easy:

$$\mathbf{a} = \mathbf{P}^{-1}\mathbf{P}'$$

- If we have more than 3, then we do **least-squares estimation**:

$$\min_{a,b,\dots,f} \|\mathbf{Pa} - \mathbf{P}'\|_2^2$$

# Computing the Affine Transformation

$$\underbrace{\begin{bmatrix} & & \vdots & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ & & \vdots & & & \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix}}_{\mathbf{P}'}$$

- If we have 3 matches, then computing  $A$  is really easy:

$$\mathbf{a} = \mathbf{P}^{-1}\mathbf{P}'$$

- If we have more than 3, then we do **least-squares estimation**:

$$\min_{a,b,\dots,f} \|\mathbf{Pa} - \mathbf{P}'\|_2^2$$

- Which has a closed form solution:

$$\mathbf{a} = (\mathbf{P}^T \mathbf{P})^{-1} \mathbf{P}^T \mathbf{P}'$$



# Image Alignment Algorithm: Affine Case

Given images  $I$  and  $J$

- 1 Compute image features for  $I$  and  $J$
- 2 Match features between  $I$  and  $J$
- 3 Compute affine transformation  $A$  between  $I$  and  $J$  using least squares on the set of matches

# Image Alignment Algorithm: Affine Case

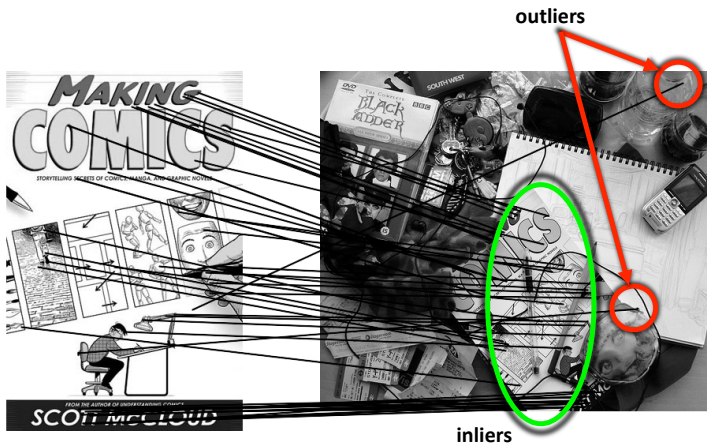
Given images  $I$  and  $J$

- 1 Compute image features for  $I$  and  $J$
- 2 Match features between  $I$  and  $J$
- 3 Compute affine transformation  $A$  between  $I$  and  $J$  using least squares on the set of matches

Is there a problem with this?

[Source: N. Snavely]

# Robustness

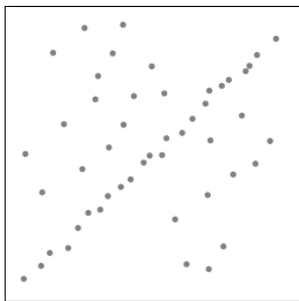


[Source: N. Snavely]

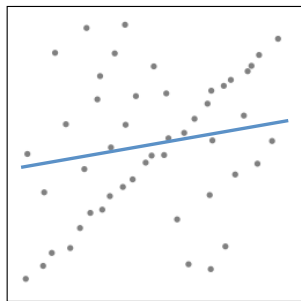
# Simple Case

(This example is unrelated to the object matching example, but it nicely shows how to robustify estimation)

- Lets consider a simpler example ... Fit a line to the points below!



Problem: Fit a line to these datapoints

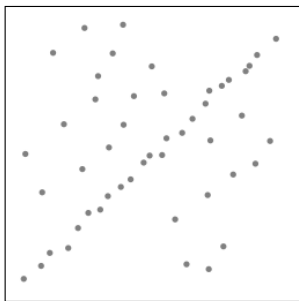


Least squares fit

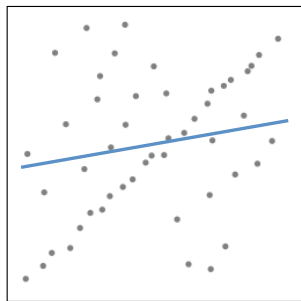
# Simple Case

(This example is unrelated to the object matching example, but it nicely shows how to robustify estimation)

- Lets consider a simpler example ... Fit a line to the points below!



Problem: Fit a line to these datapoints



Least squares fit

- How can we fix this?

