Industrial Computer Vision

- Optical flow & Panorama stitching

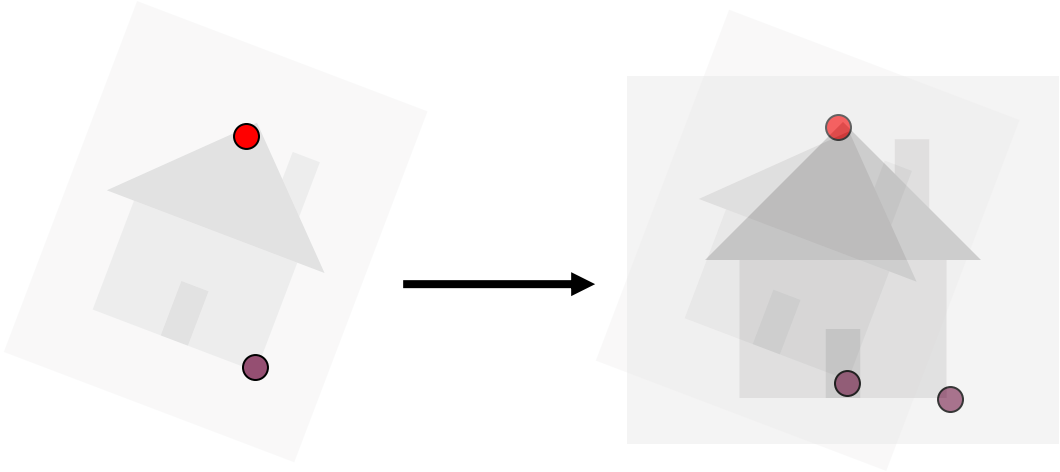
10th lecture, 2022.11.16

Lecturer: Youngbae Hwang

Contents

- Optical flow
  - What is Optical flow?
  - Aperture problems
  - Iterative approach
  - Coarse-to-fine approach
- Panorama stitching

Image Alignment



- How do we align two images automatically?
- Two broad approaches:
  - Feature-based alignment
    - Find a few matching features in both images
    - compute alignment
  - Direct (pixel-based) alignment
    - Search for alignment where most pixels agree

Direct Alignment

- The simplest approach is a brute force search (hw1)
  - Need to define image matching function
    - SSD, Normalized Correlation, edge matching, etc.
  - Search over all parameters within a reasonable range:

- e.g. for translation:

```
for tx=x0:step:x1,
  for ty=y0:step:y1,
     compare image1(x,y) to image2(x+tx,y+ty)
  end;
end;
```

- Need to pick correct `x0`, `x1` and `step`
  - What happens if `step` is too large?

Direct Alignment (brute force)

- What if we want to search for more complicated transformation, e.g. homography?

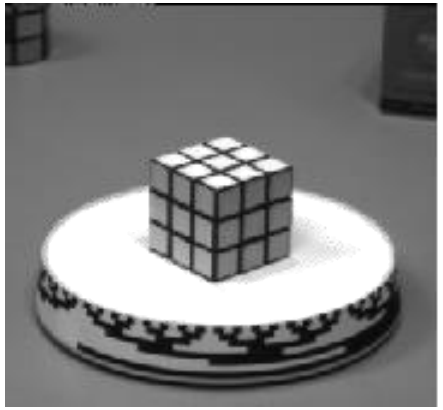
$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

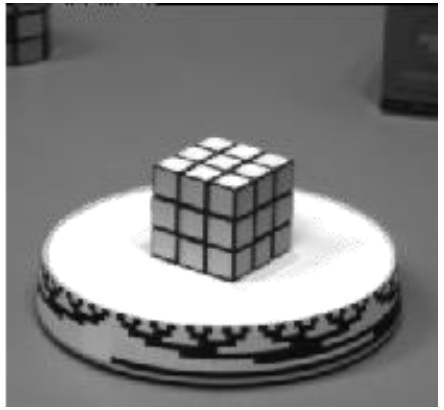
```
for a=a0:astep:a1,
  for b=b0:bstep:b1,
    for c=c0:cstep:c1,
      for d=d0:dstep:d1,
        for e=e0:estep:e1,
          for f=f0:fstep:f1,
            for g=g0:gstep:g1,
              for h=h0:hstep:h1,
     compare image1 to H(image2)
end; end; end; end; end; end; end; end;
```

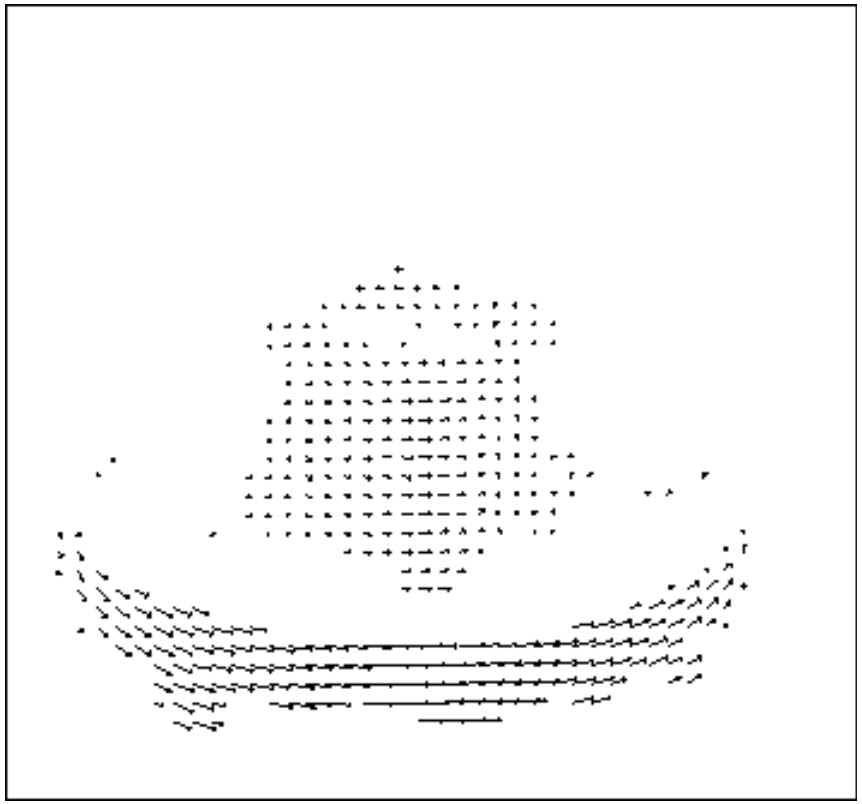
Problems with brute force

- Not realistic
  - Search in $O(N^8)$ is problematic
  - Not clear how to set starting/stopping value and step
- What can we do?
  - Use pyramid search to limit starting/stopping/step values
  - For special cases (rotational panoramas), can reduce search slightly to $O(N^4)$:
    - $H = K_1 R_1 R_2^{-1} K_2^{-1}$ (4 DOF: f and rotation)
- Alternative: gradient decent on the error function
  - i.e. how do I tweak my current estimate to make the SSD error go down?
  - Can do sub-pixel accuracy
  - BIG assumption?
    - Images are already almost aligned (<2 pixels difference!)
    - Can improve with pyramid
  - Same tool as in **motion estimation**

Motion estimation: Optical flow





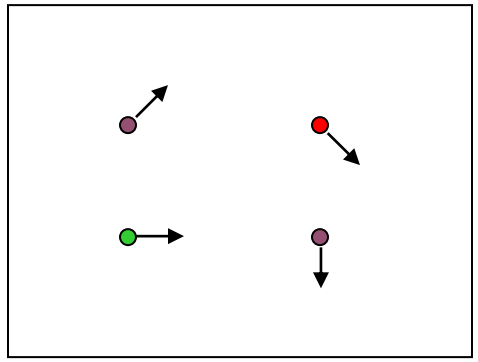


Will start by estimating motion of each pixel separately
Then will consider motion of entire image

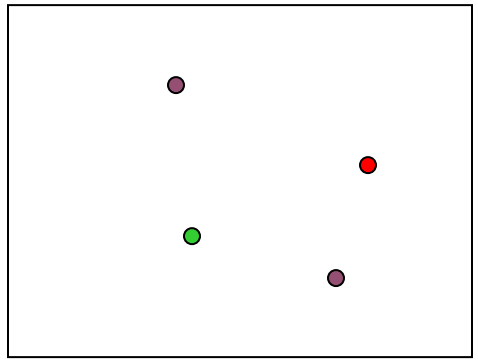
Why estimate motion?

- Lots of uses
  - Track object behavior
  - Correct for camera jitter (stabilization)
  - Align images (mosaics)
  - 3D shape reconstruction
  - Special effects

Problem definition: optical flow



$H(x, y)$



$I(x, y)$

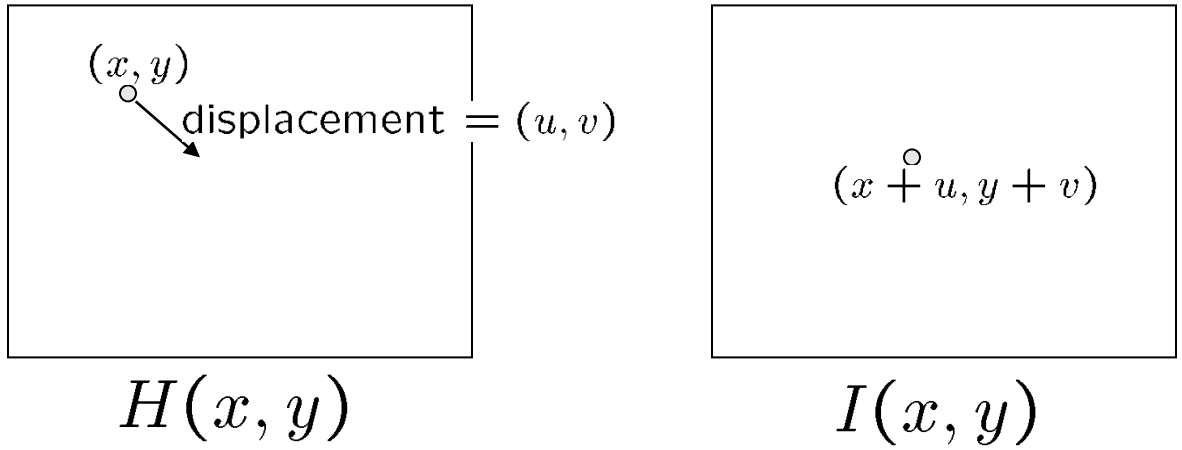
- How to estimate pixel motion from image H to image I?
  - Solve pixel correspondence problem
    - given a pixel in H, look for nearby pixels of the same color in I

Key assumptions

- color constancy: a point in H looks the same in I
  - For grayscale images, this is brightness constancy
- small motion: points do not move very far

This is called the optical flow problem

Optical flow constraints (grayscale images)



- Let's look at these constraints more closely
  - brightness constancy: Q: what's the equation?

    H(x,y)=I(x+u, y+v)

  - small motion: (u and v are less than 1 pixel)
    - suppose we take the Taylor series expansion of I:

$$I(x+u, y+v) = I(x, y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \text{higher order terms}$$

$$\approx I(x, y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v$$

Optical flow equation

- Combining these two equations

$$0 = I(x+u, y+v) - H(x,y)$$
$$\approx I(x,y) + I_x u + I_y v - H(x,y)$$
$$\approx (I(x,y) - H(x,y)) + I_x u + I_y v$$
$$\approx I_t + I_x u + I_y v$$
$$\approx I_t + \nabla I \cdot [u \ v]$$

shorthand: $I_x = \frac{\partial I}{\partial x}$

In the limit as u and v go to zero, this becomes exact

$$0 = I_t + \nabla I \cdot [\frac{\partial x}{\partial t} \ \frac{\partial y}{\partial t}]$$

Optical flow equation

$$0 = I_t + \nabla I \cdot [u \ v]$$

- Q: how many unknowns and equations per pixel?

  2 unknowns, one equation

Intuitively, what does this constraint mean?

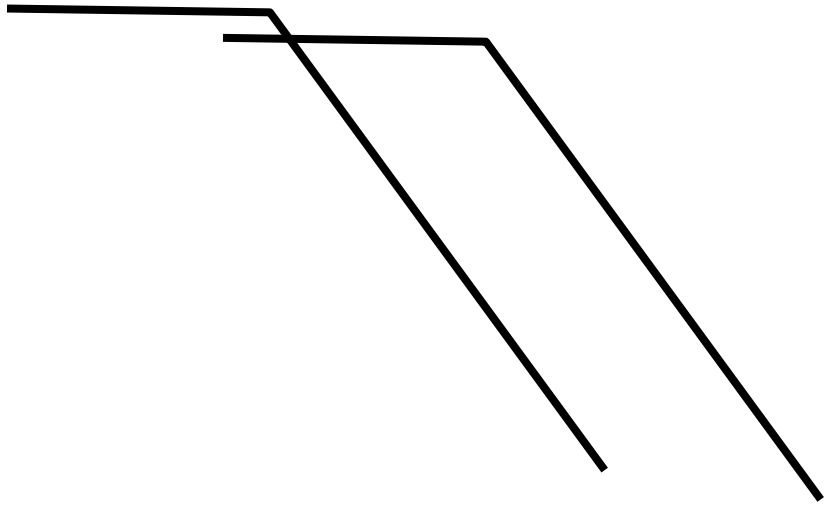
- The component of the flow in the gradient direction is determined
- The component of the flow parallel to an edge is unknown

This explains the Barber Pole illusion

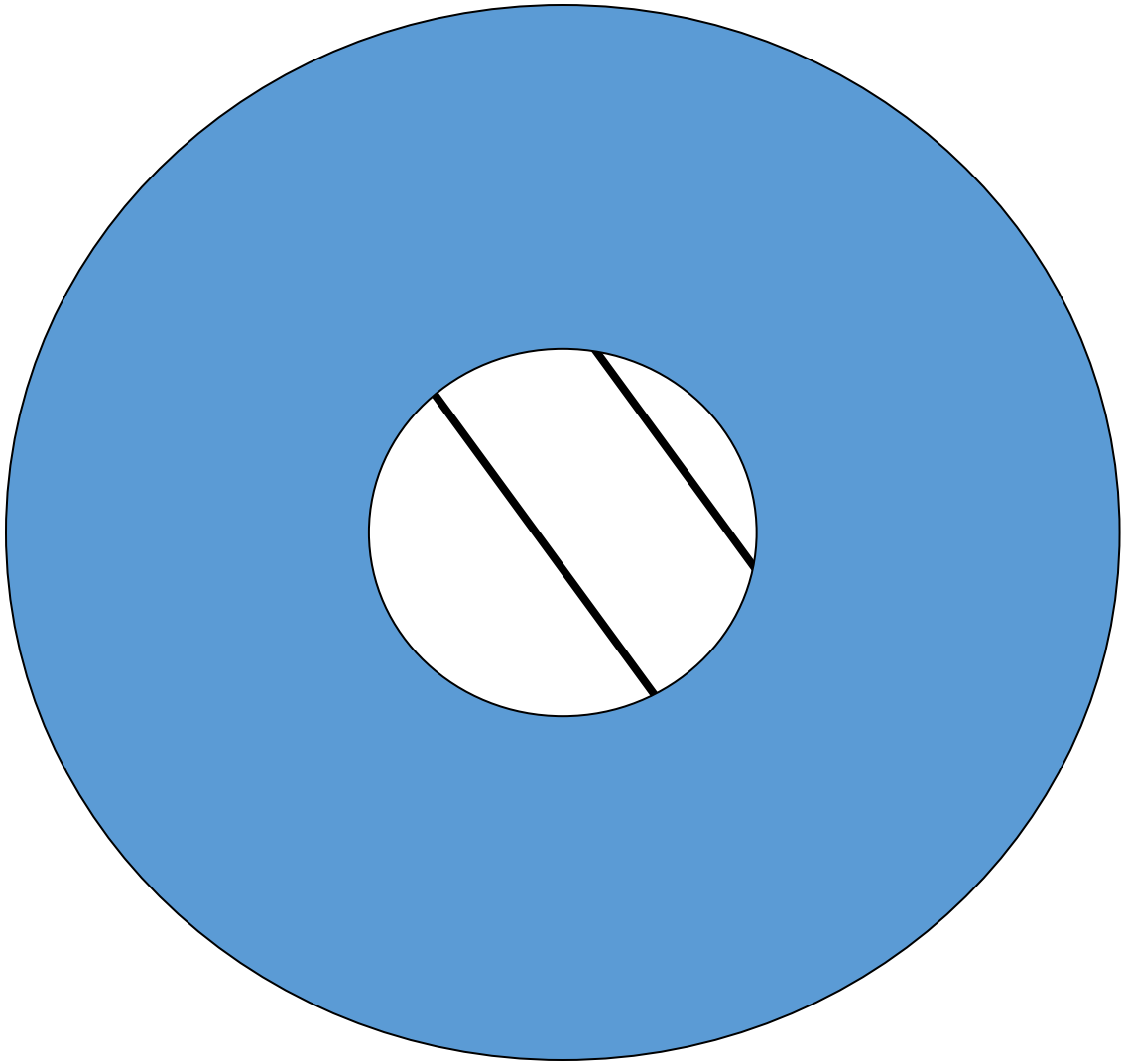
http://www.sandlotscience.com/Ambiguous/Barberpole_Illusion.htm
http://www.liv.ac.uk/~marcob/Trieste/barberpole.html

http://en.wikipedia.org/wiki/Barber's_pole

Aperture problem



Aperture problem



Solving the aperture problem

- How to get more equations for a pixel?
  - Basic idea: impose additional constraints
    - most common is to assume that the flow field is smooth locally
    - one method: pretend the pixel's neighbors have the same (u,v)
      - If we use a 5x5 window, that gives us 25 equations per pixel!

$$0 = I_t(\mathbf{p_i}) + \nabla I(\mathbf{p_i}) \cdot [u\ v]$$

$$\underbrace{\begin{bmatrix} I_x(\mathbf{p_1}) & I_y(\mathbf{p_1}) \\ I_x(\mathbf{p_2}) & I_y(\mathbf{p_2}) \\ \vdots & \vdots \\ I_x(\mathbf{p_{25}}) & I_y(\mathbf{p_{25}}) \end{bmatrix}}_{A\ 25\times2} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{d\ 2\times1} = -\underbrace{\begin{bmatrix} I_t(\mathbf{p_1}) \\ I_t(\mathbf{p_2}) \\ \vdots \\ I_t(\mathbf{p_{25}}) \end{bmatrix}}_{b\ 25\times1}$$

RGB version

- How to get more equations for a pixel?
  - Basic idea: impose additional constraints
    - most common is to assume that the flow field is smooth locally
    - one method: pretend the pixel's neighbors have the same (u,v)
      - If we use a 5x5 window, that gives us 25*3 equations per pixel!

$$0 = I_t(\mathbf{p_i})[0,1,2] + \nabla I(\mathbf{p_i})[0,1,2] \cdot [u\ v]$$

$$\underbrace{\begin{bmatrix} I_x(\mathbf{p_1})[0] & I_y(\mathbf{p_1})[0] \\ I_x(\mathbf{p_1})[1] & I_y(\mathbf{p_1})[1] \\ I_x(\mathbf{p_1})[2] & I_y(\mathbf{p_1})[2] \\ \vdots & \vdots \\ I_x(\mathbf{p_{25}})[0] & I_y(\mathbf{p_{25}})[0] \\ I_x(\mathbf{p_{25}})[1] & I_y(\mathbf{p_{25}})[1] \\ I_x(\mathbf{p_{25}})[2] & I_y(\mathbf{p_{25}})[2] \end{bmatrix}}_{A\ 75\times2} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{d\ 2\times1} = -\underbrace{\begin{bmatrix} I_t(\mathbf{p_1})[0] \\ I_t(\mathbf{p_1})[1] \\ I_t(\mathbf{p_1})[2] \\ \vdots \\ I_t(\mathbf{p_{25}})[0] \\ I_t(\mathbf{p_{25}})[1] \\ I_t(\mathbf{p_{25}})[2] \end{bmatrix}}_{b\ 75\times1}$$

Note that RGB is not enough to disambiguate because R, G & B are correlated
Just provides better gradient

Lukas-Kanade flow

- Prob: we have more equations than unknowns

$$\underset{25\text{x}2}{A}\ \underset{2\text{x}1}{d} = \underset{25\text{x}1}{b} \longrightarrow \text{minimize } \|Ad - b\|^2$$

Solution: solve least squares problem

- minimum least squares solution given by solution (in d) of:

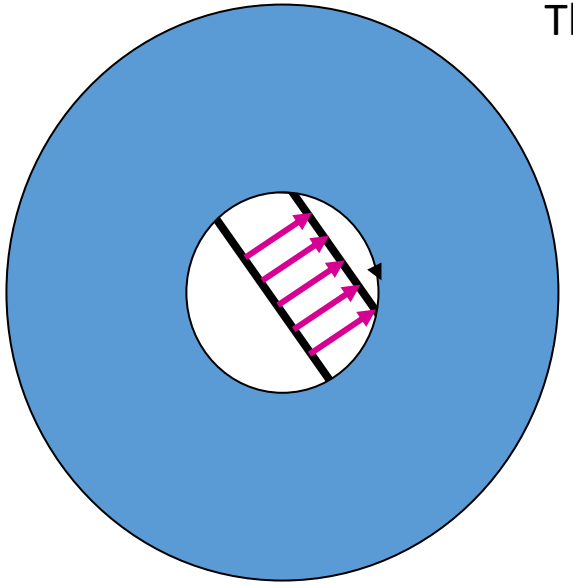
$$\underset{2\text{x}2}{(A^T A)}\ \underset{2\text{x}1}{d} = \underset{2\text{x}1}{A^T b}$$

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

$A^T A$ $A^T b$

- The summations are over all pixels in the K x K window
- This technique was first proposed by Lukas & Kanade (1981)

Aperture Problem and Normal Flow

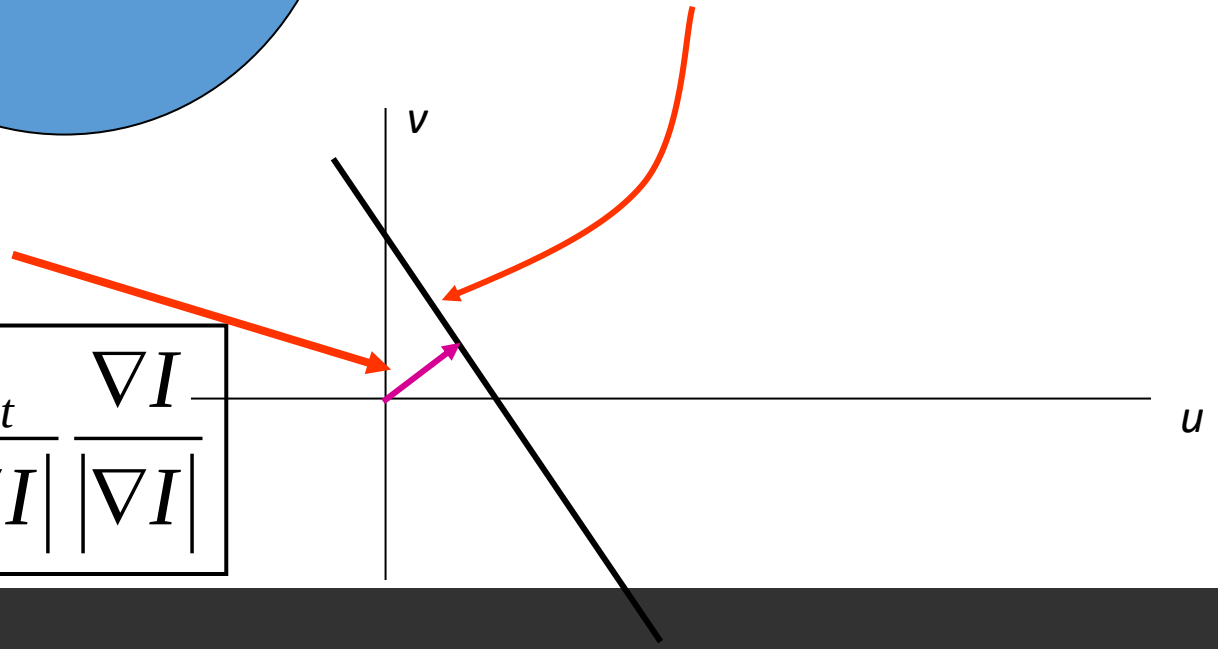


The gradient constraint:

$$I_x u + I_y v + I_t = 0$$

$$\nabla I \bullet \vec{U} = 0$$

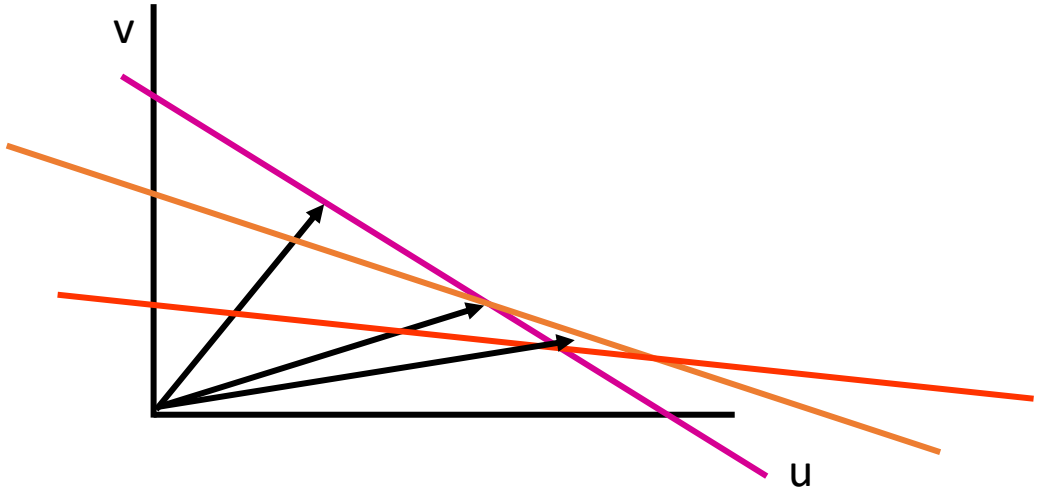
Defines a line in the *(u,v)* space



Normal Flow:

$$u_\perp = -\frac{I_t}{|\nabla I|}\frac{\nabla I}{|\nabla I|}$$

Combining Local Constraints



$$\nabla I^1 \bullet U = -I_t^1$$

$$\nabla I^2 \bullet U = -I_t^2$$

$$\nabla I^3 \bullet U = -I_t^3$$

etc.