[Source: N. Snavely]

# Simple Idea: RANSAC

- Take the minimal number of points to compute what we want. In the line example, two points (in our affine example, three matches)

# Simple Idea: RANSAC

- Take the minimal number of points to compute what we want. In the line example, two points (in our affine example, three matches)
- By “take” we mean choose at random from all points

# Simple Idea: RANSAC

- Take the minimal number of points to compute what we want. In the line example, two points (in our affine example, three matches)
- By “take” we mean choose at random from all points
- Fit a line to the selected pair of points



# Simple Idea: RANSAC

- Take the minimal number of points to compute what we want. In the line example, two points (in our affine example, three matches)
- By “take” we mean choose at random from all points
- Fit a line to the selected pair of points
- Count the number of all points that “agree” with the line: We call the agreeing points **inliers**

# Simple Idea: RANSAC

- Take the minimal number of points to compute what we want. In the line example, two points (in our affine example, three matches)
- By “take” we mean choose at random from all points
- Fit a line to the selected pair of points
- Count the number of all points that “agree” with the line: We call the agreeing points **inliers**
- “Agree” = within a small distance of the line

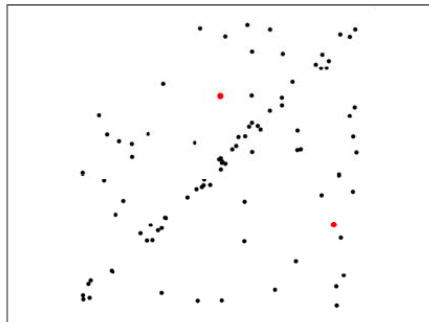
# Simple Idea: RANSAC

- Take the minimal number of points to compute what we want. In the line example, two points (in our affine example, three matches)
- By “take” we mean choose at random from all points
- Fit a line to the selected pair of points
- Count the number of all points that “agree” with the line: We call the agreeing points **inliers**
- “Agree” = within a small distance of the line
- Repeat this many times, remember the number of inliers for each trial
- Among several trials, select the one with the largest number of inliers

This procedure is called **RA**ndom **SA**mple **C**onsensus

# RANSAC for Line Fitting Example

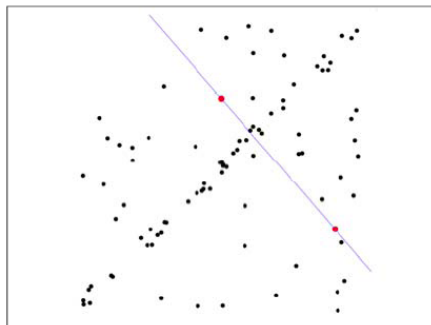
- 1 Randomly select minimal subset of points



[Source: R. Raguram]

# RANSAC for Line Fitting Example

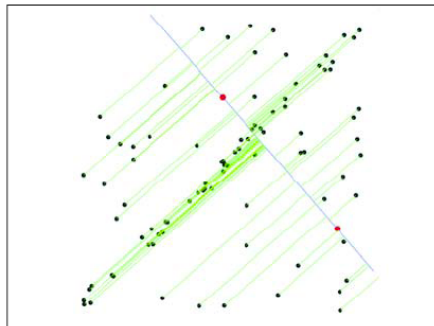
- 1 Randomly select minimal subset of points
- 2 Hypothesize a model



[Source: R. Raguram]

# RANSAC for Line Fitting Example

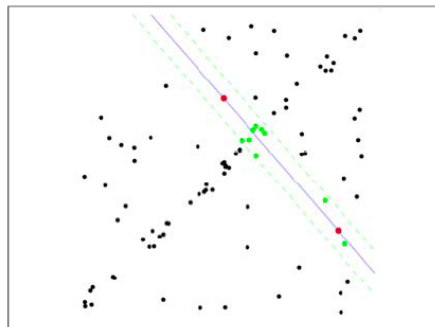
- 1 Randomly select minimal subset of points
- 2 Hypothesize a model
- 3 Compute error function



[Source: R. Raguram]

# RANSAC for Line Fitting Example

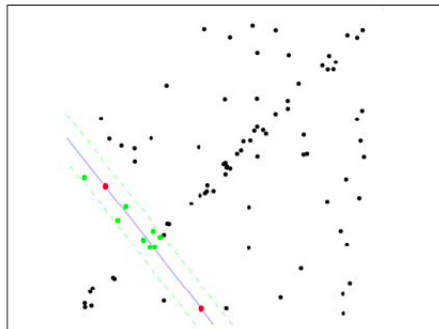
- 1 Randomly select minimal subset of points
- 2 Hypothesize a model
- 3 Compute error function
- 4 Select points consistent with model