Conditions for solvability

- Optimal (u, v) satisfies Lucas-Kanade equation

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

$$A^T A \qquad\qquad A^T b$$

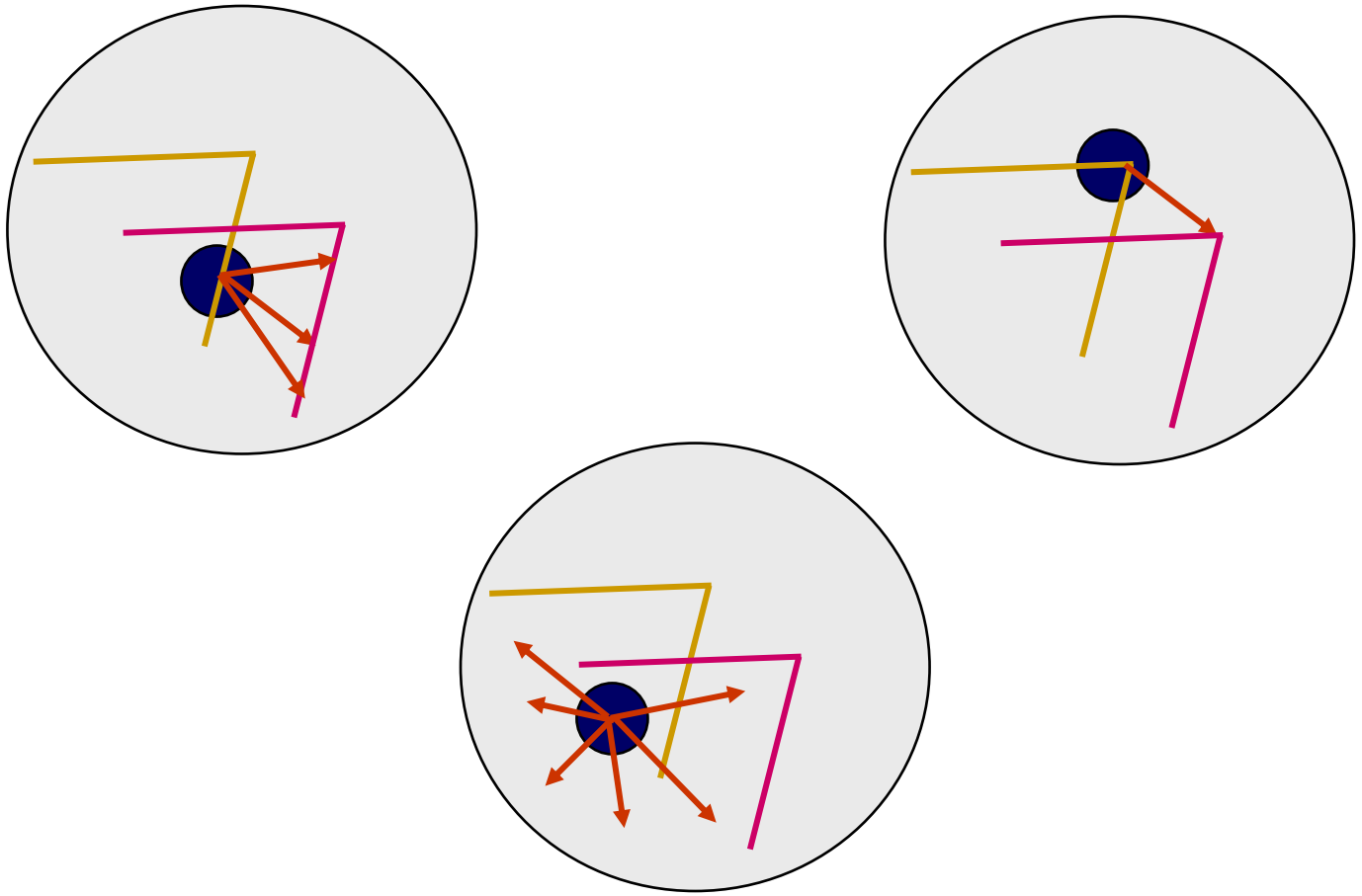
When is This Solvable?

- A^TA should be invertible
- A^TA should not be too small due to noise
  - eigenvalues λ_1 and λ_2 of A^TA should not be too small
- A^TA should be well-conditioned
  - λ_1 / λ_2 should not be too large (λ_1 = larger eigenvalue)

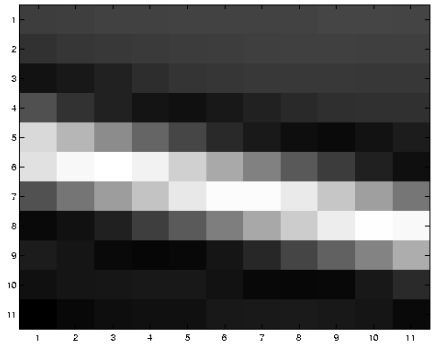
A^TA is solvable when there is no aperture problem

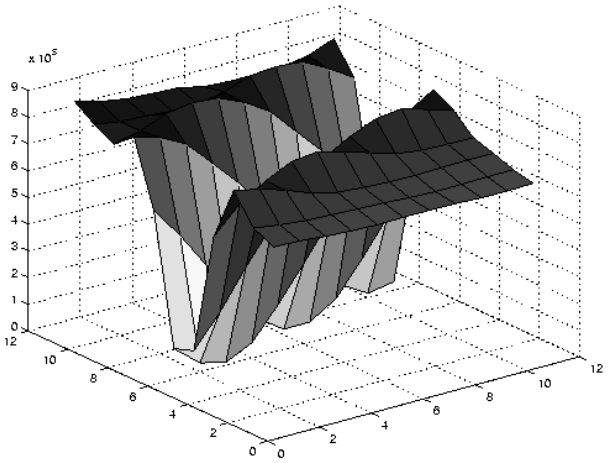
$$A^T A = \begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} = \sum \begin{bmatrix} I_x \\ I_y \end{bmatrix} [I_x \; I_y] = \sum \nabla I (\nabla I)^T$$

Local Patch Analysis



Edge



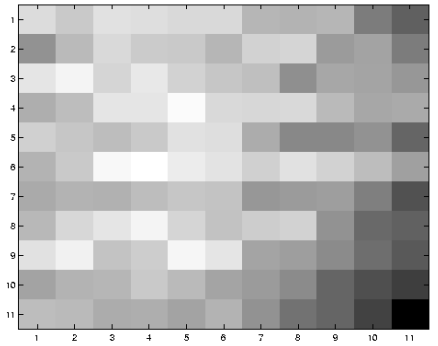


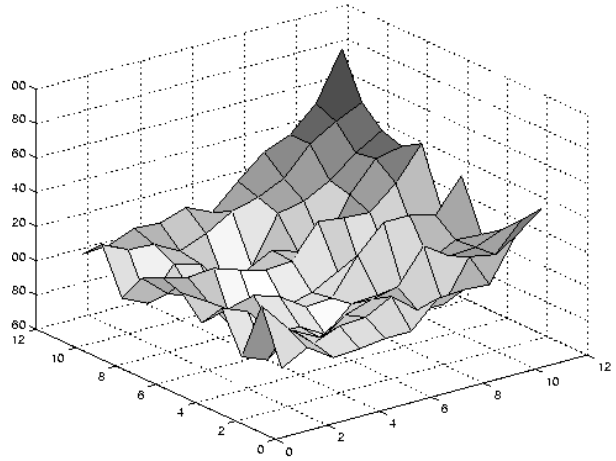
$$\sum \nabla I (\nabla I)^T$$

– large gradients, all the same

– large λ_1, small λ_2

Low texture region



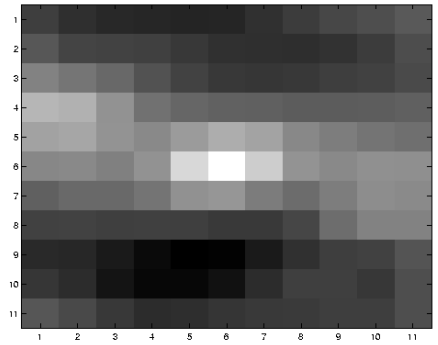


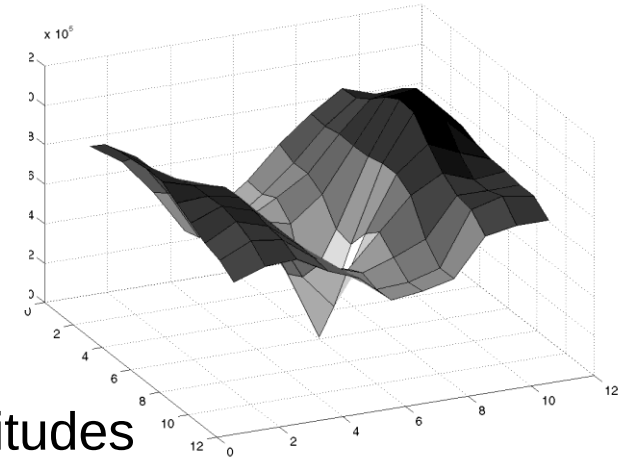
$$\sum \nabla I (\nabla I)^T$$

– gradients have small magnitude

– small λ_1, small λ_2

High textured region





$$\sum \nabla I (\nabla I)^T$$

– gradients are different, large magnitudes

– large λ_1, large λ_2

Observation

- This is a two image problem BUT
  - Can measure sensitivity by just looking at one of the images!
  - This tells us which pixels are easy to track, which are hard
    - very useful later on when we do feature tracking...

Errors in Lukas-Kanade

- What are the potential causes of errors in this procedure?
  - Suppose A^TA is easily invertible
  - Suppose there is not much noise in the image

When our assumptions are violated

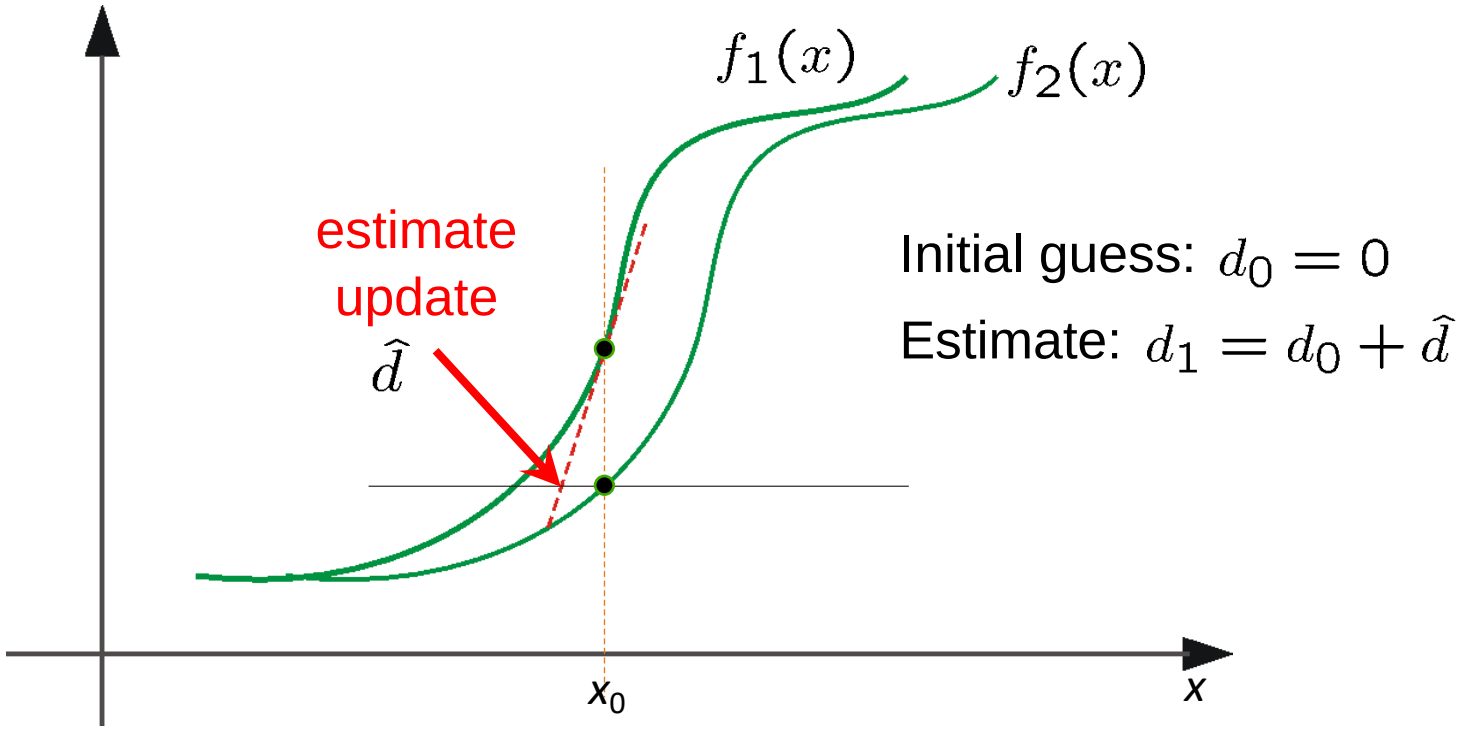
- Brightness constancy is not satisfied
- The motion is not small
- A point does not move like its neighbors
  - window size is too large
  - what is the ideal window size?

Iterative Refinement

Iterative Lukas-Kanade Algorithm

1. Estimate velocity at each pixel by solving Lucas-Kanade equations
2. Warp H towards I using the estimated flow field
   - *use image warping techniques*
3. Repeat until convergence

Optical Flow: Iterative Estimation

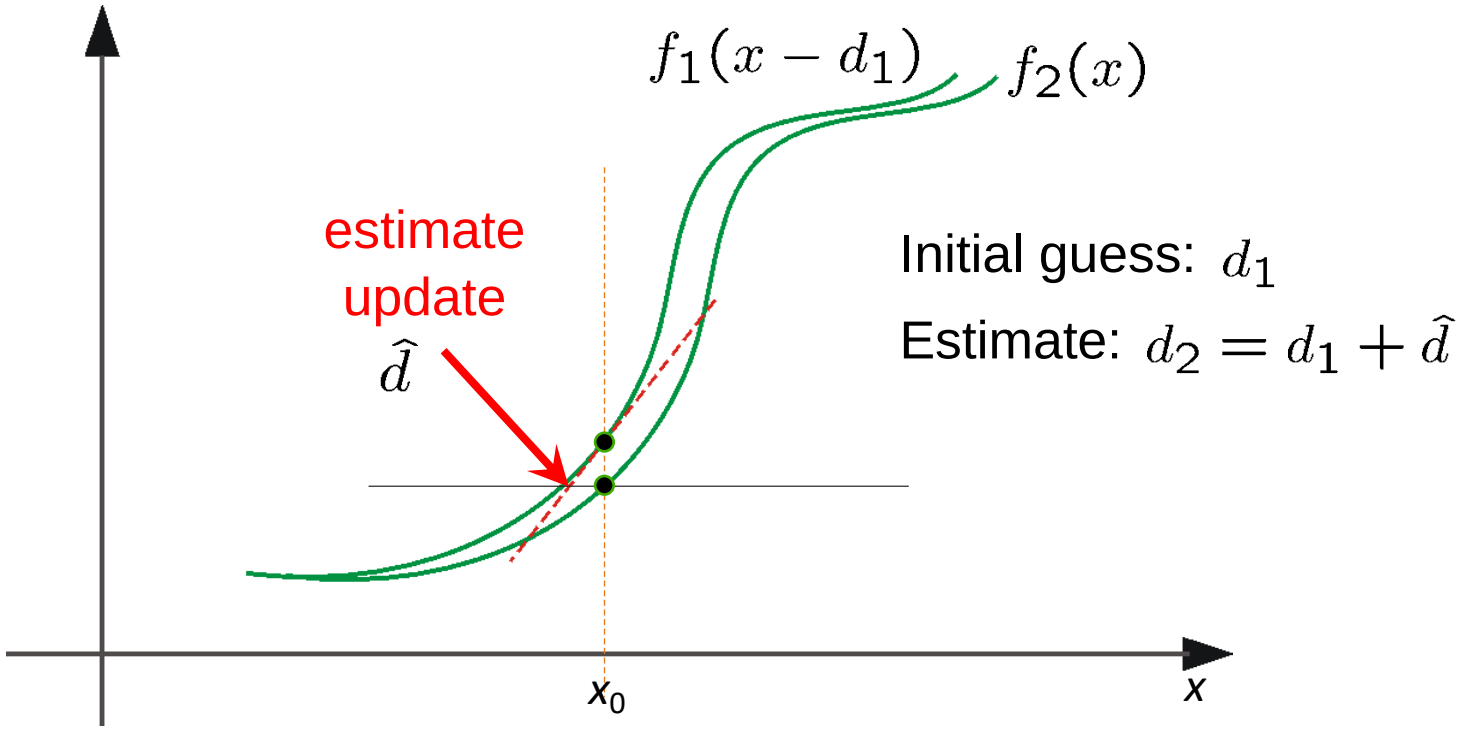


Initial guess: $d_0 = 0$

Estimate: $d_1 = d_0 + \hat{d}$

(using *d* for *displacement* here instead of *u*)

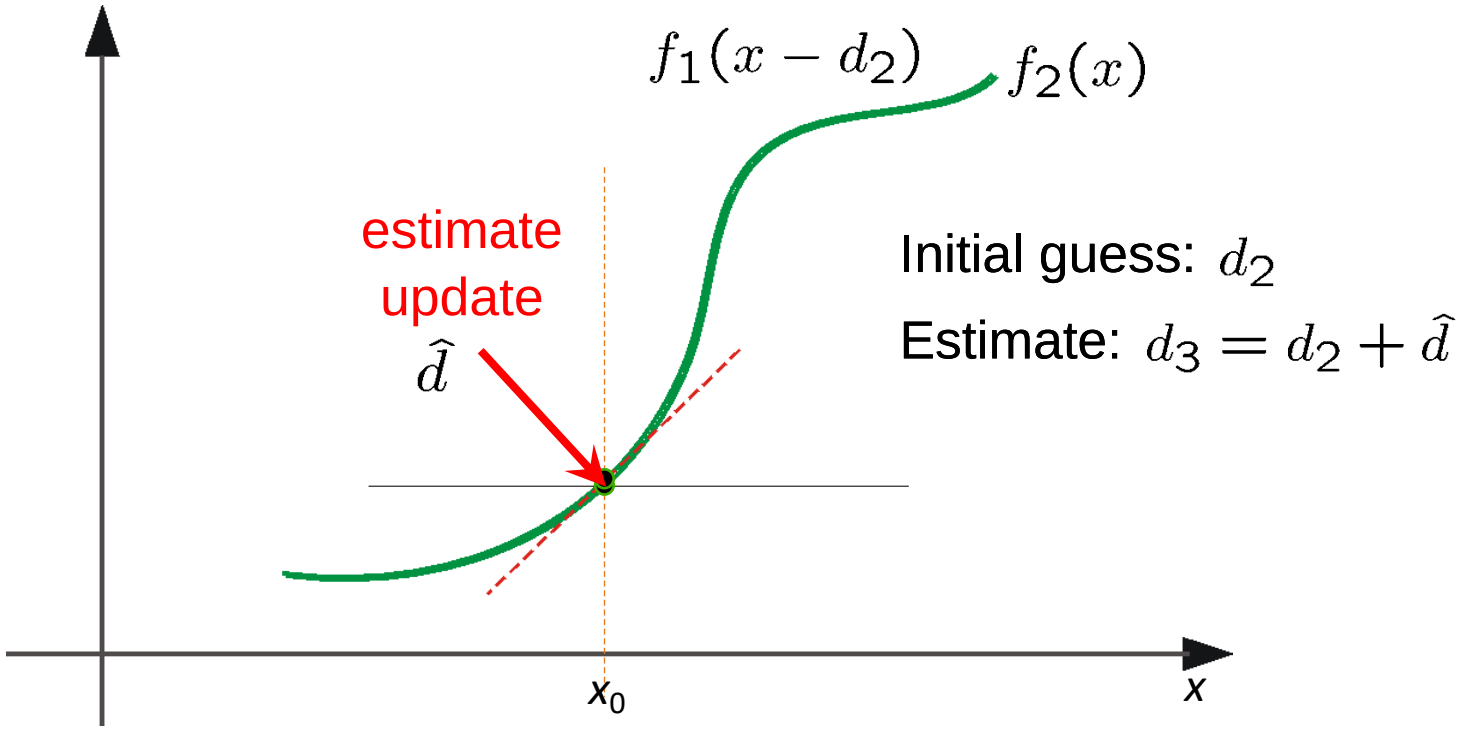
Optical Flow: Iterative Estimation



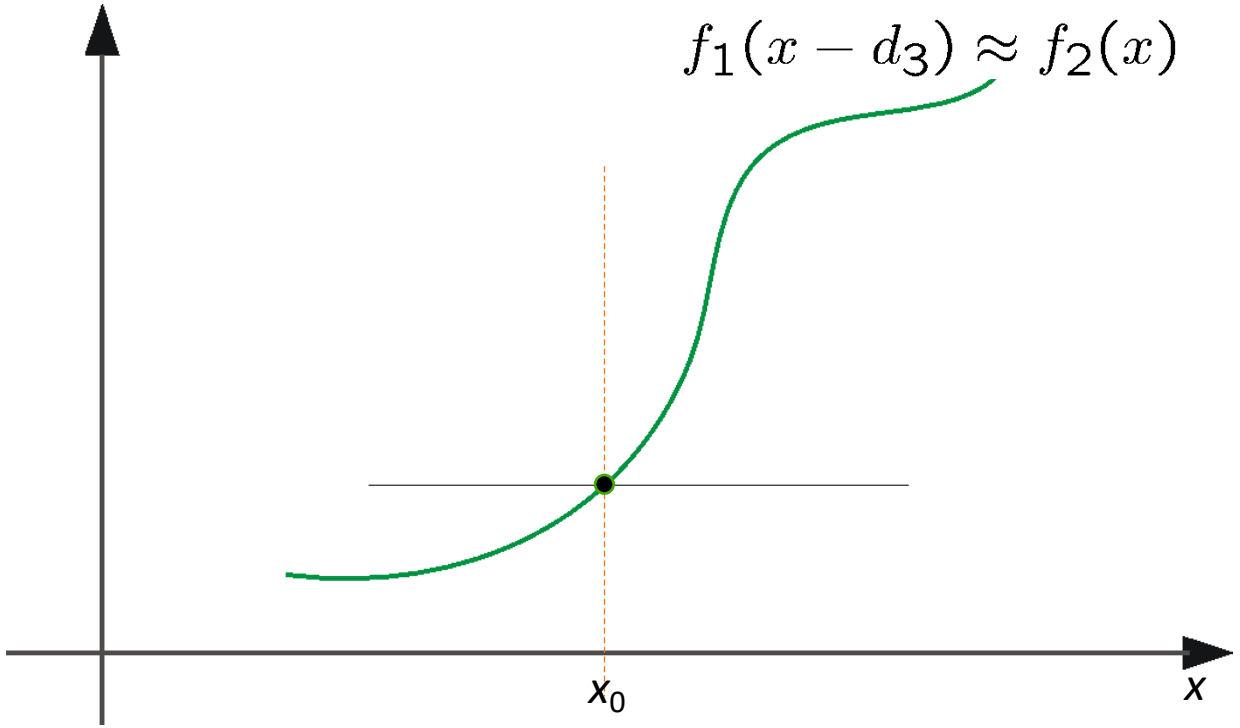
Initial guess: d_1

Estimate: $d_2 = d_1 + \hat{d}$

Optical Flow: Iterative Estimation



Optical Flow: Iterative Estimation



Optical Flow: Iterative Estimation

- Some Implementation Issues:
  - Warping is not easy (ensure that errors in warping are smaller than the estimate refinement)
  - Warp one image, take derivatives of the other so you don't need to re-compute the gradient after each iteration.
  - Often useful to low-pass filter the images before motion estimation (for better derivative estimation, and linear approximations to image intensity)

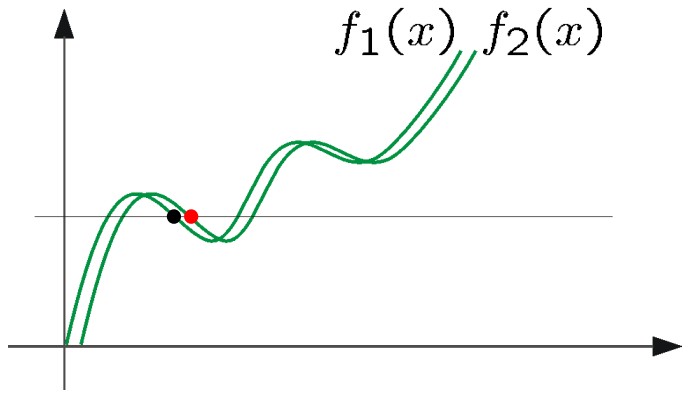
Revisiting the small motion assumption

- Is this motion small enough?
  - Probably not—it's much larger than one pixel (2nd order terms dominate)
  - How might we solve this problem?

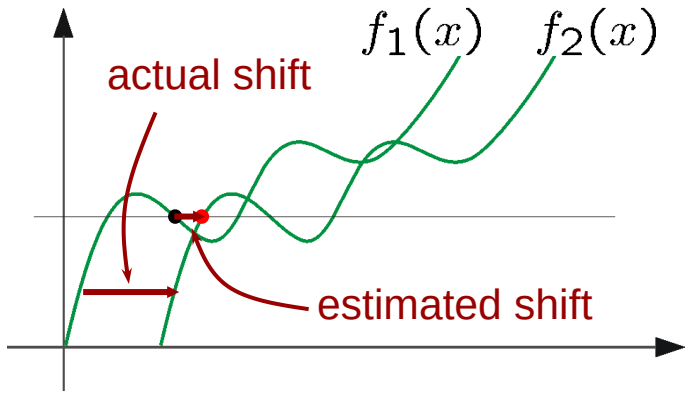
Optical Flow: Aliasing

Temporal aliasing causes ambiguities in optical flow because images can have many pixels with the same intensity.

I.e., how do we know which ‘correspondence’ is correct?



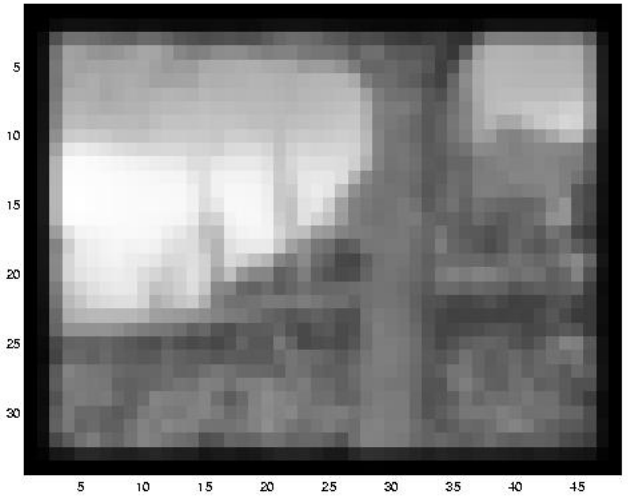
nearest match is correct
(no aliasing)

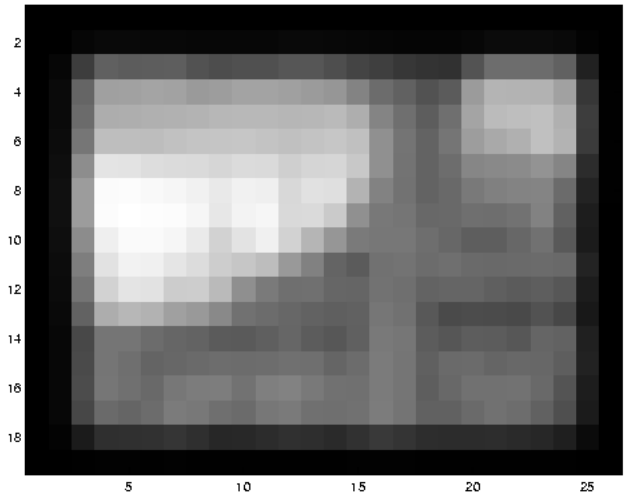


nearest match is incorrect
(aliasing)

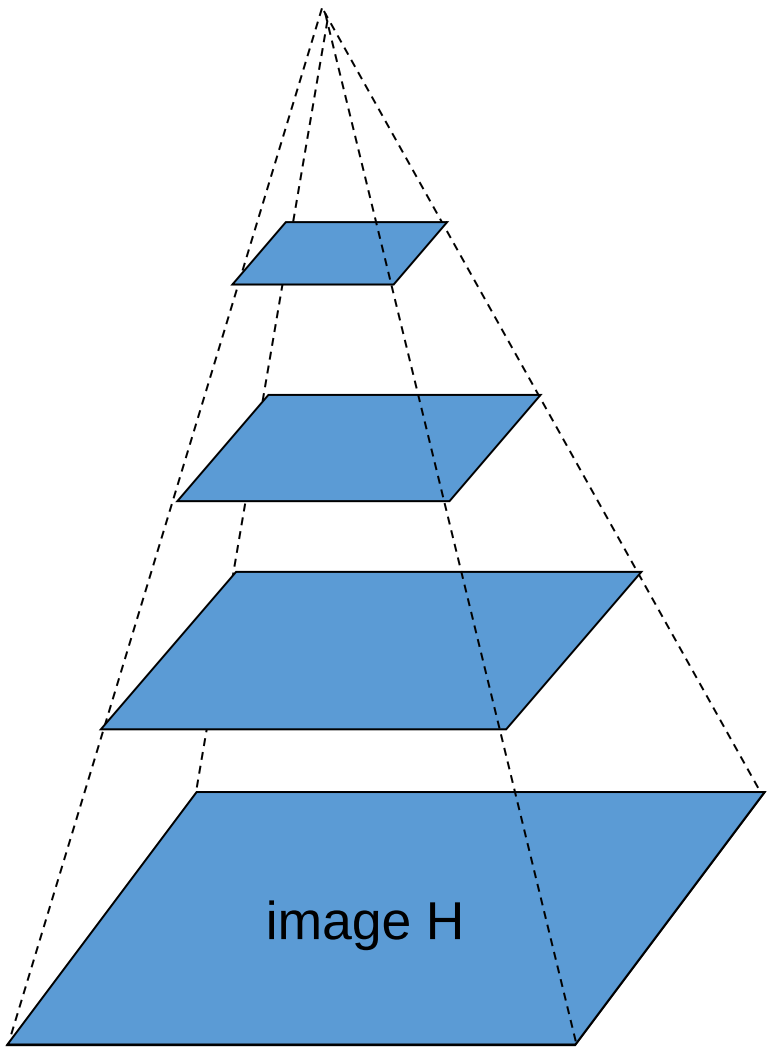
To overcome aliasing: coarse-to-fine estimation.

Reduce the resolution!





Coarse-to-fine optical flow estimation

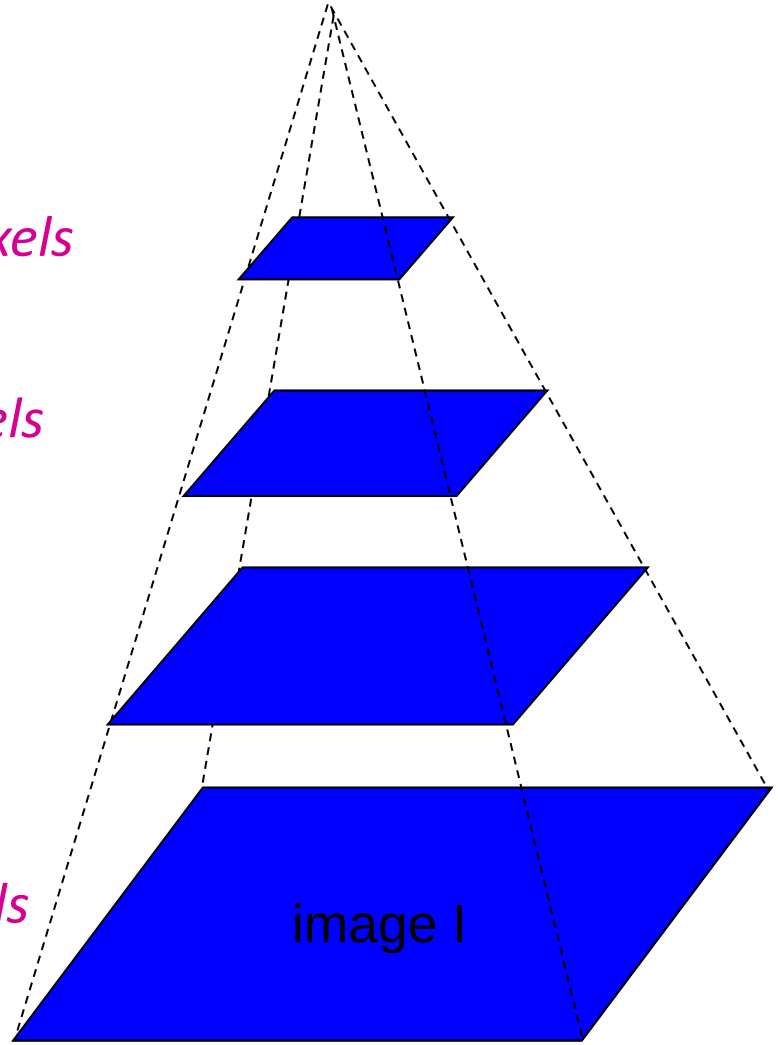


u=1.25 pixels

u=2.5 pixels

u=5 pixels

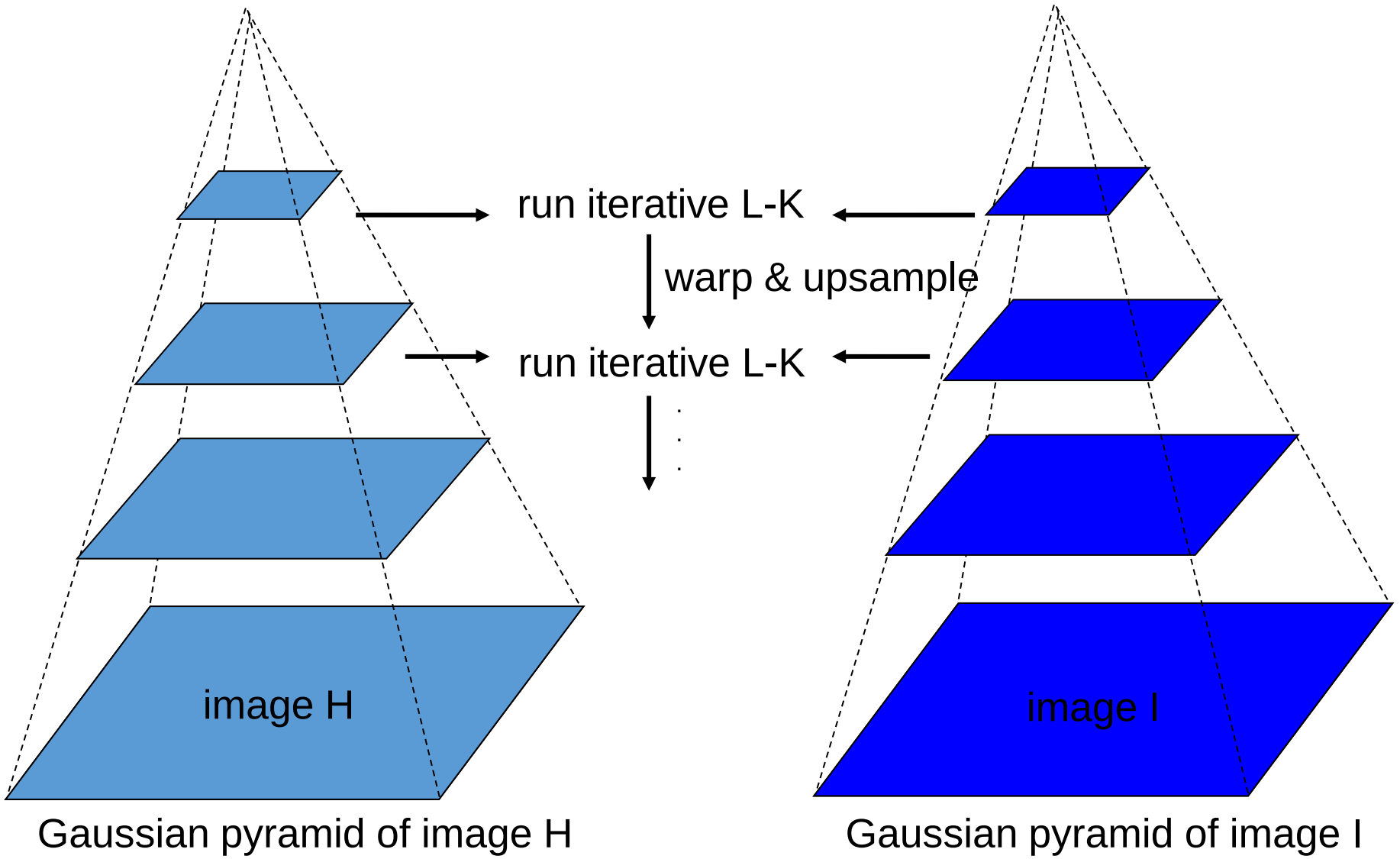
u=10 pixels



Gaussian pyramid of image H

Gaussian pyramid of image I

Coarse-to-fine optical flow estimation



Beyond Translation

- So far, our patch can only translate in (u,v)
- What about other motion models?
  - rotation, affine, perspective

- Same thing but need to add an appropriate Jacobian
- See Szeliski's survey of Panorama stitching

$$\mathbf{A}^{\mathbf{T}}\mathbf{A} = \sum_i \mathbf{J}\nabla\mathbf{I}(\nabla\mathbf{I})^{\mathbf{T}}\mathbf{J}^{\mathbf{T}}$$

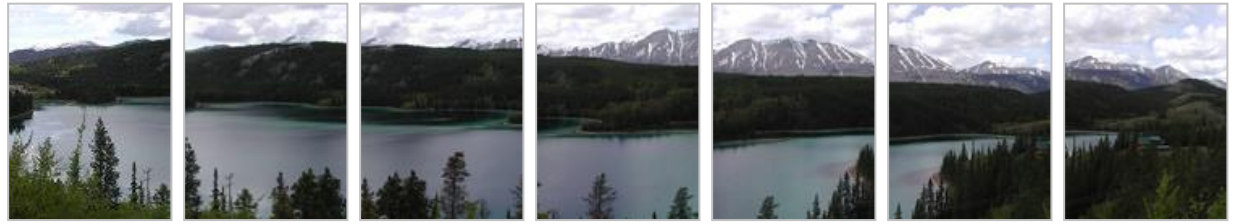
$$\mathbf{A}^{\mathbf{T}}\mathbf{b} = -\sum_i \mathbf{J}^{\mathbf{T}} I_t (\nabla\mathbf{I})^{\mathbf{T}}$$

Recap: Classes of Techniques

- ***Feature-based methods (e.g. SIFT+Ransac+regression)***
  - Extract visual features (corners, textured areas) and track them over multiple frames
  - Sparse motion fields, but possibly robust tracking
  - Suitable especially when image motion is large (10-s of pixels)

- ***Direct-methods (e.g. optical flow)***
  - Directly recover image motion from spatio-temporal image brightness variations
  - Global motion parameters directly recovered without an intermediate feature motion calculation
  - Dense motion fields, but more sensitive to appearance variations
  - Suitable for video and when image motion is small (< 10 pixels)

Combine two or more overlapping images to make one larger image



Slide credit: Vaibhav Vaish

How to do it?

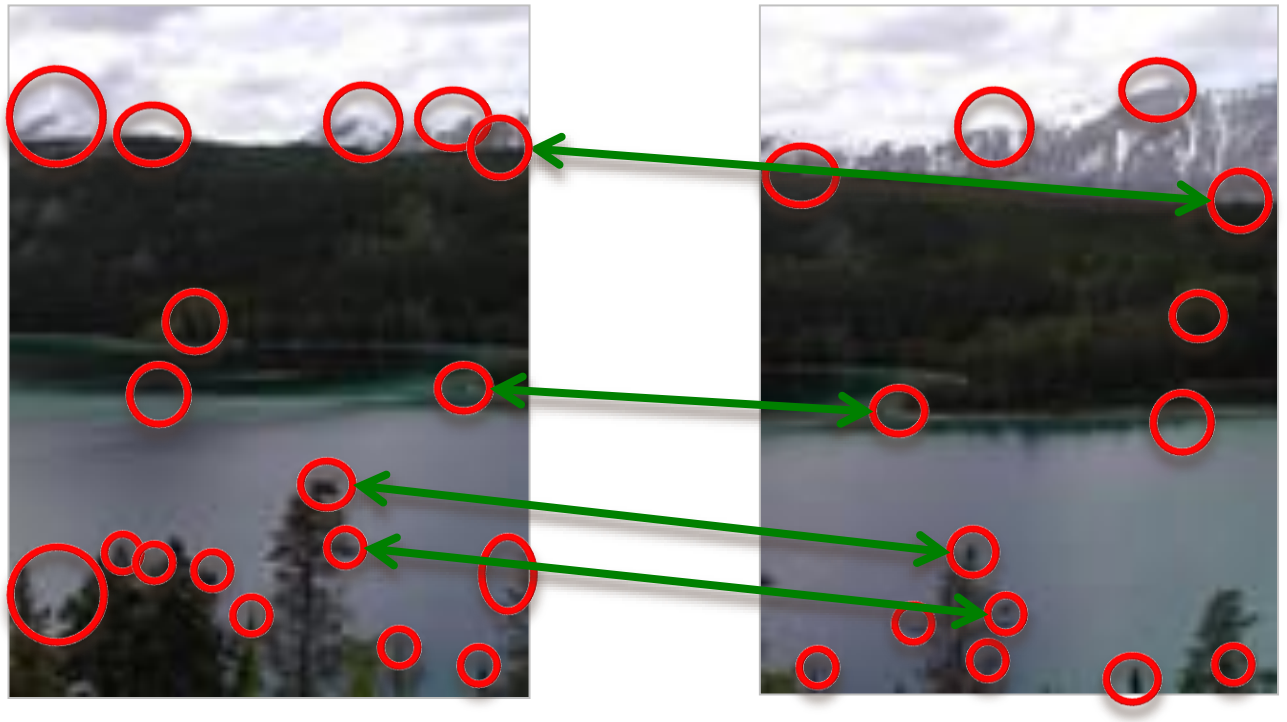
- Basic Procedure
  - Take a sequence of images from the same position
    - Rotate the camera about its optical center
  - Compute transformation between second image and first
  - Shift the second image to overlap with the first
  - Blend the two together to create a mosaic
  - If there are more images, repeat

1. Take a sequence of images from the same position

- Rotate the camera about its optical center

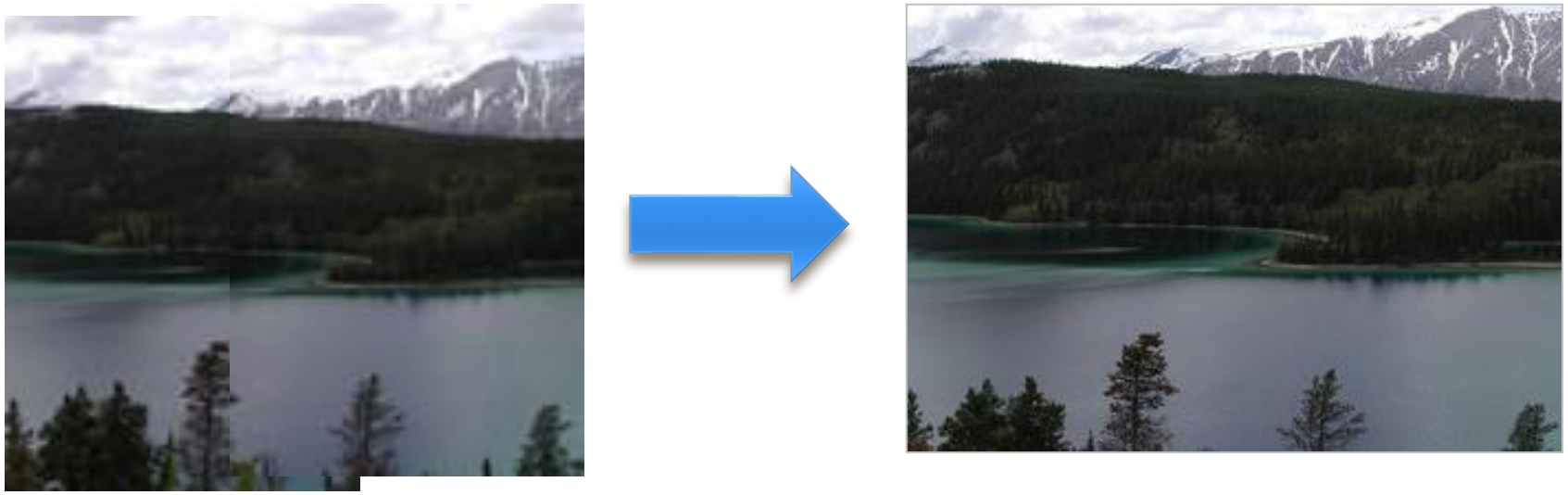
2. Compute transformation between images

- Extract interest points
- Find Matches
- Compute transformation

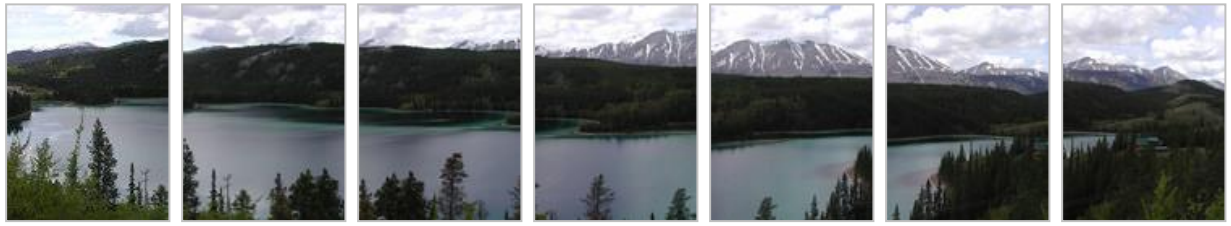


3. Shift the images to overlap

4. Blend the two together to create a mosaic



5. Repeat for all images



How to do it?