[Source: R. Raguram]

# RANSAC for Line Fitting Example

- 1 Randomly select minimal subset of points
- 2 Hypothesize a model
- 3 Compute error function
- 4 Select points consistent with model
- 5 Repeat hypothesize and verify loop

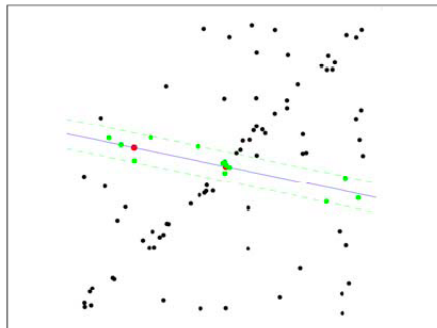


[Source: R. Raguram]



# RANSAC for Line Fitting Example

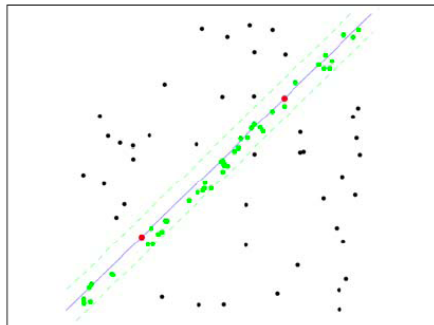
- 1 Randomly select minimal subset of points
- 2 Hypothesize a model
- 3 Compute error function
- 4 Select points consistent with model
- 5 Repeat hypothesize and verify loop



[Source: R. Raguram]

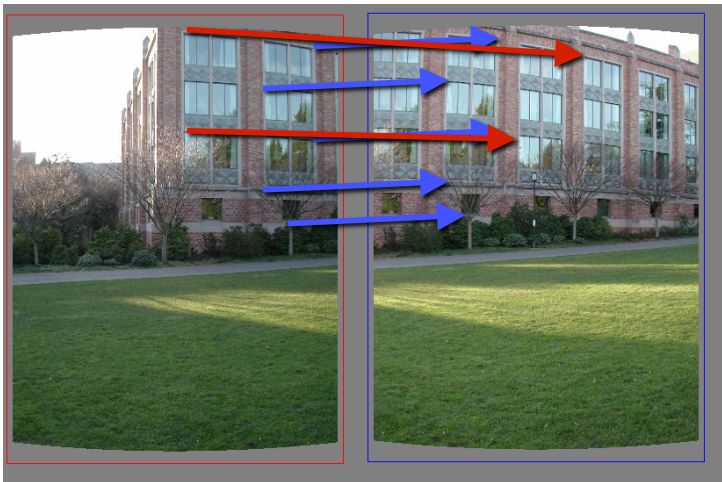
# RANSAC for Line Fitting Example

- 1 Randomly select minimal subset of points
- 2 Hypothesize a model
- 3 Compute error function
- 4 Select points consistent with model
- 5 Repeat hypothesize and verify loop
- 6 Choose model with largest set of inliers



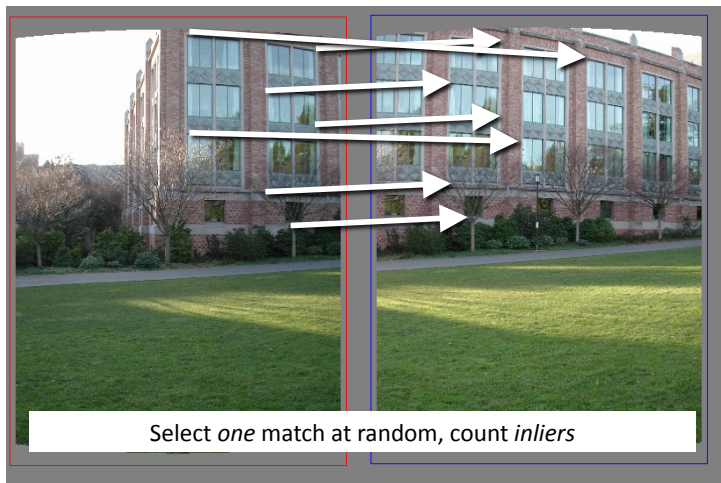
[Source: R. Raguram]