- Basic Procedure
  - Take a sequence of images from the same position
    - Rotate the camera about its optical center
  - Compute transformation between second image and first
  - Shift the second image to overlap with the first
  - Blend the two together to create a mosaic
  - If there are more images, repeat

Compute Transformations

- Extract interest points
- Find good matches ✓
- Compute transformation ✓

Let’s assume we are given a set of good matching inter
est points

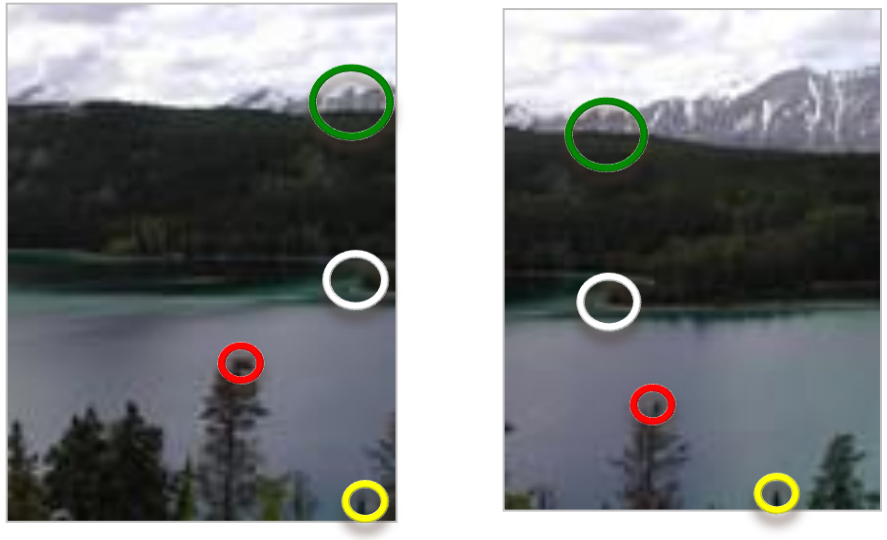
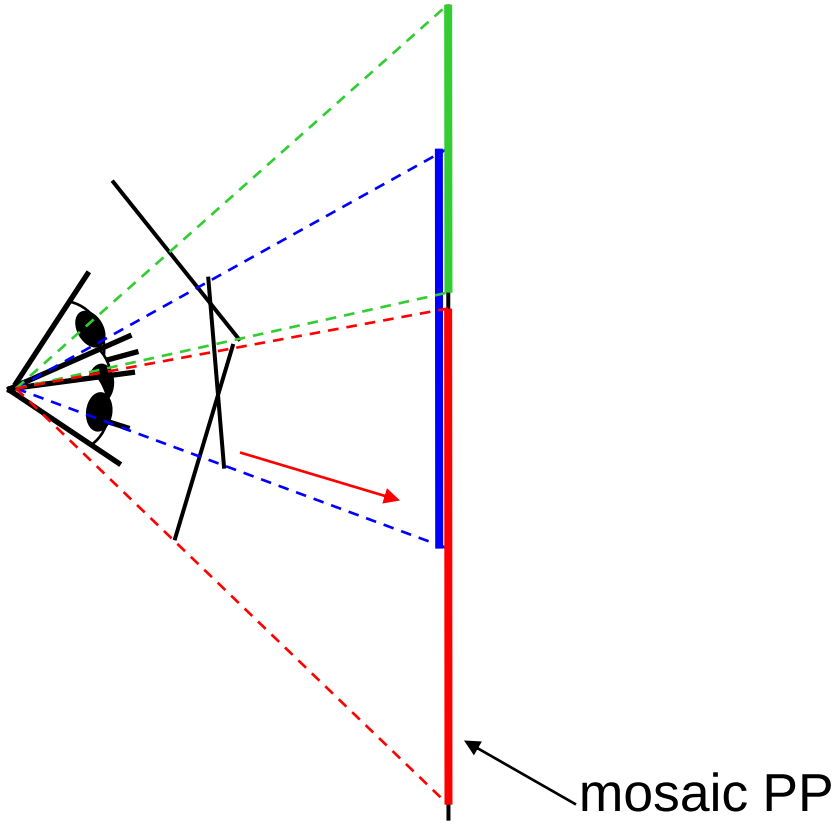


Image reprojection

- The mosaic has a natural interpretation in 3D
  - The images are reprojected onto a common plane
  - The mosaic is formed on this plane



Example

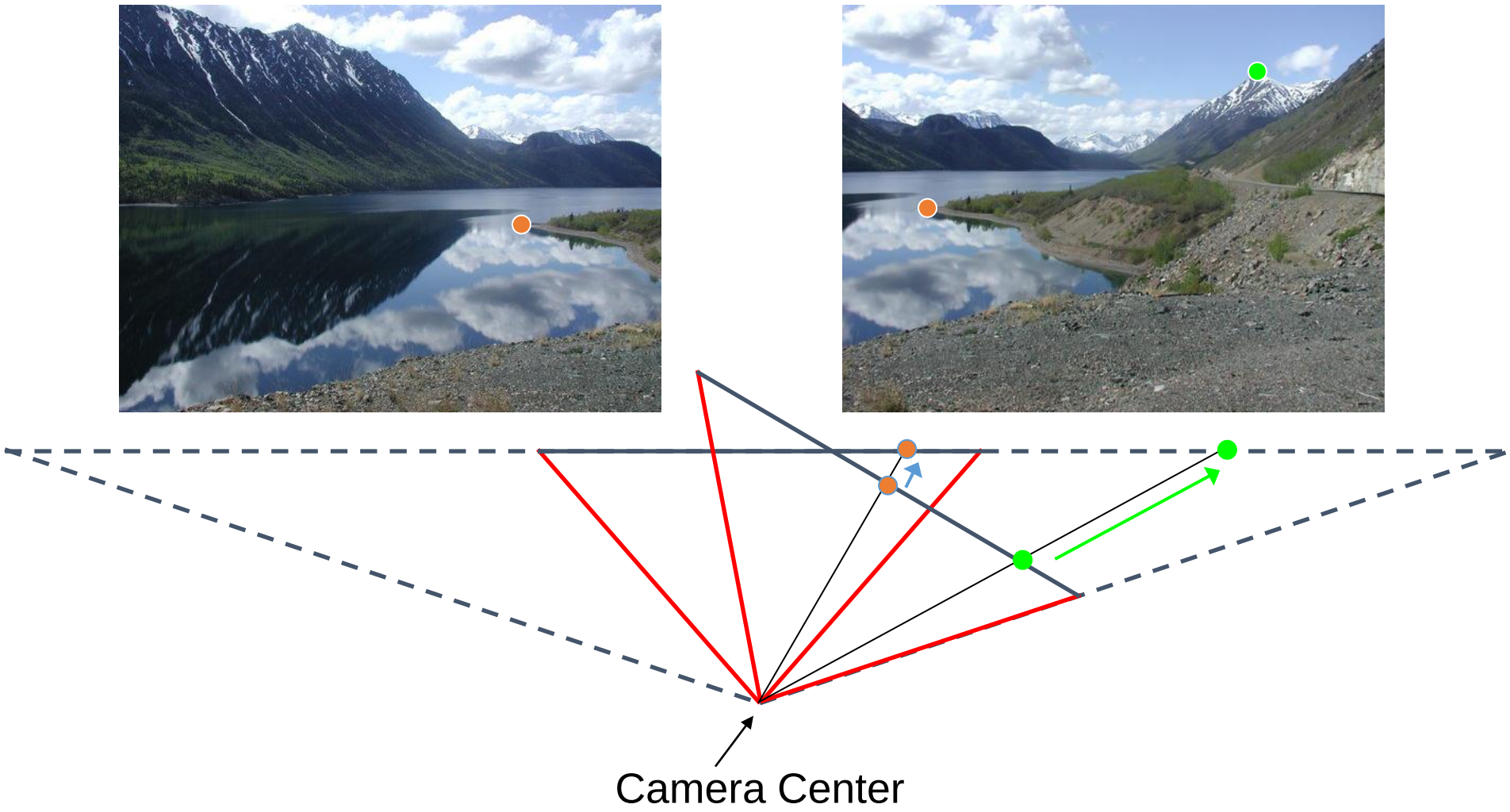
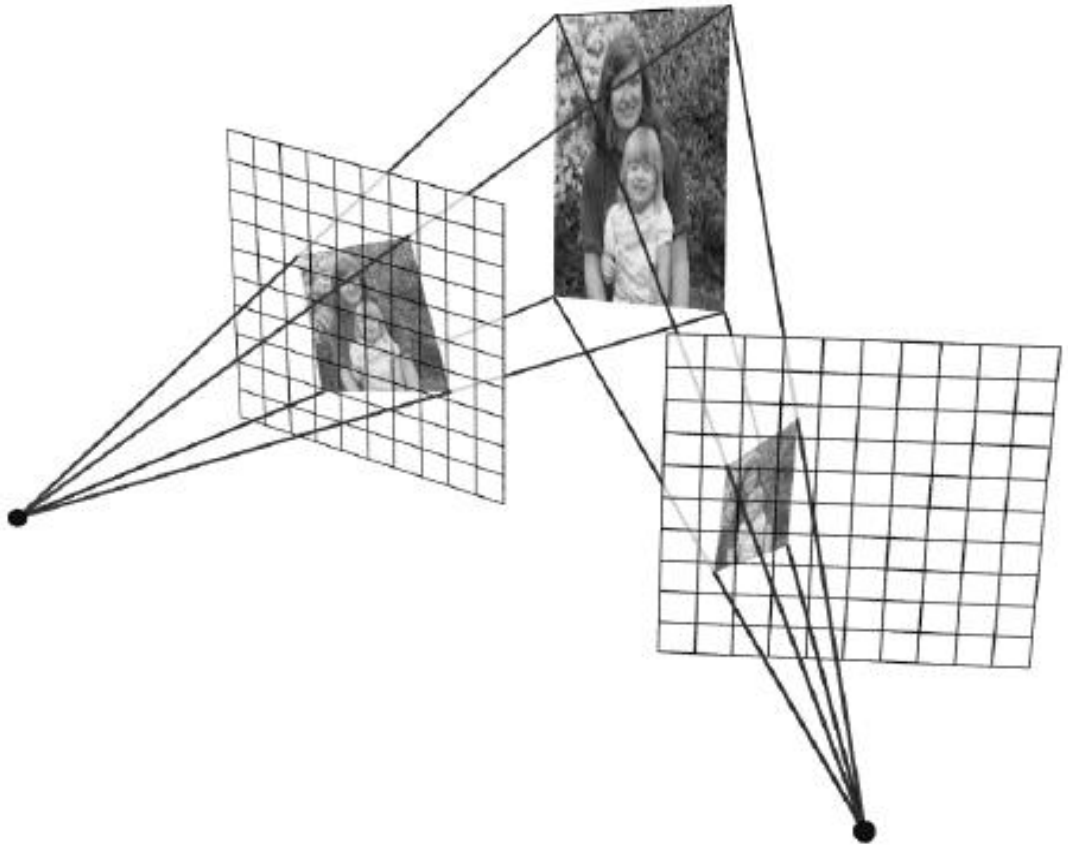


Image reprojection

- Observation
  - Rather than thinking of this as a 3D reprojection, think of it as a 2D image warp from on e image to another



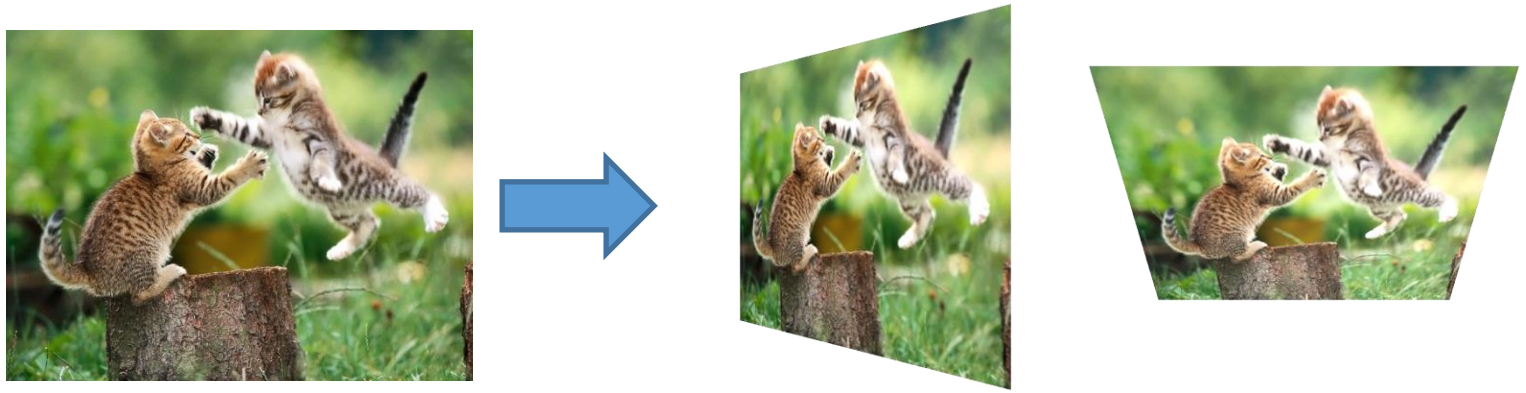
Motion models

- What happens when we take two images with a camera and try to align them?
  - translation?
  - rotation?
  - scale?
  - affine?
  - Perspective?

Recall: Projective transformations

- (aka homographies)

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \qquad x' = u/w \qquad y' = v/w$$



Parametric (global) warping

- Examples of parametric warps:

translation

rotation

aspect

affine

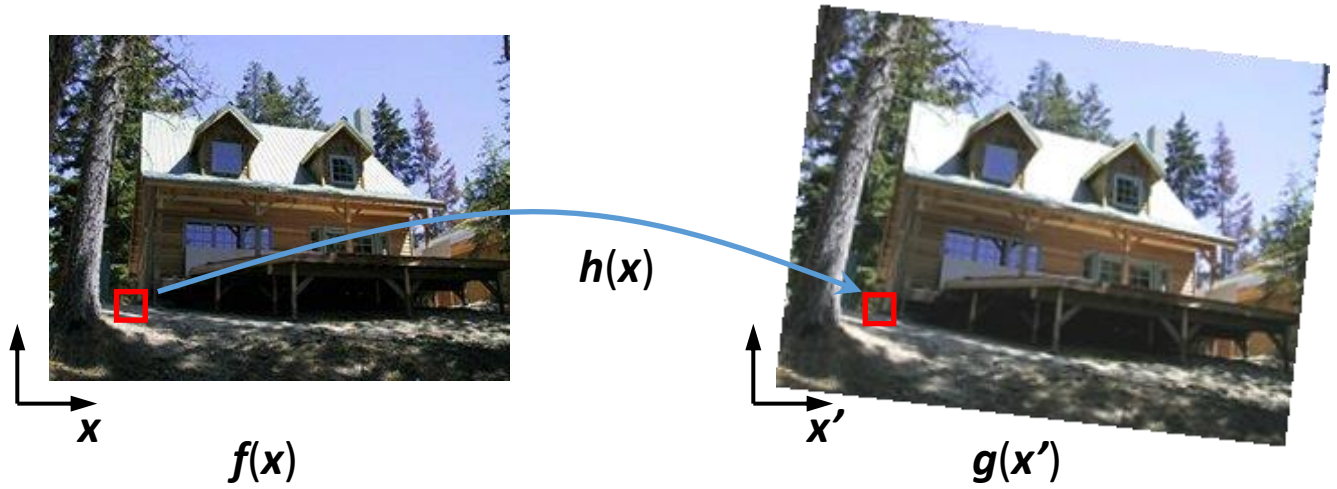
perspective

2D coordinate transformations

- translation: $x' = x + t$ $x = (x,y)$
- rotation: $x' = R\,x + t$
- similarity: $x' = s\,R\,x + t$
- affine: $x' = A\,x + t$
- perspective: $x' \cong H\,x$ $x = (x,y,1)$
  (x is a homogeneous coordinate)

Image Warping

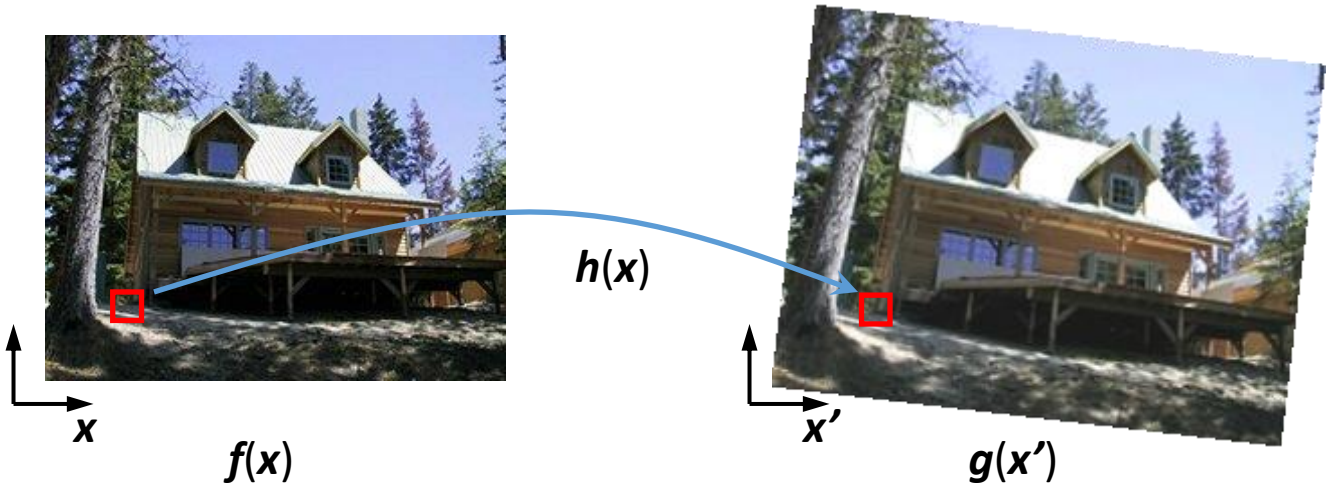
- Given a coordinate transform x’ = h(x) and a source image f(x), how do we co mpute a transformed image g(x’) = f(h(x))?



Forward Warping

- Send each pixel f(x) to its corresponding location x’ = h(x) in g(x’)

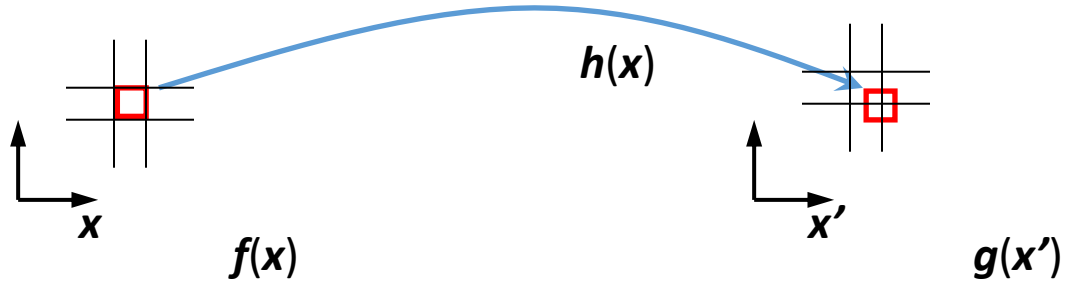
  - What if pixel lands “between” two pixels?



Forward Warping

- Send each pixel f(x) to its corresponding location x’ = h(x) in g(x’)

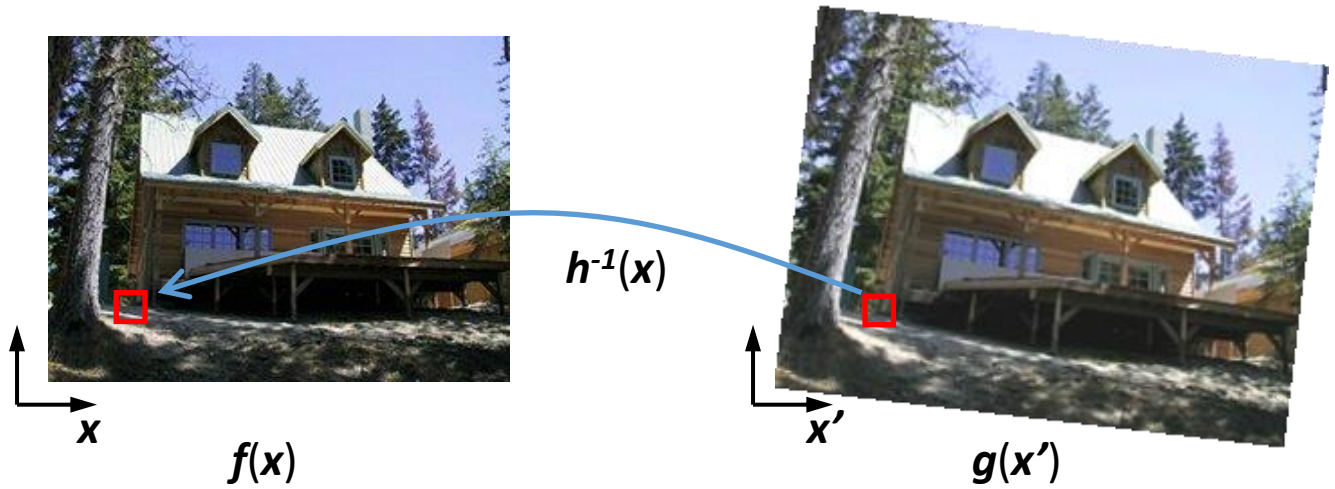
  - What if pixel lands “between” two pixels?
  - Answer: add “contribution” to several pixels, n ormalize later (*splatting*)



Inverse Warping

- Get each pixel g(x’) from its corresponding location x’ = h(x) in f(x)

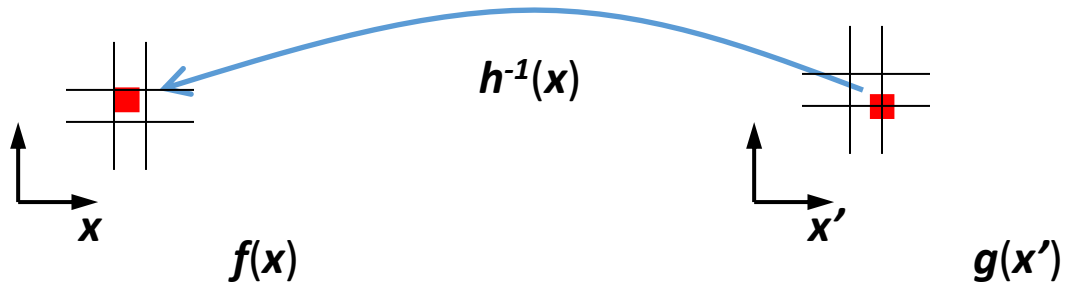
  - What if pixel comes from “between” two pixels?



Inverse Warping

- Get each pixel g(x’) from its corresponding location x’ = h(x) in f(x)

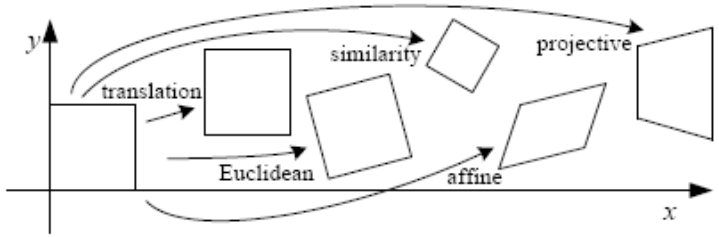
  - What if pixel comes from “between” two pixels?
  - Answer: *resample* color value from *interpolated* source image

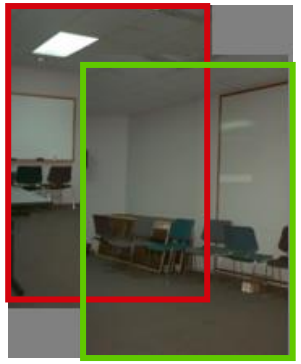
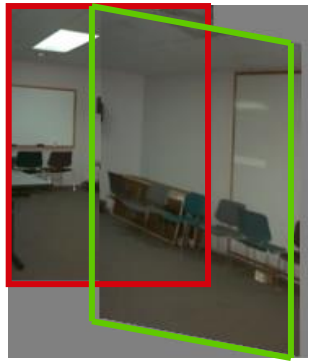
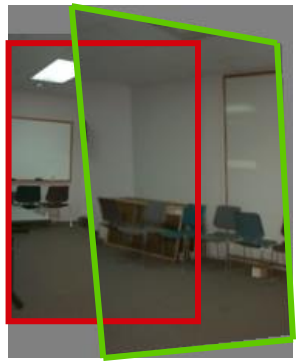
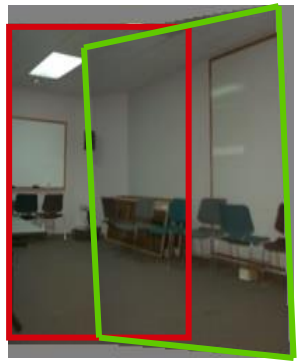


Interpolation

- Possible interpolation filters:
  - nearest neighbor
  - bilinear
  - bicubic (interpolating)

Motion models



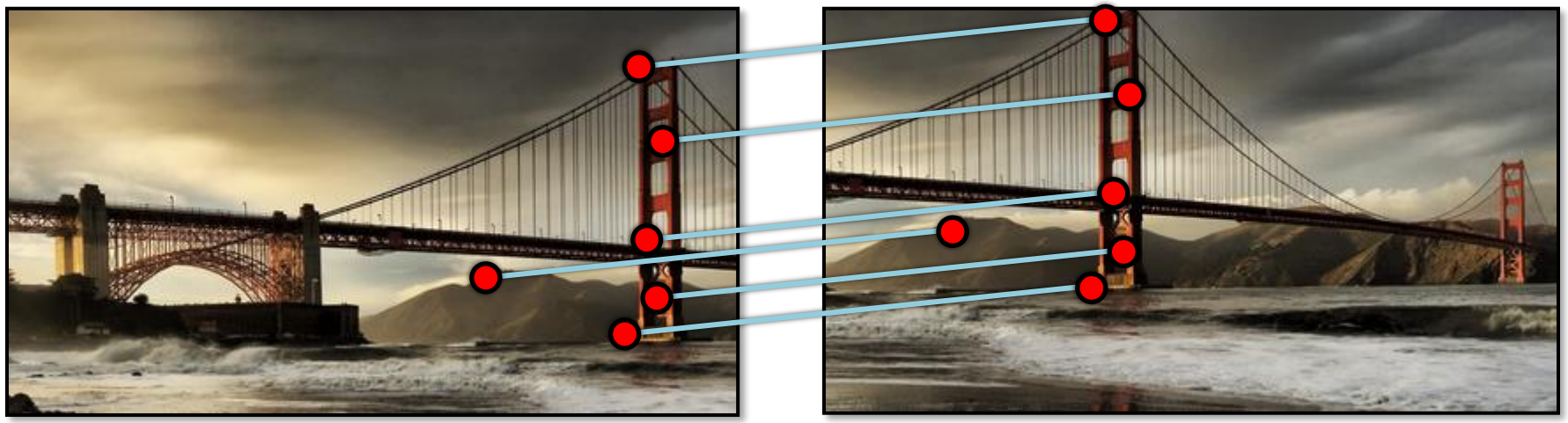
Translation	Affine	Perspective	3D rotation
			
2 unknowns	6 unknowns	8 unknowns	3 unknowns

Finding the transformation

- Translation = 2 degrees of freedom
- Similarity = 4 degrees of freedom
- Affine = 6 degrees of freedom
- Homography = 8 degrees of freedom

- How many corresponding points do we need to solve?

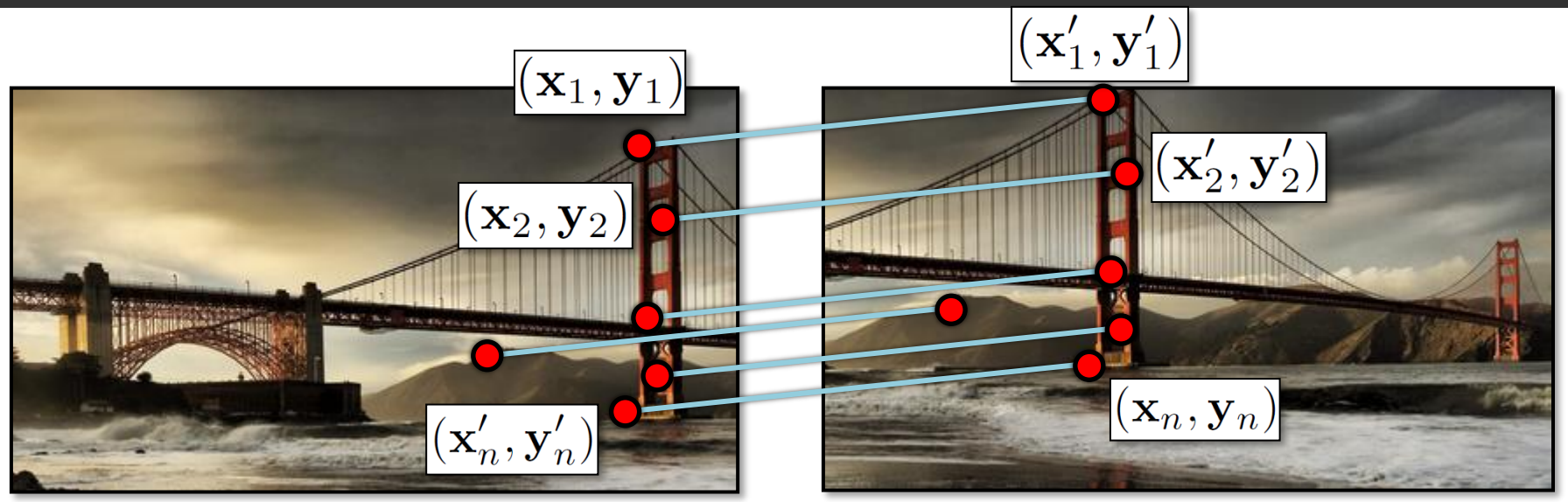
Simple case: translations



$(\mathbf{x}_t, \mathbf{y}_t)$

How do we solve for $(\mathbf{x}_t, \mathbf{y}_t)$?

Simple case: translations

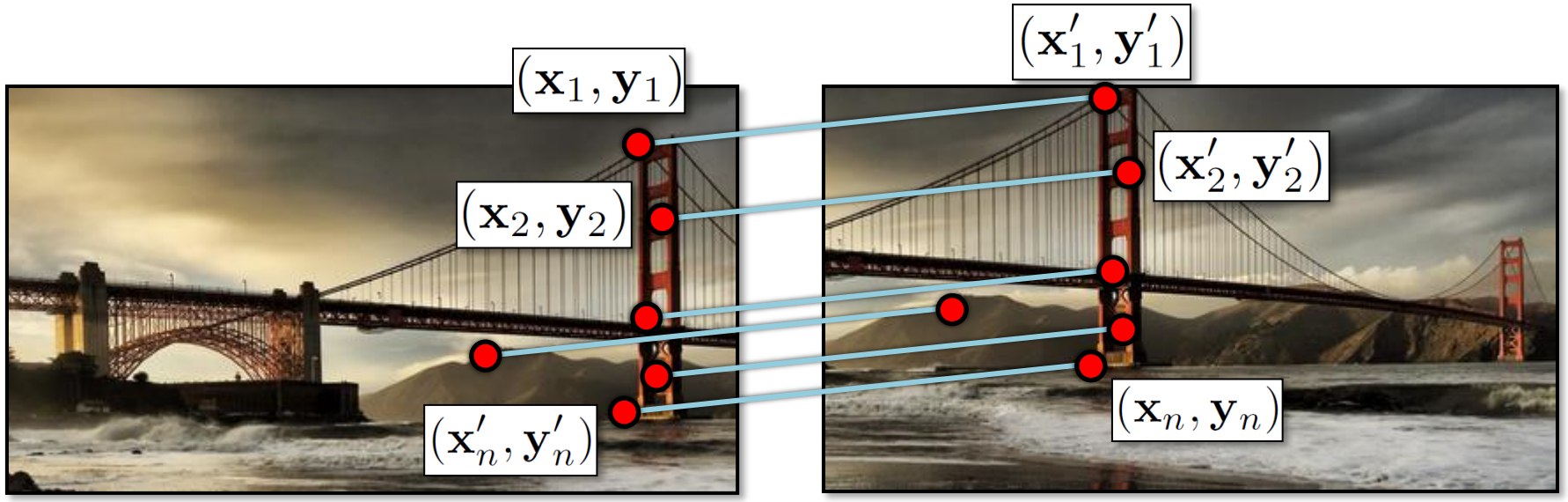


Displacement of match *i* = $(\mathbf{x}'_i - \mathbf{x}_i, \mathbf{y}'_i - \mathbf{y}_i)$

$$(\mathbf{x}_t, \mathbf{y}_t) = \left(\frac{1}{n} \sum_{i=1}^{n} \mathbf{x}'_i - \mathbf{x}_i, \frac{1}{n} \sum_{i=1}^{n} \mathbf{y}'_i - \mathbf{y}_i \right)$$

Simple case: translations

- System of linear equations
  - What are the knowns? Unknowns?
  - How many unknowns? How many equations (per match)?

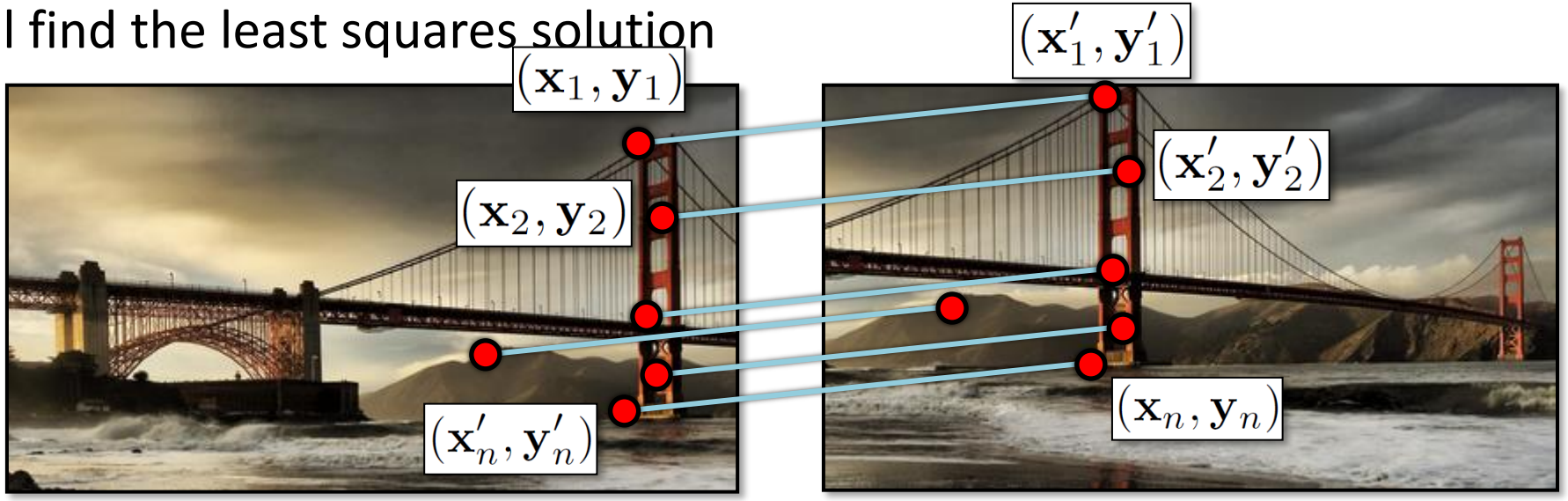


$$\mathbf{x}_i + \mathbf{x_t} = \mathbf{x}'_i$$

$$\mathbf{y}_i + \mathbf{y_t} = \mathbf{y}'_i$$

Simple case: translations

- Problem: more equations than unknowns
  - "Overdetermined" system of equations
  - We will find the least squares solution



$$\mathbf{x}_i + \mathbf{x_t} = \mathbf{x}'_i$$

$$\mathbf{y}_i + \mathbf{y_t} = \mathbf{y}'_i$$

Least squares formulation

- For each point $(\mathbf{x}_i, \mathbf{y}_i)$

$$\begin{aligned} \mathbf{x}_i + \mathbf{x_t} &= \mathbf{x}'_i \\ \mathbf{y}_i + \mathbf{y_t} &= \mathbf{y}'_i \end{aligned}$$

- we define the residuals as

$$\begin{aligned} r_{\mathbf{x}_i}(\mathbf{x}_t) &= (\mathbf{x}_i + \mathbf{x}_t) - \mathbf{x}'_i \\ r_{\mathbf{y}_i}(\mathbf{y}_t) &= (\mathbf{y}_i + \mathbf{y}_t) - \mathbf{y}'_i \end{aligned}$$

Least squares formulation

- Goal: minimize sum of squared residuals

$$C(\mathbf{x}_t, \mathbf{y}_t) = \sum_{i=1}^{n} \left(r_{\mathbf{x}_i}(\mathbf{x}_t)^2 + r_{\mathbf{y}_i}(\mathbf{y}_t)^2 \right)$$

- “Least squares” solution
- For translations, is equal to mean displacement

Least squares

- Find t that minimizes

$$\mathbf{A}\mathbf{t} = \mathbf{b}$$

- To solve, form the normal equations

$$||\mathbf{A}\mathbf{t} - \mathbf{b}||^2$$

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{t} = \mathbf{A}^{\mathrm{T}}\mathbf{b}$$

$$\mathbf{t} = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b}$$

Solving for translations

- Using least squares

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ \vdots & \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} x'_1 - x_1 \\ y'_1 - y_1 \\ x'_2 - x_2 \\ y'_2 - y_2 \\ \vdots \\ x'_n - x_n \\ y'_n - y_n \end{bmatrix}$$

$$\underset{2n \text{ x } 2}{\mathbf{A}} \quad \underset{2 \text{ x } 1}{\mathbf{t}} = \underset{2n \text{ x } 1}{\mathbf{b}}$$

Affine transformations

- How many unknowns?
- How many equations per match?
- How many matches do we need?

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

Affine transformations

- Residuals:

$$
\begin{aligned}
r_{x_i}(a,b,c,d,e,f) &= (ax_i + by_i + c) - x'_i \\
r_{y_i}(a,b,c,d,e,f) &= (dx_i + ey_i + f) - y'_i
\end{aligned}
$$

- Cost function:

$$
C(a,b,c,d,e,f) = \sum_{i=1}^{n} \left(r_{x_i}(a,b,c,d,e,f)^2 + r_{y_i}(a,b,c,d,e,f)^2 \right)
$$

Affine transformations

- Matrix form

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ & & \vdots & & & \\ x_n & y_n & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ x'_n \\ y'_n \end{bmatrix}$$

$$\underset{2n \times 6}{\mathbf{A}} \qquad \underset{6 \times 1}{\mathbf{t}} = \underset{2n \times 1}{\mathbf{b}}$$

Computing homography

- Assume we have four matched points: How do we compute homography H?

- Normalized DLT
- Normalize coordinates for each image
  - Translate for zero mean
  - Scale so that average distance to origin is ~sqrt(2)

$$\tilde{\mathbf{x}} = \mathbf{T}\mathbf{x} \qquad \tilde{\mathbf{x}}' = \mathbf{T}'\mathbf{x}'$$

  - This makes problem better behaved numerically

- Compute $\tilde{\mathbf{H}}$ using DLT in normalized coordinates
- Unnormalize:

$$\mathbf{H} = \mathbf{T}'^{-1}\tilde{\mathbf{H}}\mathbf{T}$$

$$\mathbf{x}_i' = \mathbf{H}\mathbf{x}_i$$

Computing homography

- Assume we have matched points with outliers: How do we compute homography H?

- Automatic Homography Estimation with RANSAC
- Choose number of samples N
- Choose 4 random potential matches
- Compute H using normalized DLT $\mathbf{x}_i' = \mathbf{H}\mathbf{x}_i$
- Project points from x to x’ for each potentially matching pair:
- Count points with projected distance < t
  - E.g., t = 3 pixels
- Repeat steps 2-5 N times
  - Choose H with most inliers

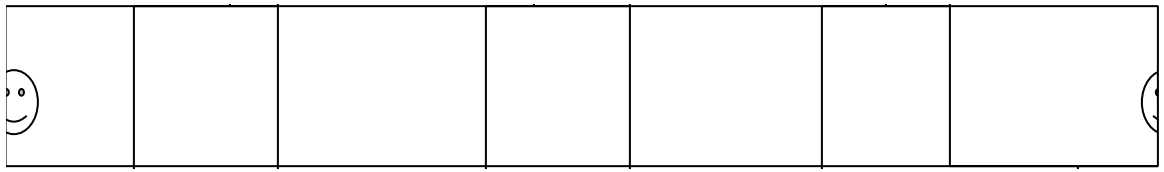
HZ Tutorial ‘99

Automatic Image Stitching

- Compute interest points on each image
- Find candidate matches
- Estimate homography H using matched points and RANSAC with normalized DLT
- Project each image onto the same surface and blend

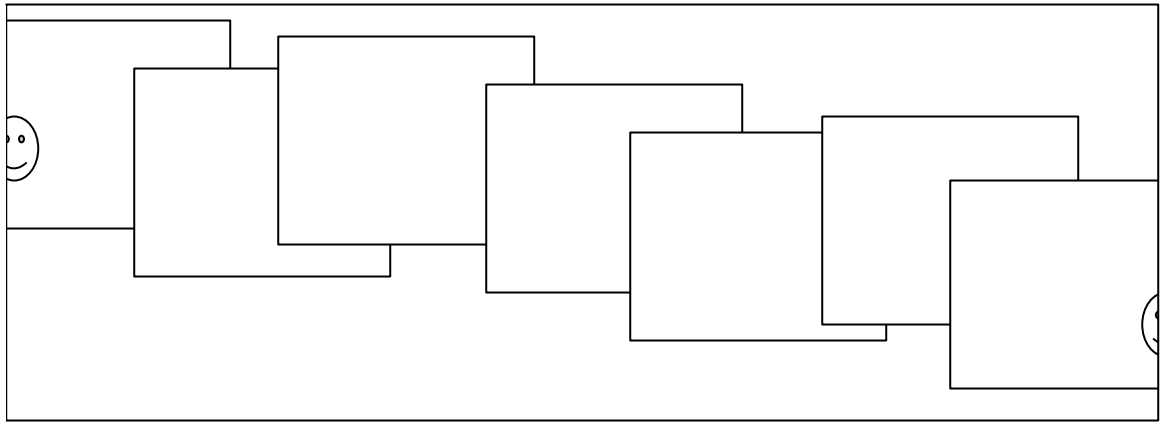
Assembling the panorama

- Stitch pairs together, blend, then crop



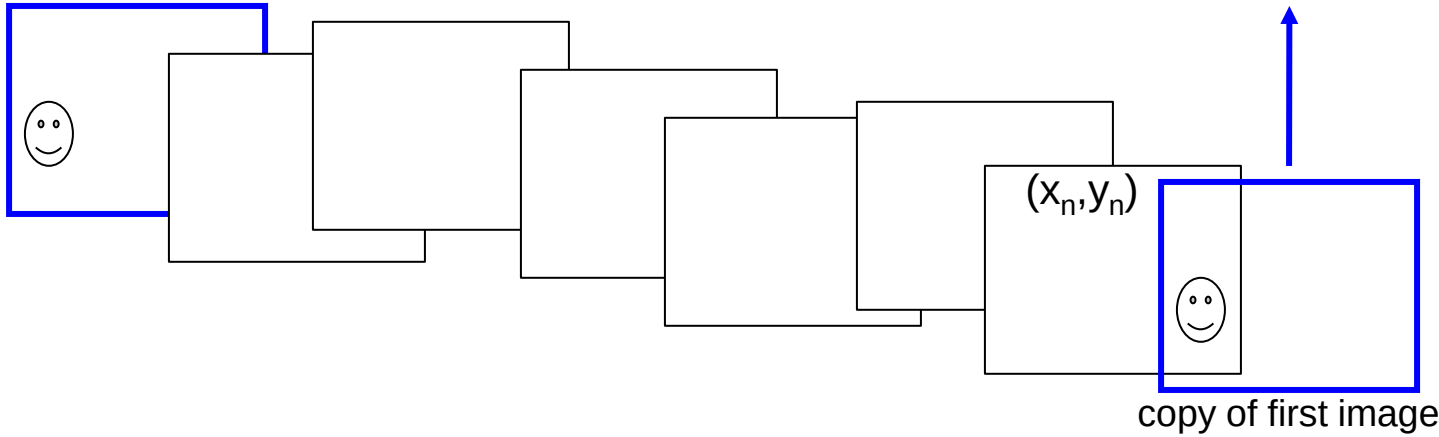
Problem: Drift

- Error accumulation
  - small errors accumulate over time



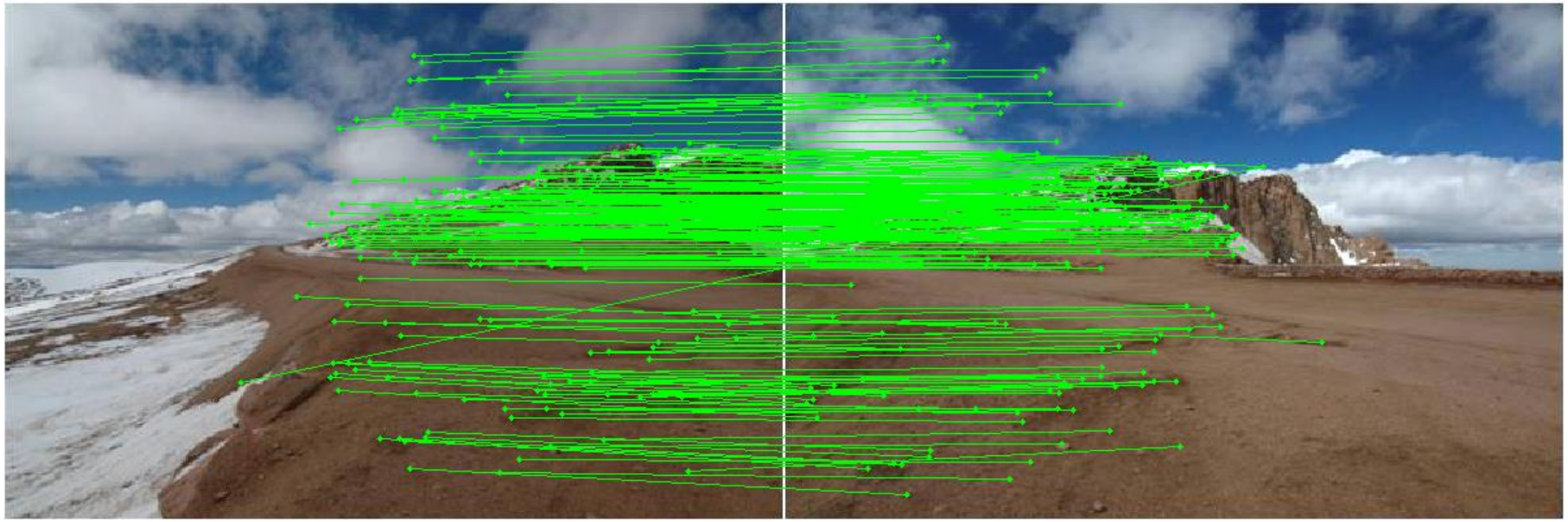
Problem: Drift

- Solution
  - add another copy of first image at the end
  - this gives a constraint: yn = y1
  - there are a bunch of ways to solve this problem
    - add displacement of (y1 – yn)/(n -1) to each image after the first
    - compute a global warp: y’ = y + ax
    - run a big optimization problem, incorporating this constraint
      - best solution, but more complicated
      - known as “bundle adjustment”

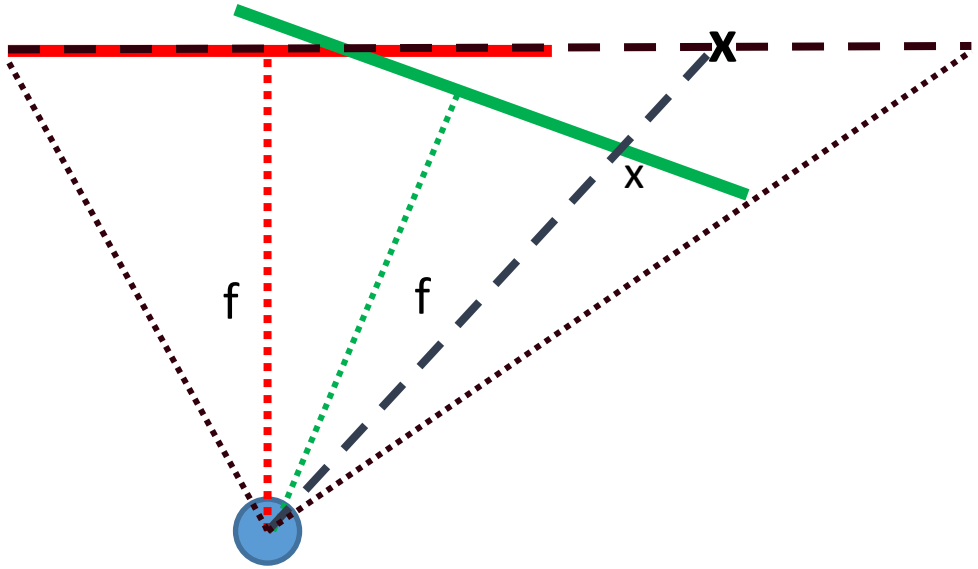


Choosing a Projection Surface

- Many to choose: planar, cylindrical, spherical, cubic, etc.