# Translations



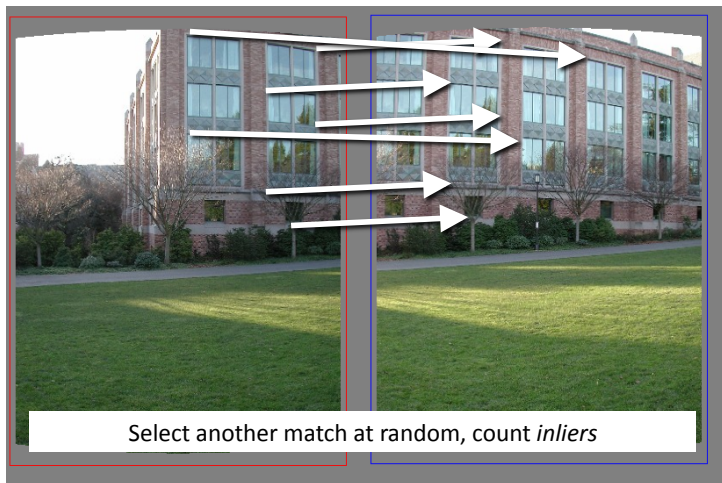
[Source: N. Snavely]

# RANdom SAmple Consensus



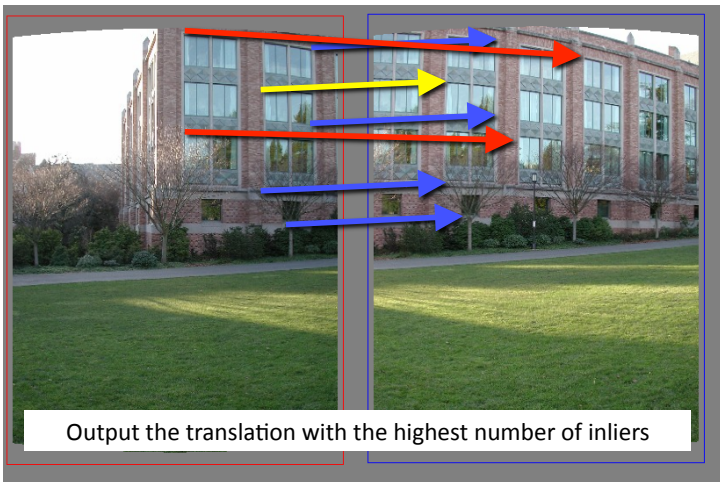
[Source: N. Snavely]

# RANdom SAmple Consensus



[Source: N. Snavely]

# RANdom SAmple Consensus



[Source: N. Snavely]

- 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

- 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



# RANSAC

- 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
- "All good matches are alike; every bad match is bad in its own way." – [Tolstoy via Alyosha Efros]

[Source: N. Snavely]

# Affine Transformation

How?

# Affine Transformation

How?

- Find matches across images  $I$  and  $J$ . This gives us points  $X_I$  in image  $I$  and  $X_J$  in  $J$ , where we know that the point  $X_I^k$  is a match with  $X_J^k$
- Iterate:
  - Choose 3 pairs of matches randomly
  - Compute the affine transformation
  - Project all matched points  $X_I$  from  $I$  to  $J$  via the computed transformation. This gives us  $\hat{X}_I$
  - Find how many matches are inliers, i.e.,  $\|\hat{X}_I^k - X_J^k\| < \text{thresh.}$
- Choose the transformation with the most inliers

- **Inlier threshold** related to the amount of noise we expect in inliers

- **Inlier threshold** related to the amount of noise we expect in inliers
- Often model noise as Gaussian with some standard deviation (e.g., 3 pixels)

- **Inlier threshold** related to the amount of noise we expect in inliers
- Often model noise as Gaussian with 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

- **Inlier threshold** related to the amount of noise we expect in inliers
- Often model noise as Gaussian with 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

- **Inlier threshold** related to the amount of noise we expect in inliers
- Often model noise as Gaussian with 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?

[Source: R. Urtasun]



# How many rounds?

- Sufficient number of trials  $S$  must be tried.

# How many rounds?

- Sufficient number of trials  $S$  must be tried.
- Let  $p$  be the probability that any given correspondence is valid and  $P$  be the total probability of success after  $S$  trials.