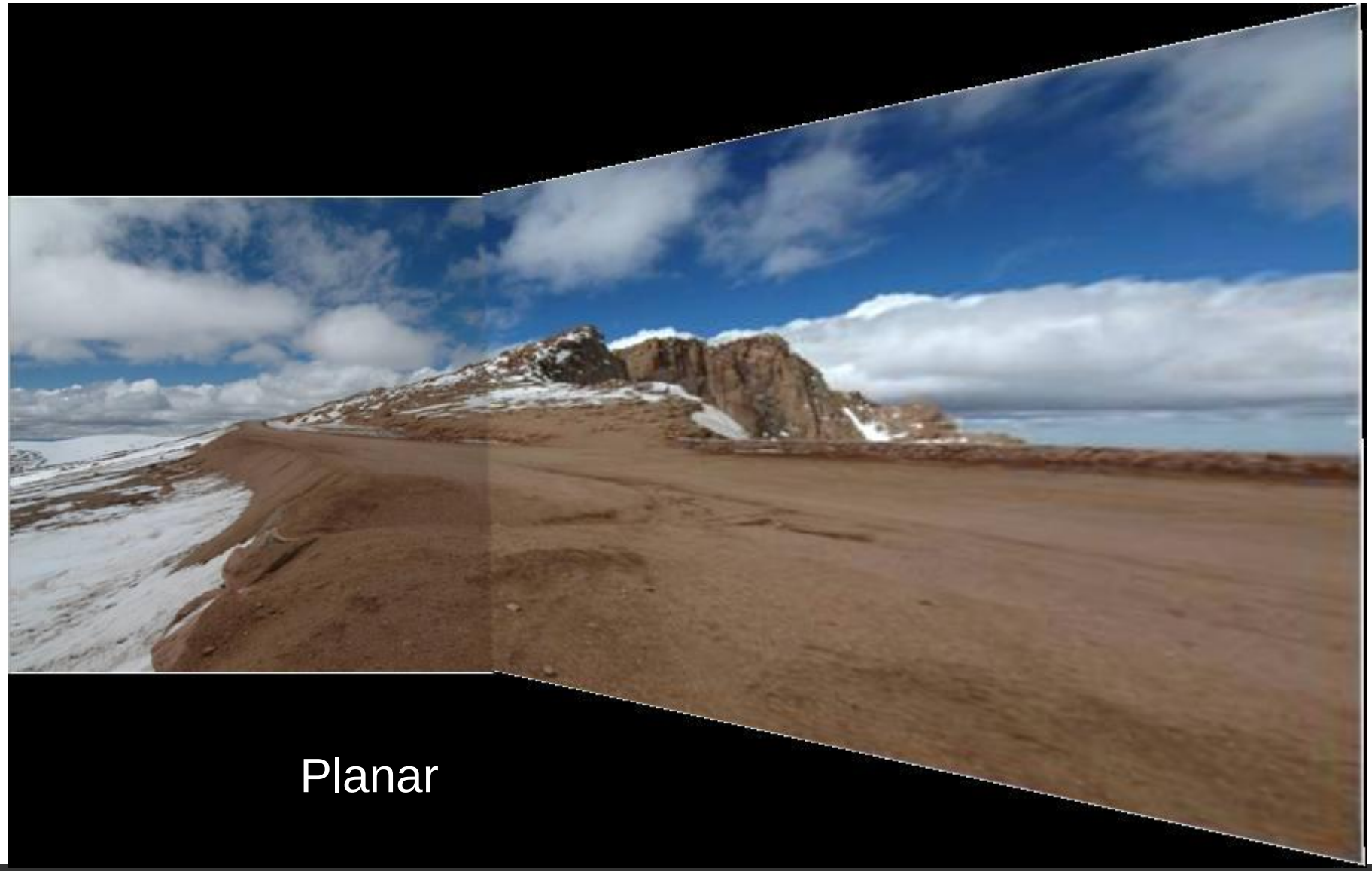


Planar Mapping



1) For red image: pixels are already on the planar surface
2) For green image: map to first image plane

Planar Projection

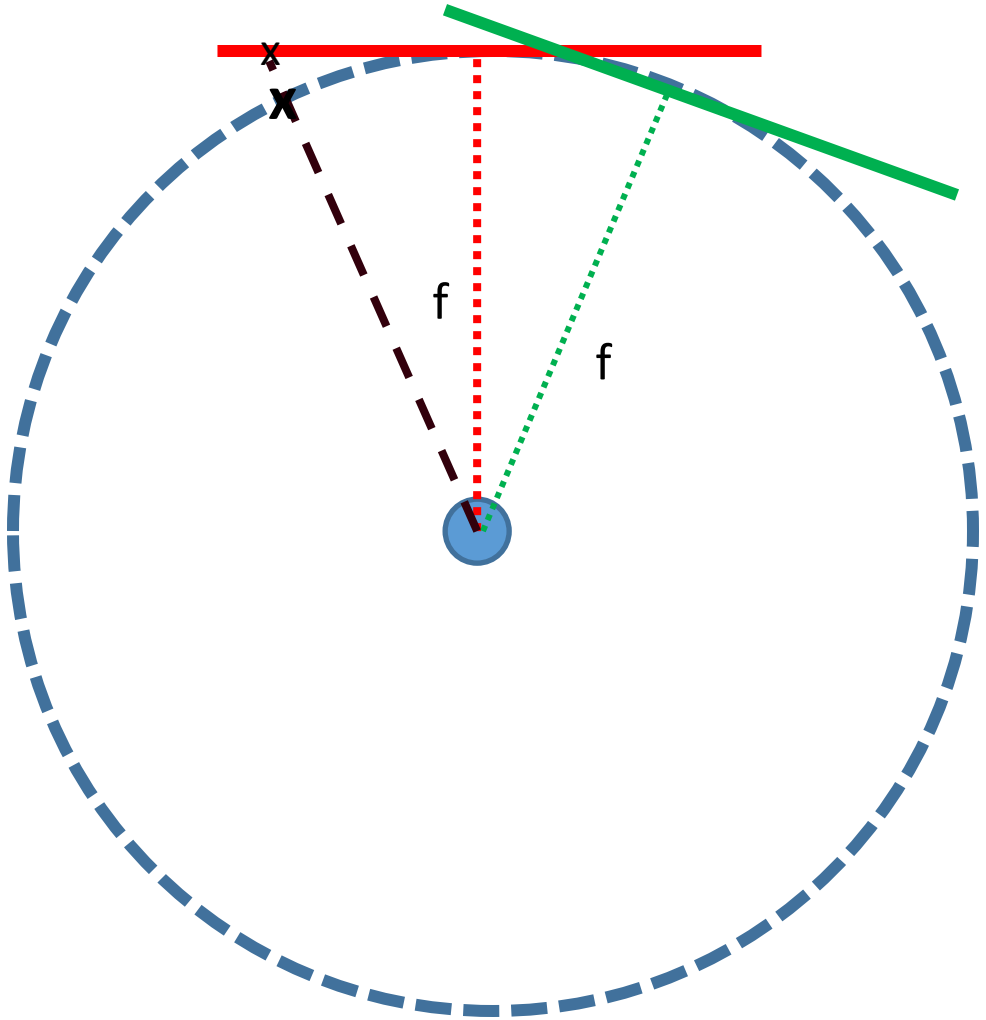


Photos by Russ Hewett

Planar Projection

Planar

Cylindrical Mapping



1) For red image: compute h, theta on cylindrical surface from (u, v)
2) For green image: map to first image plane, than map to cylindrical surface

Cylindrical Projection

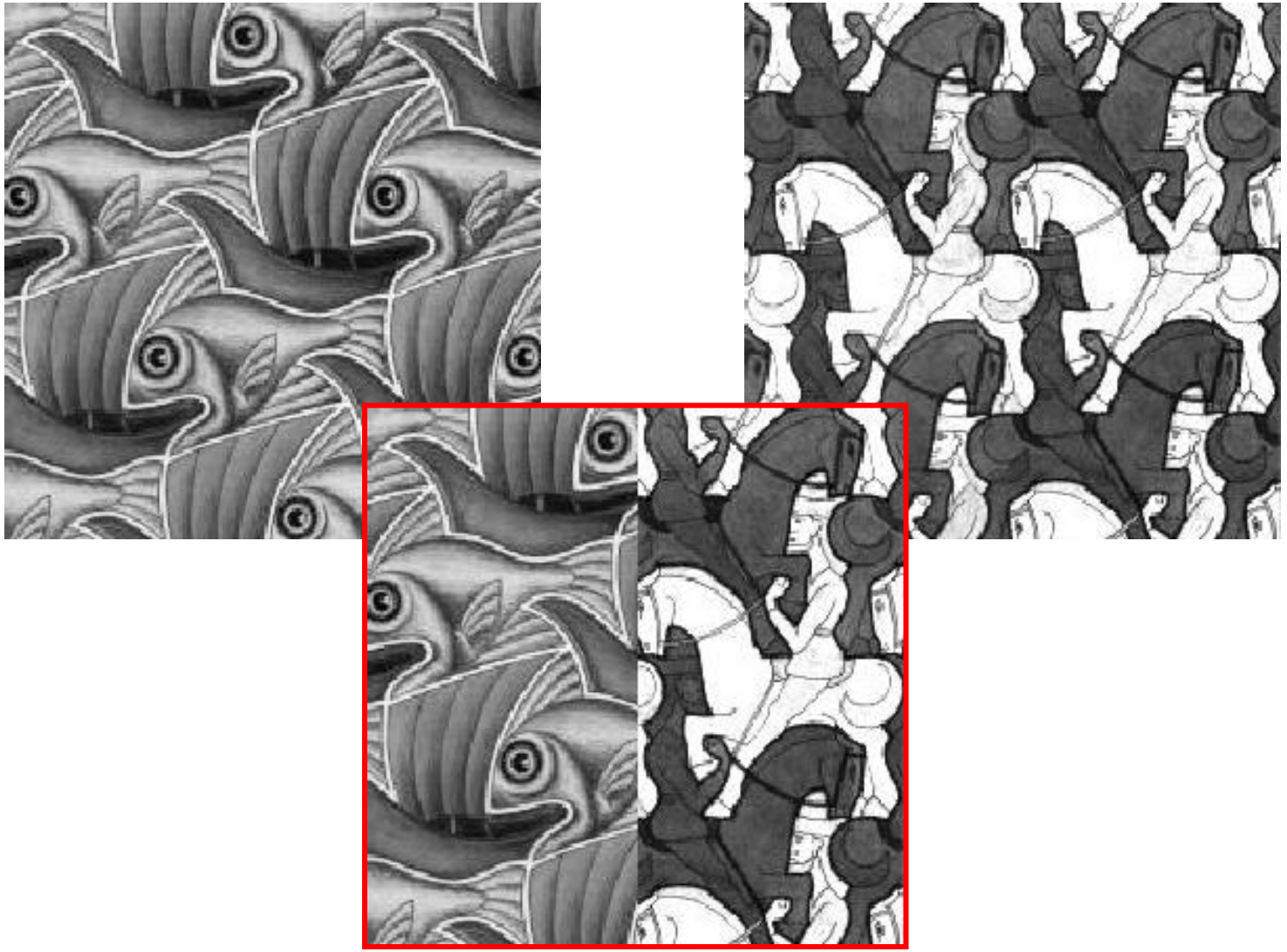
Cylindrical

Cylindrical Projection

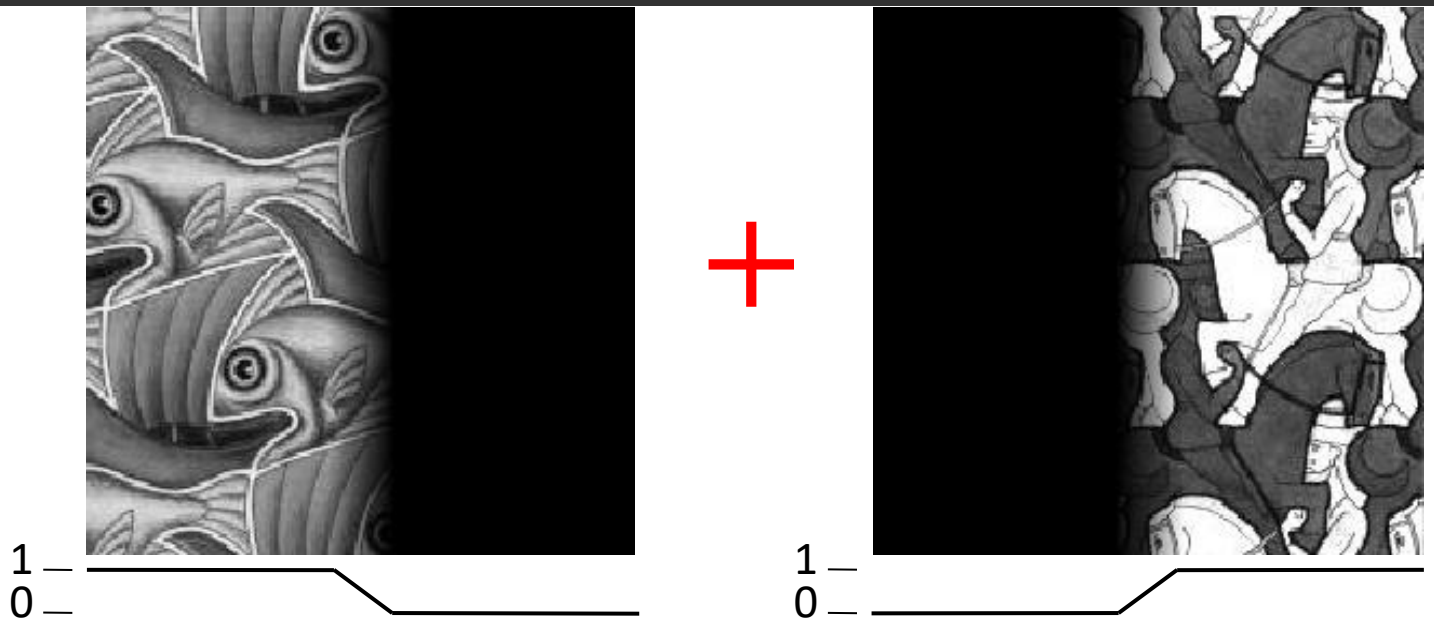
Cylindrical

Comparisons

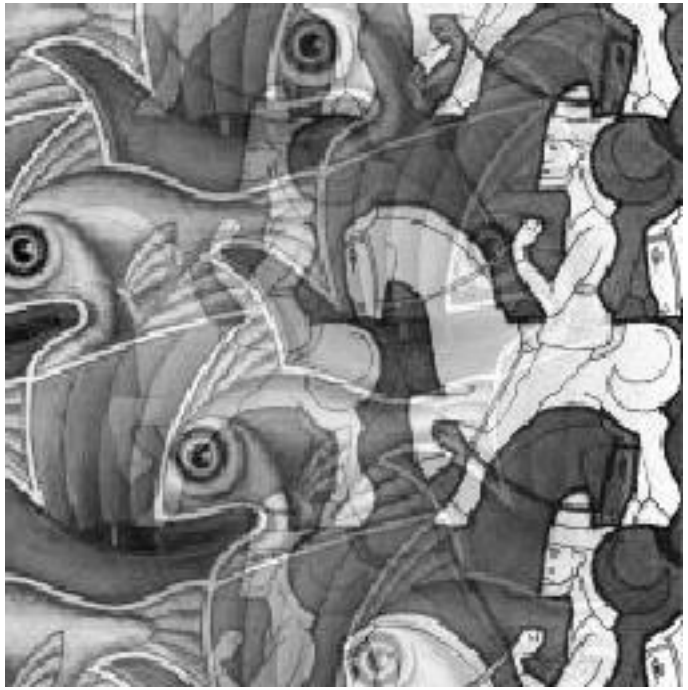
Image Blending

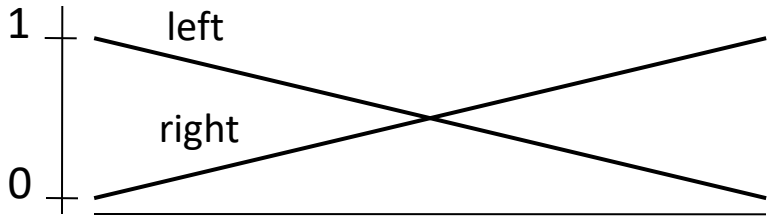


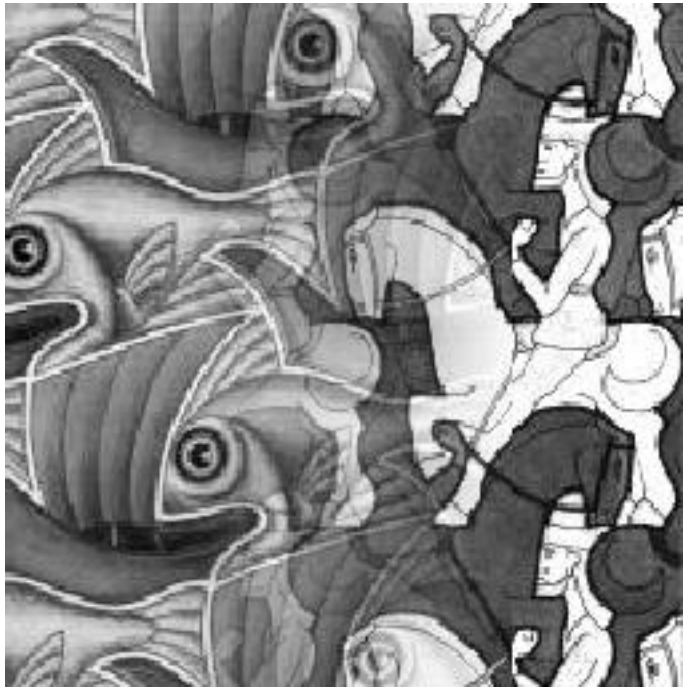
Feathering

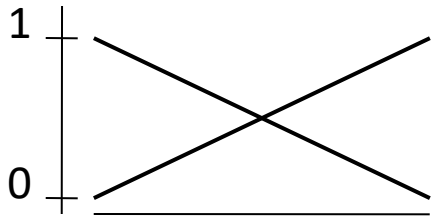


Effect of window (ramp-width) size

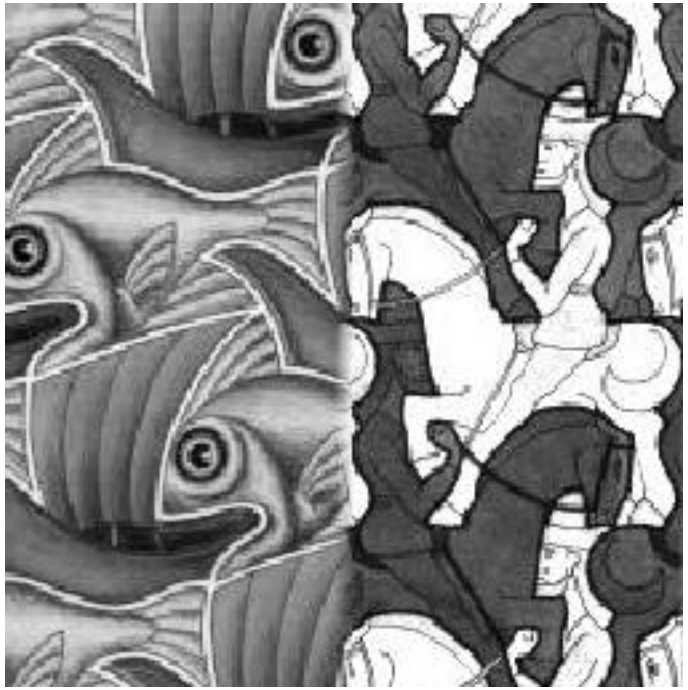


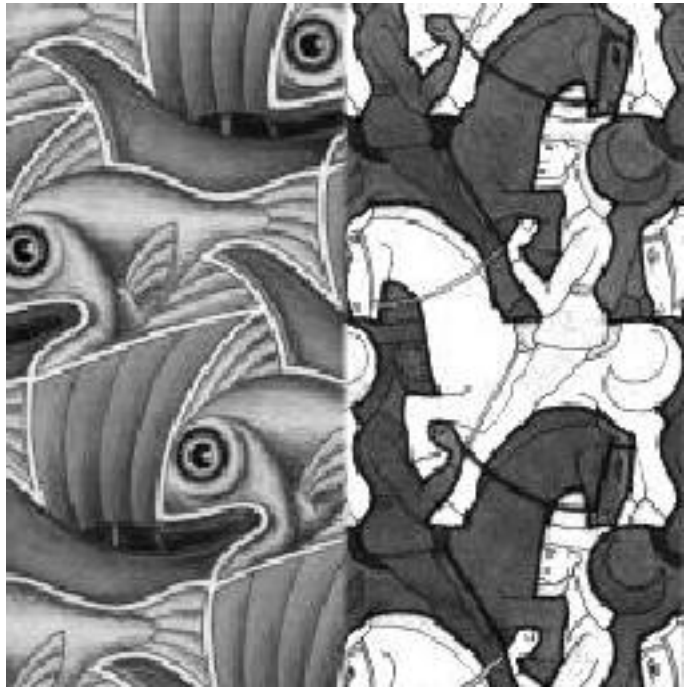




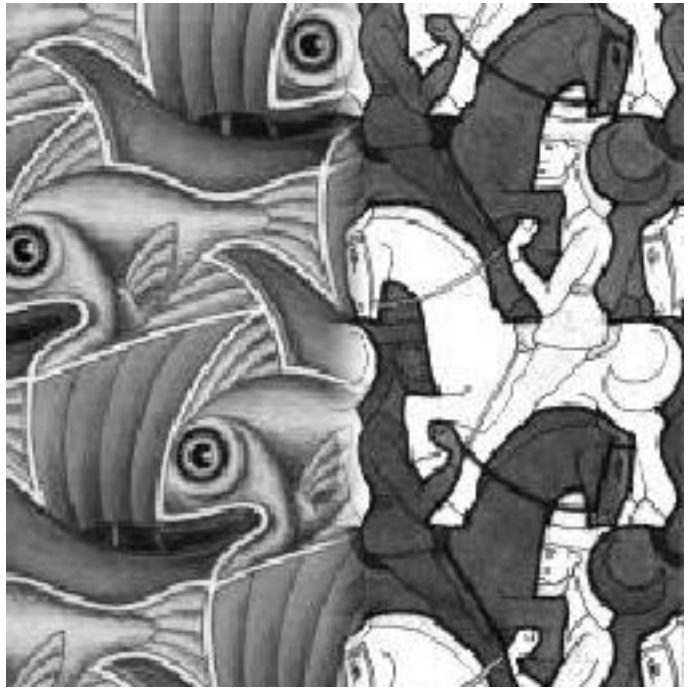


Effect of window size





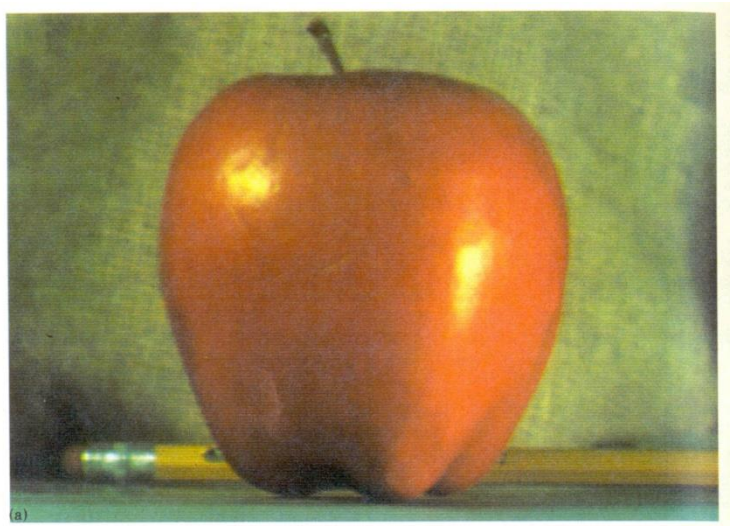
Good window size

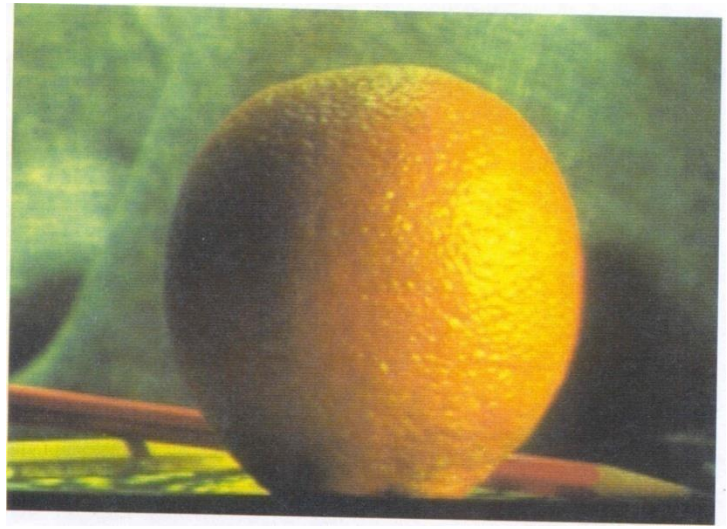


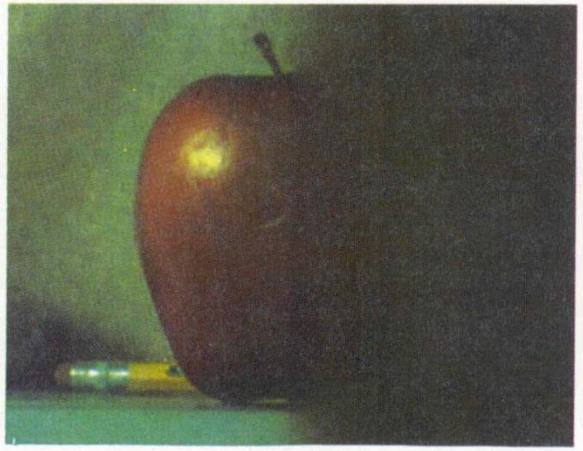
“Optimal” window: smooth but not ghosted

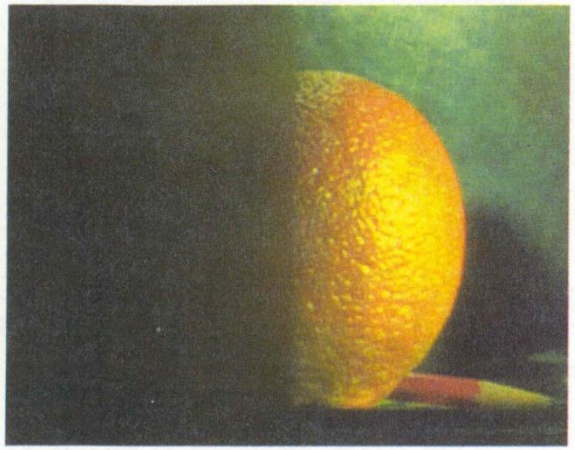
- Doesn’t always work...

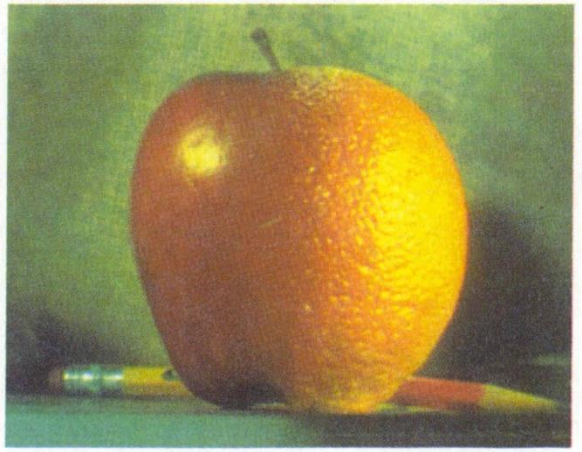
Pyramid blending











Create a Laplacian pyramid, blend each level

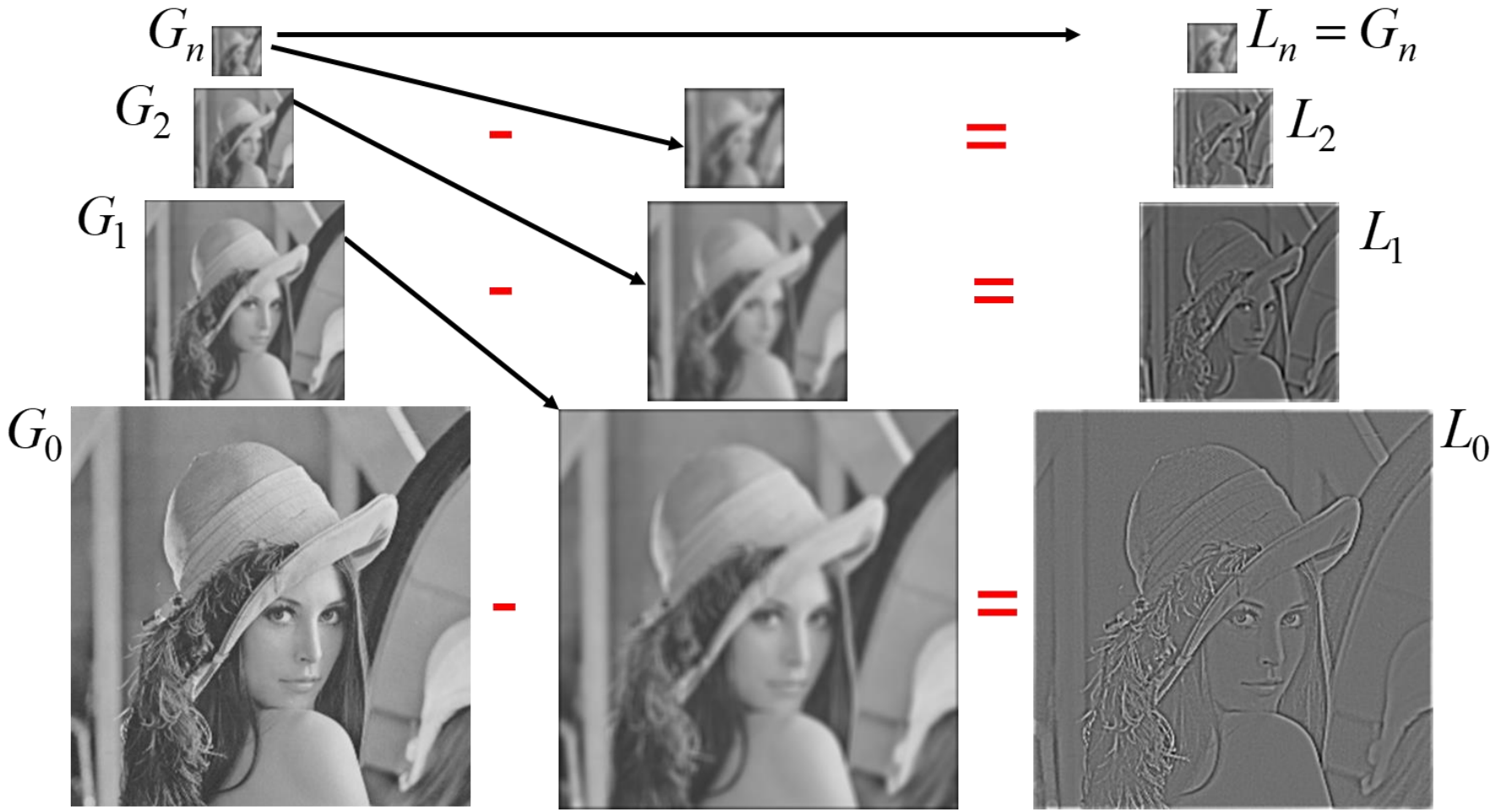
The Laplacian Pyramid

$$L_i = G_i - \text{expand}(G_{i+1})$$

$$G_i = L_i + \text{expand}(G_{i+1})$$

Gaussian Pyramid

Laplacian Pyramid

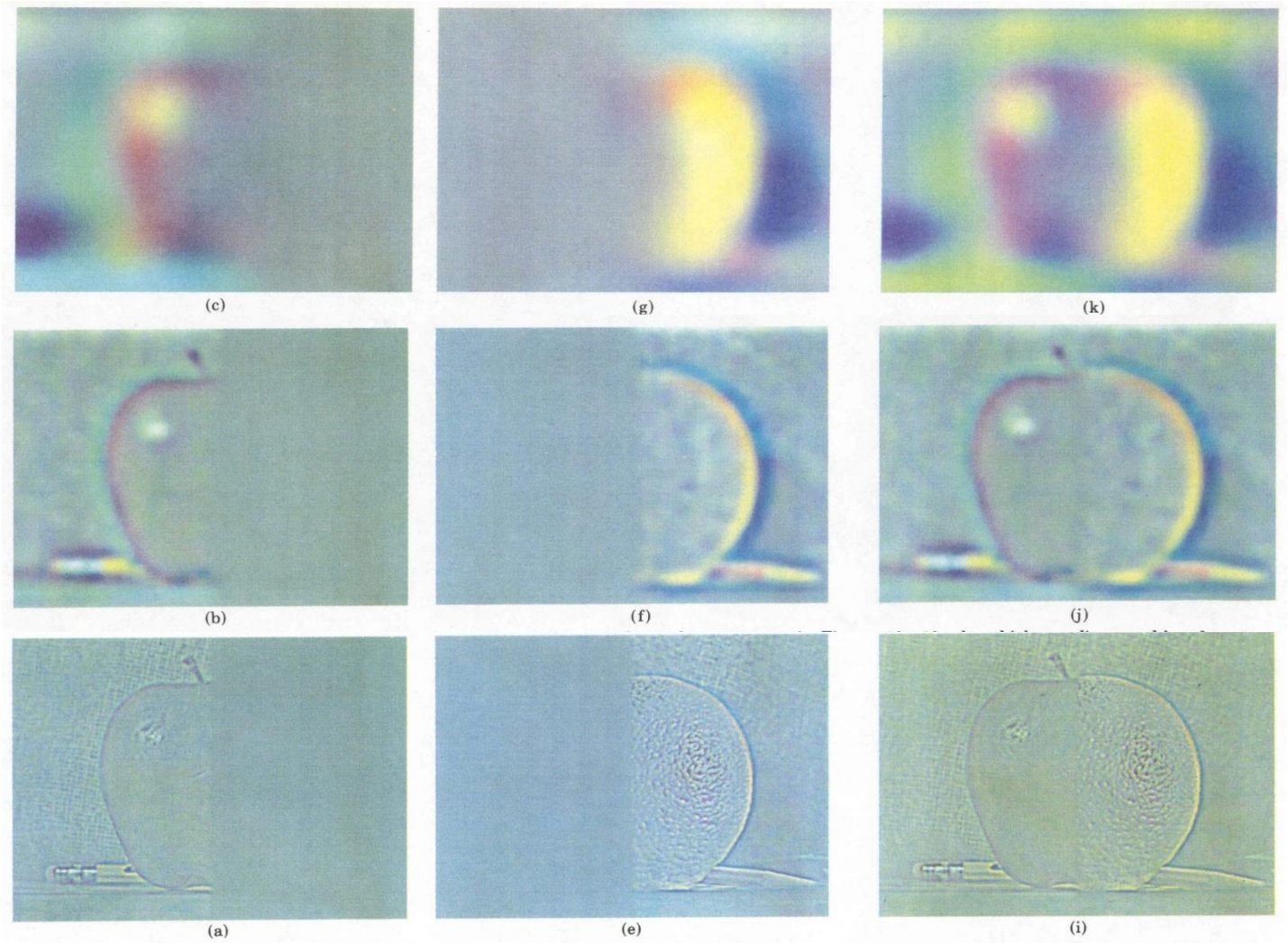


The Laplacian Pyramid

Laplacian level 4

Laplacian level 2

Laplacian level 0



left pyramid

right pyramid

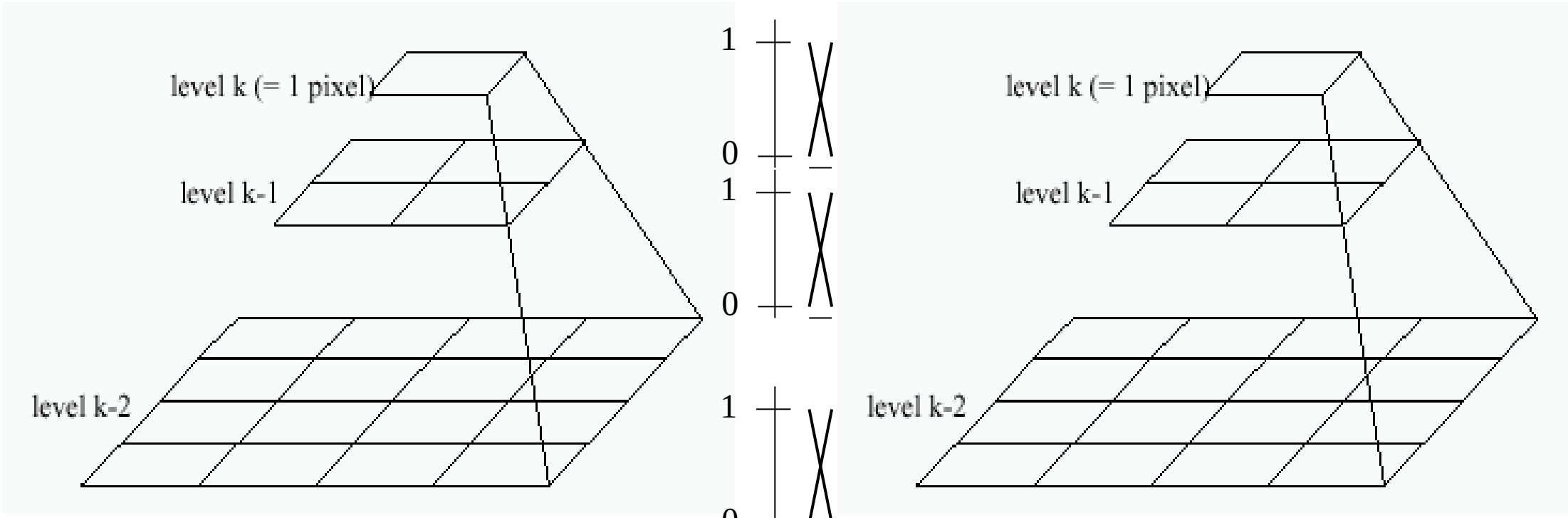
blended pyramid

Laplacian image blend

- Compute Laplacian pyramid
- Compute Gaussian pyramid on weight image (can put this in A channel)
- Blend Laplacians using Gaussian blurred weights
- Reconstruct the final image
- Q: How do we compute the original weights?

Multiband Blending with Laplacian Pyramid

- At low frequencies, blend slowly
- At high frequencies, blend quickly



Left pyramid blend Right pyramid

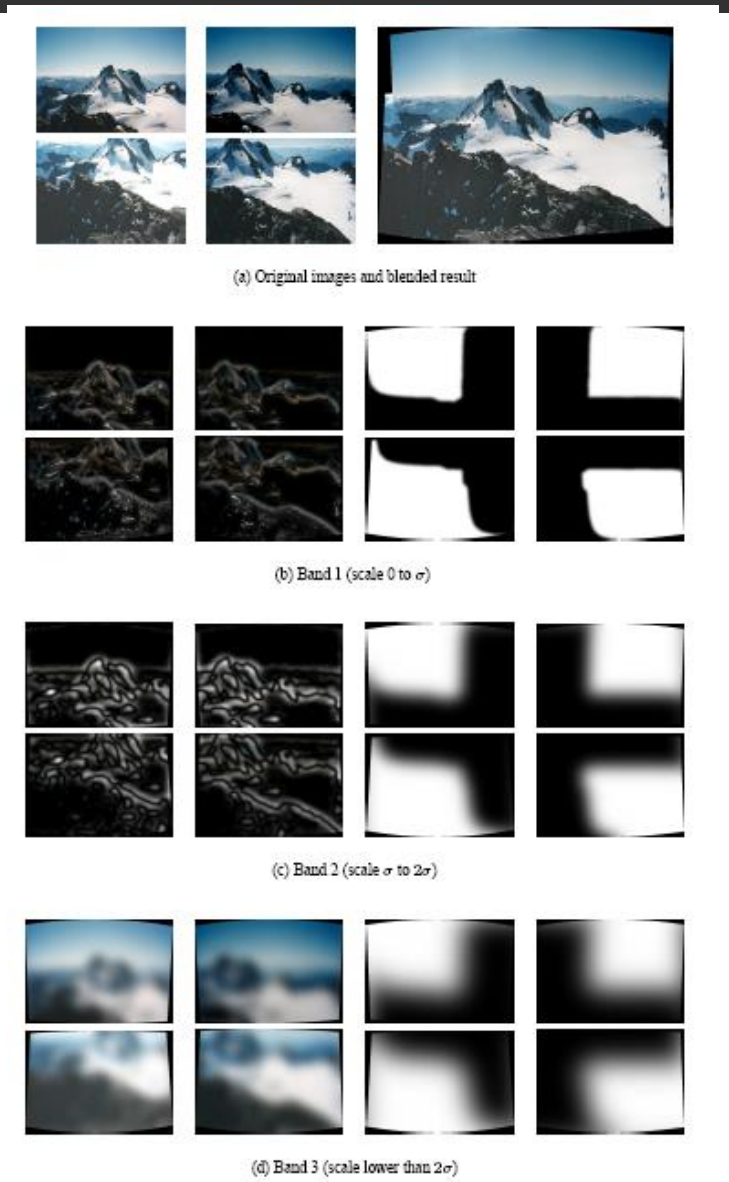
Multiband blending

Laplacian pyramids

1. Compute Laplacian pyramid of images and mask

2. Create blended image at eac h level of pyramid

3. Reconstruct complete image



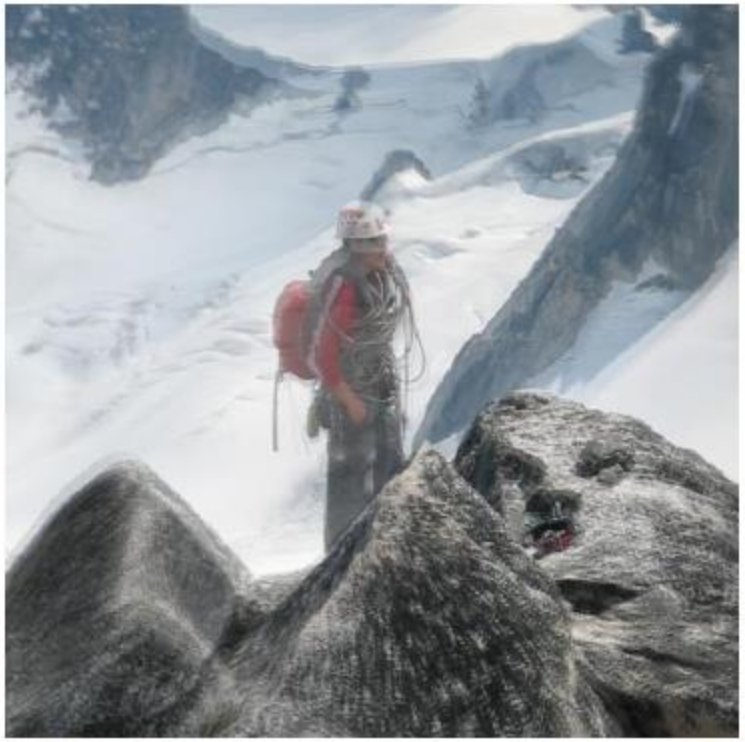
(a) Original images and blended result

(b) Band 1 (scale 0 to σ)

(c) Band 2 (scale σ to 2σ)

(d) Band 3 (scale lower than 2σ)

Blending comparison (IJCV 2007)

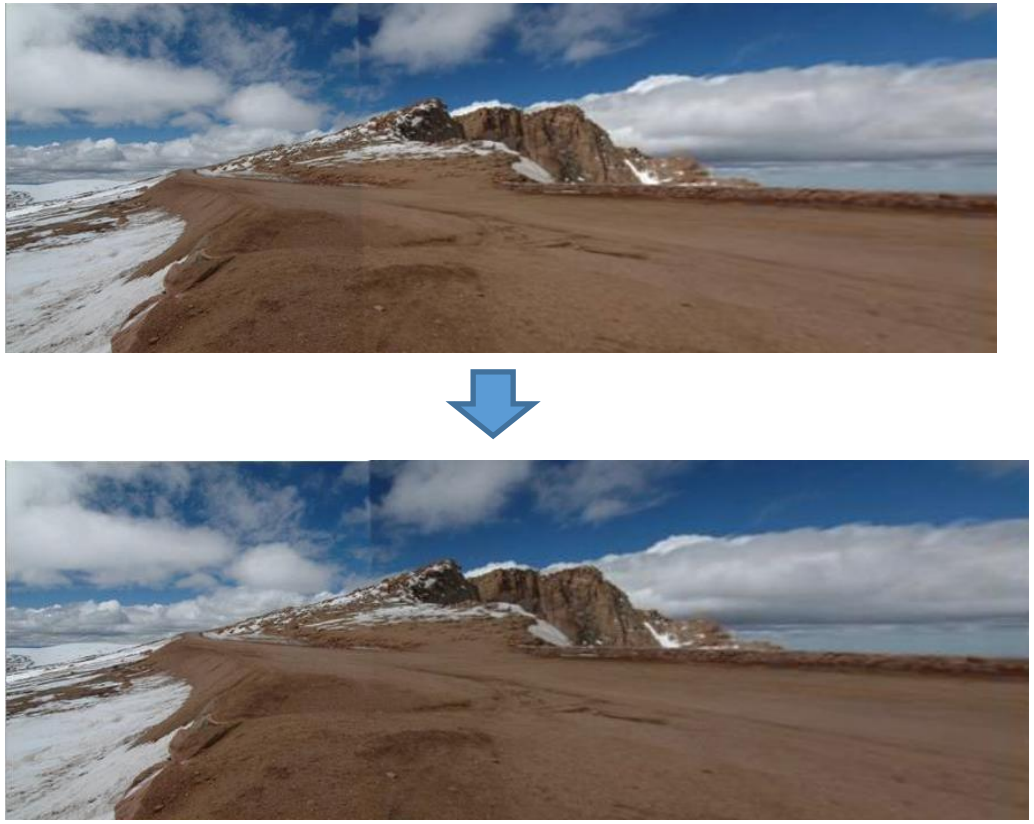


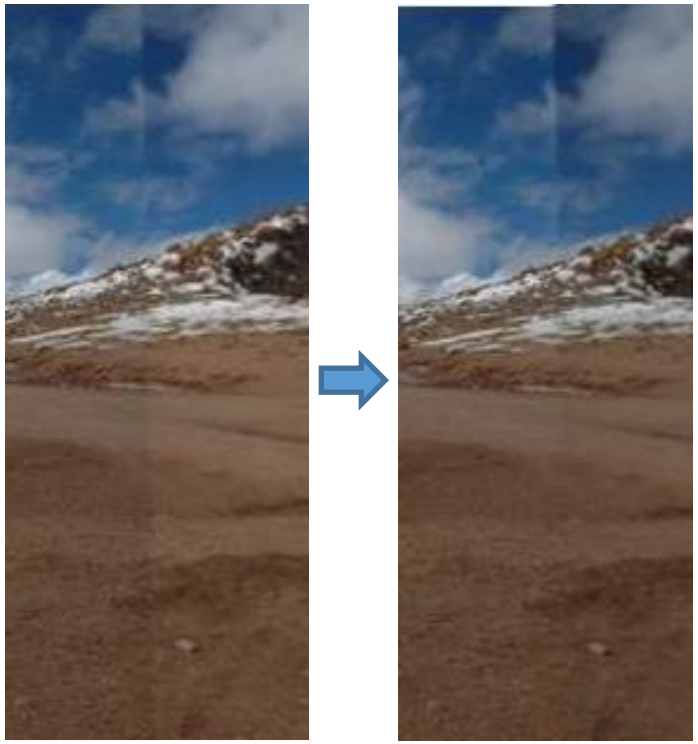
(a) Linear blending

(b) Multi-band blending

Gain compensation

- Simple gain adjustment
  - Compute average RGB intensity of each image in overlapping region
  - Normalize intensities by ratio of averages