# How many rounds?

- Sufficient number of trials  $S$  must be tried.
- Let  $p$  be the probability that any given correspondence is valid and  $P$  be the total probability of success after  $S$  trials.
- The likelihood in one trial that all  $k$  random samples are inliers is  $p^k$

# How many rounds?

- Sufficient number of trials  $S$  must be tried.
- Let  $p$  be the probability that any given correspondence is valid and  $P$  be the total probability of success after  $S$  trials.
- The likelihood in one trial that all  $k$  random samples are inliers is  $p^k$
- The likelihood that  $S$  such trials will all fail is

$$1 - P = (1 - p^k)^S$$

# How many rounds?

- Sufficient number of trials  $S$  must be tried.
- Let  $p$  be the probability that any given correspondence is valid and  $P$  be the total probability of success after  $S$  trials.
- The likelihood in one trial that all  $k$  random samples are inliers is  $p^k$
- The likelihood that  $S$  such trials will all fail is

$$1 - P = (1 - p^k)^S$$

- The required minimum number of trials is

$$S = \frac{\log(1 - P)}{\log(1 - p^k)}$$

# How many rounds?

- Sufficient number of trials  $S$  must be tried.
- Let  $p$  be the probability that any given correspondence is valid and  $P$  be the total probability of success after  $S$  trials.
- The likelihood in one trial that all  $k$  random samples are inliers is  $p^k$
- The likelihood that  $S$  such trials will all fail is

$$1 - P = (1 - p^k)^S$$

- The required minimum number of trials is

$$S = \frac{\log(1 - P)}{\log(1 - p^k)}$$

- The number of trials grows quickly with the number of sample points used.

# How many rounds?

- Sufficient number of trials  $S$  must be tried.
- Let  $p$  be the probability that any given correspondence is valid and  $P$  be the total probability of success after  $S$  trials.
- The likelihood in one trial that all  $k$  random samples are inliers is  $p^k$
- The likelihood that  $S$  such trials will all fail is

$$1 - P = (1 - p^k)^S$$

- The required minimum number of trials is

$$S = \frac{\log(1 - P)}{\log(1 - p^k)}$$

- The number of trials grows quickly with the number of sample points used.

[Source: R. Urtasun]

# RANSAC pros and cons

## Pros

- Simple and general



# RANSAC pros and cons

## Pros

- Simple and general
- Applicable to many different problems

# RANSAC pros and cons

## Pros

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

# RANSAC pros and cons

## Pros

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

## Cons

- Parameters to tune

# 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

# 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

# RANSAC pros and cons

## Pros

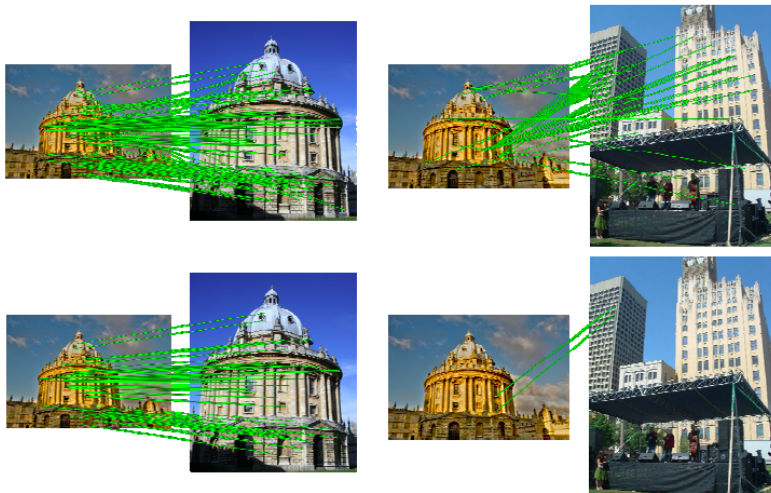
- Simple and general
- Applicable to many different problems
- Often works well in practice

## 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

[Source: N. Snavely, slide credit: R. Urtasun]

# Ransac Verification



[Source: K. Grauman, slide credit: R. Urtasun]

# Summary – Stuff You Need To Know

To match image  $I$  and  $J$  under affine transformation:

- Compute scale and rotation invariant keypoints in both images
- Compute a (rotation invariant) feature vector in each keypoint (e.g., SIFT)
- Match all features in  $I$  to all features in  $J$
- For each feature in reference image  $I$  find closest match in  $J$
- If ratio between closest and second closest match is  $< 0.8$ , keep match
- Do RANSAC to compute affine transformation  $A$ :
  - Select 3 matches at random
  - Compute  $A$
  - Compute the number of inliers
  - Repeat
  - Find  $A$  that gave the most inliers