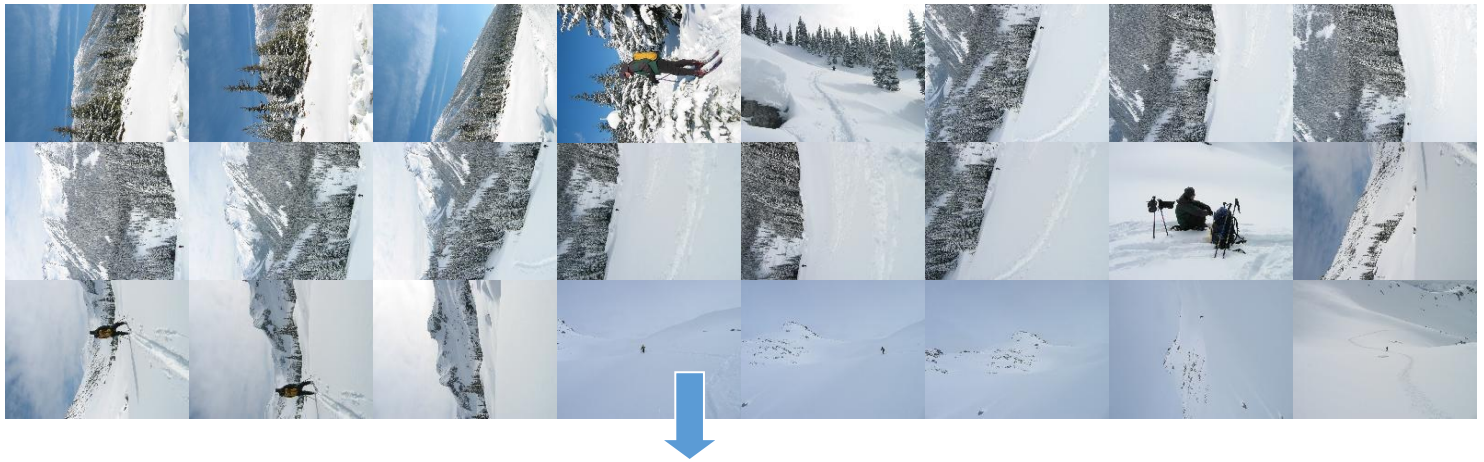
Blending Comparison

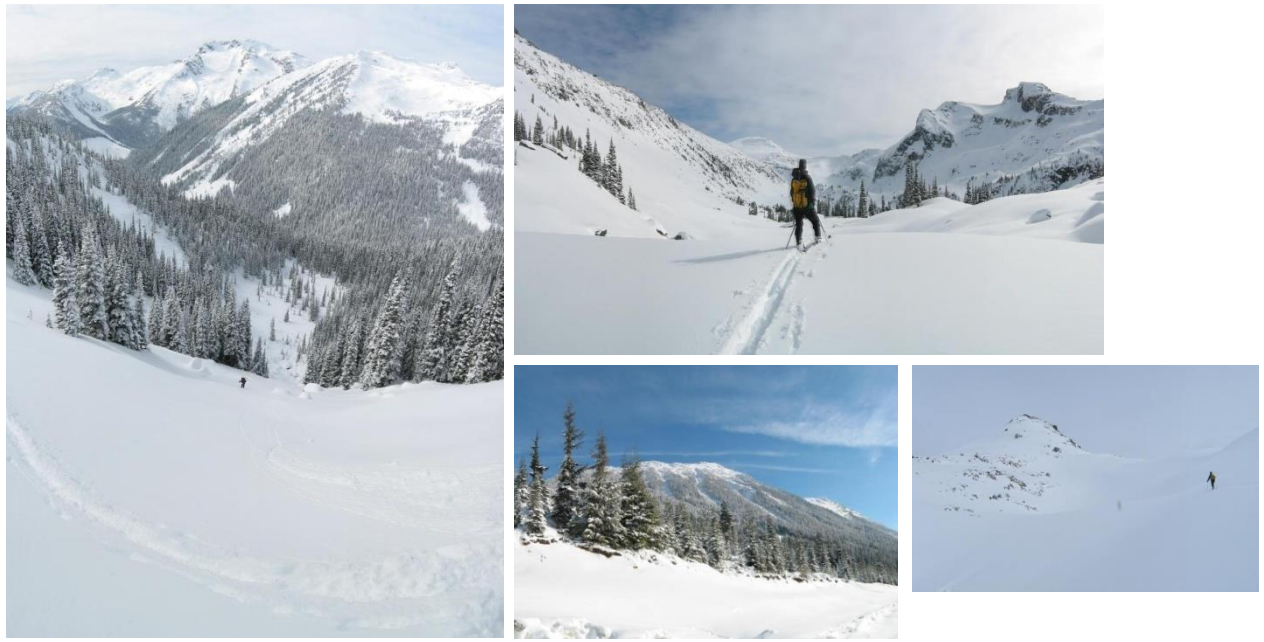
(b) Without gain compensation

(c) With gain compensation

(d) With gain compensation and multi-band blending

Recognizing Panoramas





Some of following material from Brown and Lowe 2003 talk

Brown and Lowe 2003, 2007

Recognizing Panoramas

- Input: N images
- Extract SIFT points, descriptors from all images
- Find K-nearest neighbors for each point (K=4)
- For each image
  - Select M candidate matching images by counting matched keypoints (m=6)
  - Solve homography Hij for each matched image

Recognizing Panoramas

- Input: N images
- Extract SIFT points, descriptors from all images
- Find K-nearest neighbors for each point (K=4)
- For each image
  - Select M candidate matching images by counting matched keypoints (m=6)
  - Solve homography Hij for each matched image
  - Decide if match is valid (ni > 8 + 0.3 nf)

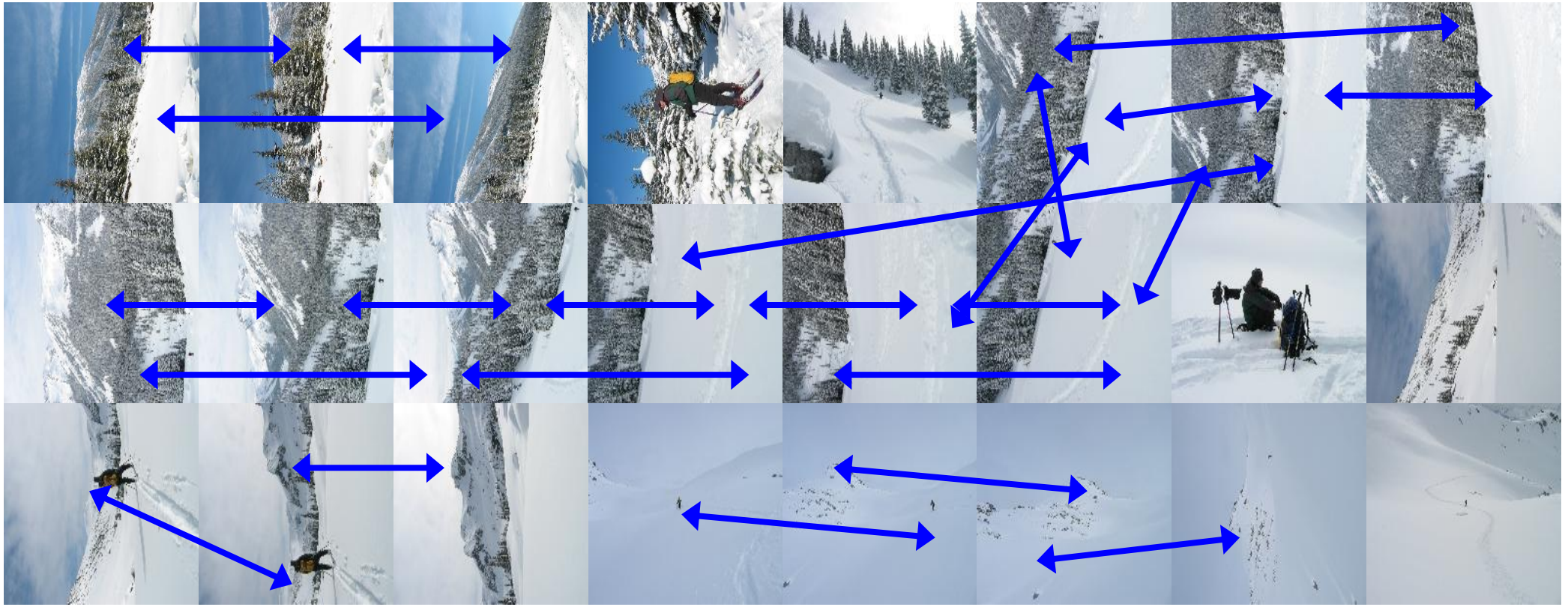
inliers

keypoints in ov
erlapping area

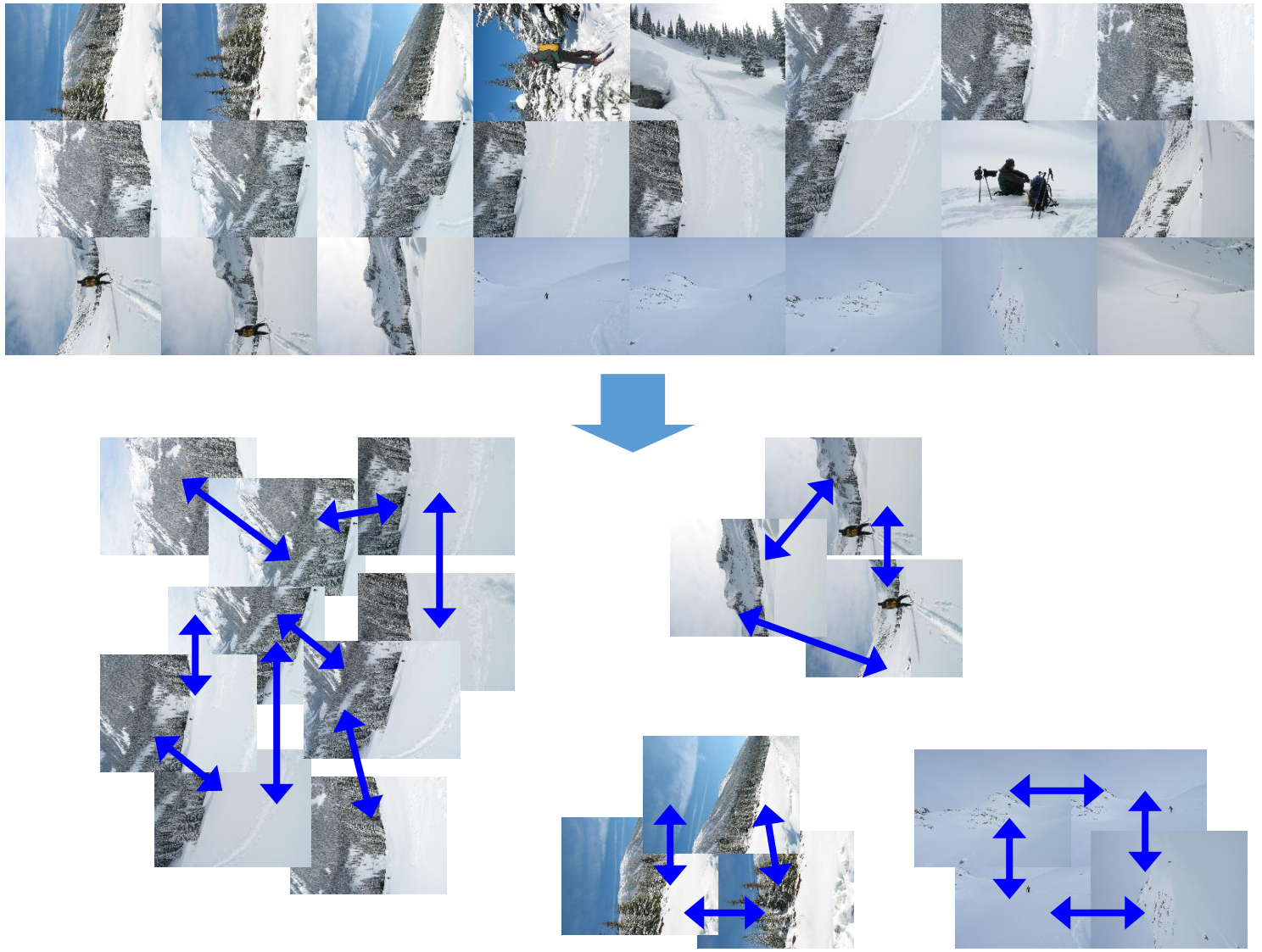
Recognizing Panoramas (cont.)

- (now we have matched pairs of images)
- Find connected components

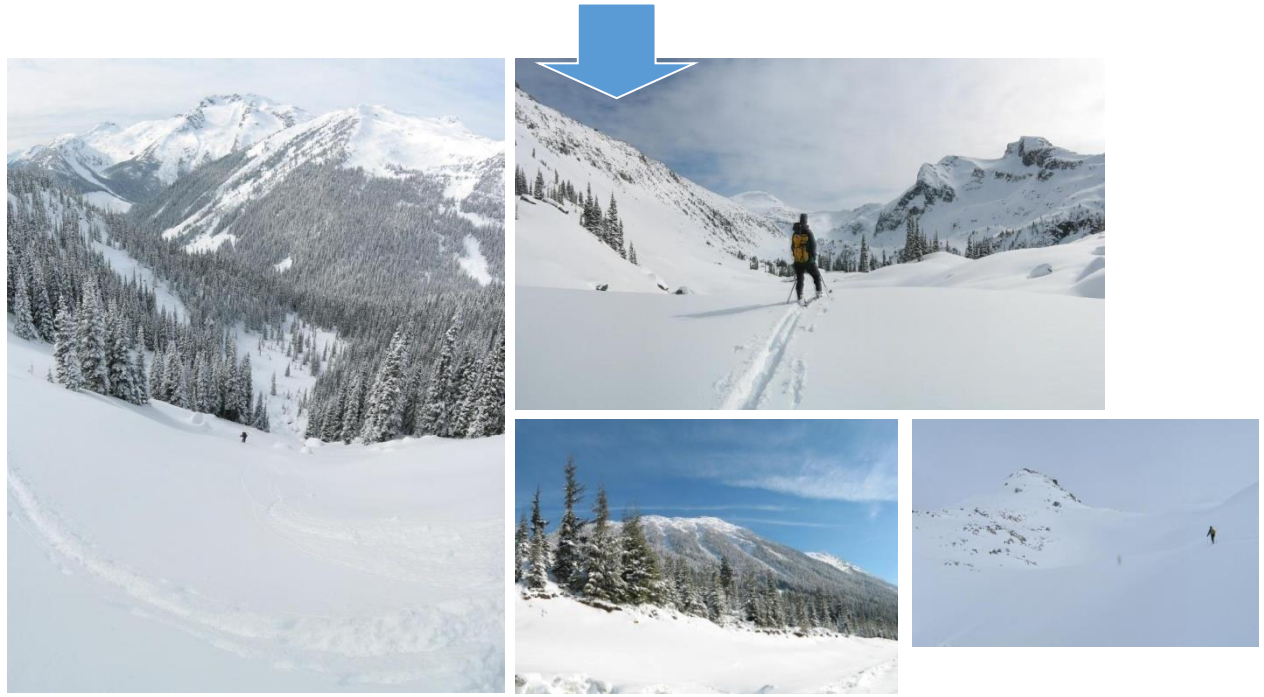
Finding the panoramas



Finding the panoramas



Finding the panoramas



Recognizing Panoramas (cont.)

- (now we have matched pairs of images)
- Find connected components
- For each connected component
  - Solve for rotation and f
  - Project to a surface (plane, cylinder, or sphere)
  - Render with multiband blending

Warping image using affine and perspective transformations

```python
import cv2
import numpy as np

img = cv2.imread('../data/circlesgrid.png', cv2.IMREAD_COLOR)
show_img = np.copy(img)
selected_pts = []


def mouse_callback(event, x, y, flags, param):
    global selected_pts, show_img
    if event == cv2.EVENT_LBUTTONUP:
        selected_pts.append([x, y])
        cv2.circle(show_img, (x, y), 10, (0, 255, 0), 3)


def select_points(image, points_num):
    global selected_pts
    selected_pts = []
    cv2.namedWindow('image')
    cv2.setMouseCallback('image', mouse_callback)

    while True:
        cv2.imshow('image', image)
        k = cv2.waitKey(1)
        if k == 27 or len(selected_pts) == points_num:
            break

    cv2.destroyAllWindows()
    return np.array(selected_pts, dtype=np.float32)
```

```python
show_img = np.copy(img)

src_pts = select_points(show_img, 3)
dst_pts = np.array([[0, 240], [0, 0], [240, 0]], dtype=np.float32)

affine_m = cv2.getAffineTransform(src_pts, dst_pts)
unwarped_img = cv2.warpAffine(img, affine_m, (240, 240))

cv2.imshow('result', np.hstack((show_img, unwarped_img)))
k = cv2.waitKey()
cv2.destroyAllWindows()

inv_affine = cv2.invertAffineTransform(affine_m)
warped_img = cv2.warpAffine(unwarped_img, inv_affine, (320, 240))
cv2.imshow('result', np.hstack((show_img, unwarped_img, warped_img)))
k = cv2.waitKey()
cv2.destroyAllWindows()

rotation_mat = cv2.getRotationMatrix2D(tuple(src_pts[0]), 6, 1)
rotated_img = cv2.warpAffine(img, rotation_mat, (240, 240))
cv2.imshow('result', np.hstack((show_img, rotated_img)))
k = cv2.waitKey()
cv2.destroyAllWindows()

show_img = np.copy(img)
src_pts = select_points(show_img, 4)
dst_pts = np.array([[0, 240], [0, 0], [240, 0], [240, 240]], dtype=np.float32)
perspective_m = cv2.getPerspectiveTransform(src_pts, dst_pts)
unwarped_img = cv2.warpPerspective(img, perspective_m, (240, 240))
cv2.imshow('result', np.hstack((show_img, unwarped_img)))
k = cv2.waitKey()
cv2.destroyAllWindows()
```

Remapping using arbitrary transformation

```python
import math
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('../data/Lena.png')

xmap = np.zeros((img.shape[1], img.shape[0]), np.float32)
ymap = np.zeros((img.shape[1], img.shape[0]), np.float32)
for y in range(img.shape[0]):
    for x in range(img.shape[1]):
        xmap[y,x] = x + 30 * math.cos(20 * x / img.shape[0])
        ymap[y,x] = y + 30 * math.sin(20 * y / img.shape[1])

remapped_img = cv2.remap(img, xmap, ymap, cv2.INTER_LINEAR, None, cv2.BORDER_REPLICATE)

plt.figure(0)
plt.axis('off')
plt.imshow(remapped_img[:,:,[2,1,0]])
plt.show()
```

Tracking keypoints between frames Lucas-Kanade algorithm

```python
import cv2
import numpy as np

video = cv2.VideoCapture('../data/traffic.mp4')
prev_pts = None
prev_gray_frame = None
tracks = None

while True:
    retval, frame = video.read()
    if not retval: break
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    if prev_pts is not None:
        pts, status, errors = cv2.calcOpticalFlowPyrLK(
            prev_gray_frame, gray_frame, prev_pts, None, winSize=(15,15), maxLevel=5,
            criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
        good_pts = pts[status == 1]
        if tracks is None: tracks = good_pts
        else: tracks = np.vstack((tracks, good_pts))
        for p in tracks:
            cv2.circle(frame, (p[0], p[1]), 3, (0, 255, 0), -1)
    else:
        pts = cv2.goodFeaturesToTrack(gray_frame, 500, 0.05, 10)
        pts = pts.reshape(-1, 1, 2)
    prev_pts = pts
    prev_gray_frame = gray_frame

    cv2.imshow('frame', frame)
    key = cv2.waitKey() & 0xff
    if key == 27: break
    if key == ord('c'):
        tracks = None
        prev_pts = None

cv2.destroyAllWindows()
```

Dense optical flow between two frames

```python
import cv2
import numpy as np


def display_flow(img, flow, stride=40):
    for index in np.ndindex(flow[::stride, ::stride].shape[:2]):
        pt1 = tuple(i*stride for i in index)
        delta = flow[pt1].astype(np.int32)[::-1]
        pt2 = tuple(pt1 + 10*delta)
        if 2 <= cv2.norm(delta) <= 10:
            cv2.arrowedLine(img, pt1[::-1], pt2[::-1],
                            (0,0,255), 5, cv2.LINE_AA, 0, 0.4)

    norm_opt_flow = np.linalg.norm(flow, axis=2)
    norm_opt_flow = cv2.normalize(norm_opt_flow, None, 0, 1,
                                  cv2.NORM_MINMAX)

    cv2.imshow('optical flow', img)
    cv2.imshow('optical flow magnitude', norm_opt_flow)
    k = cv2.waitKey(1)

    if k == 27:
        return 1
    else:
        return 0
```

```python
cap = cv2.VideoCapture("../data/traffic.mp4")
_, prev_frame = cap.read()

prev_frame = cv2.cvtColor(prev_frame, cv2.COLOR_BGR2GRAY)
prev_frame = cv2.resize(prev_frame, (0,0), None, 0.5, 0.5)
init_flow = True

while True:
    status_cap, frame = cap.read()
    frame = cv2.resize(frame, (0,0), None, 0.5, 0.5)
    if not status_cap:
        break
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    if init_flow:
        opt_flow = cv2.calcOpticalFlowFarneback(
            prev_frame, gray, None, 0.5, 5, 13, 10,
            5, 1.1, cv2.OPTFLOW_FARNEBACK_GAUSSIAN)
        init_flow = False
    else:
        opt_flow = cv2.calcOpticalFlowFarneback(
            prev_frame, gray, opt_flow, 0.5, 5, 13,
            10, 5, 1.1, cv2.OPTFLOW_USE_INITIAL_FLOW)

    prev_frame = np.copy(gray)

    if display_flow(frame, opt_flow):
        break;
```

```python
cv2.destroyAllWindows()

cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
_, prev_frame = cap.read()

prev_frame = cv2.cvtColor(prev_frame, cv2.COLOR_BGR2GRAY)
prev_frame = cv2.resize(prev_frame, (0,0), None, 0.5, 0.5)

flow_DualTVL1 = cv2.createOptFlow_DualTVL1()

while True:
    status_cap, frame = cap.read()
    frame = cv2.resize(frame, (0,0), None, 0.5, 0.5)
    if not status_cap:
        break
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    if not flow_DualTVL1.getUseInitialFlow():
        opt_flow = flow_DualTVL1.calc(prev_frame, gray, None)
        flow_DualTVL1.setUseInitialFlow(True)
    else:
        opt_flow = flow_DualTVL1.calc(prev_frame, gray, opt_flow)

    prev_frame = np.copy(gray)

    if display_flow(frame, opt_flow):
        break;

cv2.destroyAllWindows()
```

Panorama image using many images

```python
import cv2
import numpy as np

images = []
images.append(cv2.imread('../data/panorama/0.jpg', cv2.IMREAD_COLOR))
images.append(cv2.imread('../data/panorama/1.jpg', cv2.IMREAD_COLOR))

stitcher = cv2.createStitcher()
ret, pano = stitcher.stitch(images)

if ret == cv2.STITCHER_OK:
    pano = cv2.resize(pano, dsize=(0, 0), fx=0.2, fy=0.2)
    cv2.imshow('panorama', pano)
    cv2.waitKey()

    cv2.destroyAllWindows()
else:
    print('Error during stiching')
```