Lecture 15:

Image Processing Algorithm Grab Bag

Visual Computing Systems
CMU 15-869, Fall 2013

Today

- **A grab bag of basic image-processing techniques**
- **Goals:**
  - Provide an overview of solution strategies to select image processing problems
  - Provide a flavor of the types of operations future image signal processors (ISPs) will need to perform

Simple noise reduction techniques

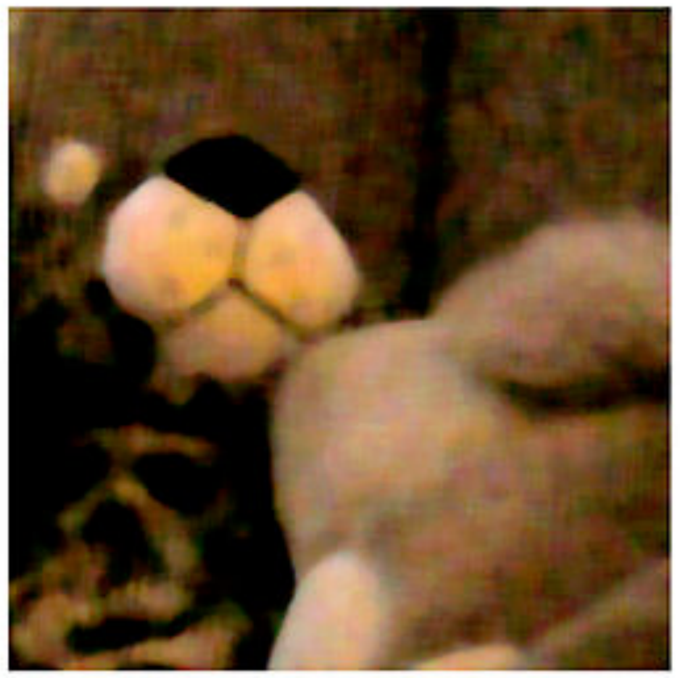
Median filter

- Noise reduction filter
  - Unlike gaussian blur, one bright pixel doesn't drag up the average for entire region
- Not linear, not separable
  - Filter weights are 1 or 0 (depending on image content)
- Naive algorithm for width-N square kernel support region:
  - Sort N^2 elements in support region, pick median: $O(N^2 \log(N^2))$ work per pixel

```
int WIDTH = 1024;
int HEIGHT = 1024;
uint8 input[(WIDTH+2) * (HEIGHT+2)];
uint8 output[WIDTH * HEIGHT];

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    output[j*WIDTH + i] = // median of pixels in
                          // surrounding 5x5 pixel window
  }
}
```

original image

1px median filter



3px median filter

10px median filter

5x5 median filter

- $O(N^2)$ work-per-pixel solution: radix sort 8 bit-integer data
  - Bin all pixels in support region, then scan histogram to find median

```
int WIDTH = 1024;
int HEIGHT = 1024;
uint8 input[(WIDTH+2) * (HEIGHT+2)];
uint8 output[WIDTH * HEIGHT];
int histogram[256];

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    for (int ii=0; ii<256; ii++)
      histogram[ii] = 0;
    for (int jj=0; jj<5; jj++)
      for (int ii=0; ii<5; ii++)
        histogram[input[(j+jj)*(WIDTH+2) + (i+ii)]]++;
    int count = 0;
    for (int ii=0; ii<256; i++) {          // scan all 256 bins
       if (count + histogram[i] >= 13)     // median of 25 elements is bin containing 13th value
         output[j*WIDTH + i] = uint8(i);
       count += histogram[i];
    }
  }
}
```

Can you think of how to modify this code to implement a O(N) work-per-pixel median filter?

See Weiss [SIGGRAPH 2006] for O(lg N) work-per-pixel median filter

Bilateral filter

Example use of bilateral filter: removing noise while preserving image edges

Bilateral filter

$$BF[I](p) = \sum_{q \in S} f(|I_p - I_q|) G_\sigma(\|p-q\|) I(q)$$

Output pixel *p* is the weighted sum of all pixels in the support region *S* of a truncated gaussian kernel (width σ)

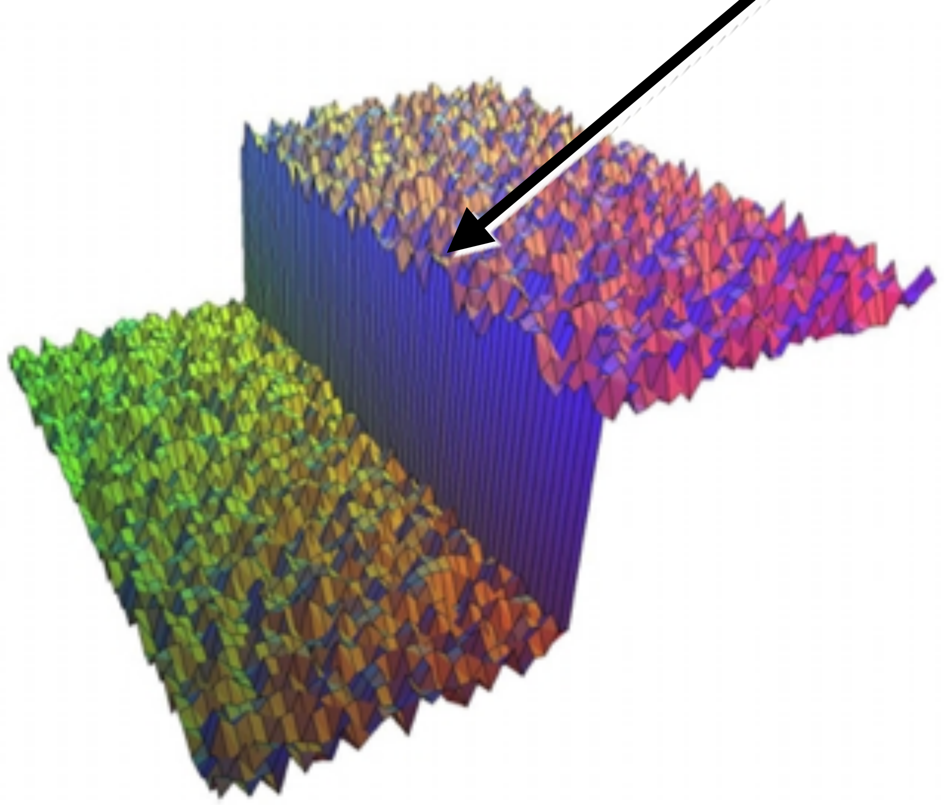
But weight is combination of <u>spatial distance</u> and <u>input image pixel intensity</u> difference. (non-linear filter: like the median filter, the filter's weights depend on input image content)

- **An "edge preserving" filter: down-weight contribution of pixels on the other side of strong edges.** $f(x)$ **defines what "strong edge means"**
- **Spatial distance weight term** $f(x)$ **could be a gaussian**
  - **Or very simple:** $f(x) = 0$ *if x > threshold, 1 otherwise*

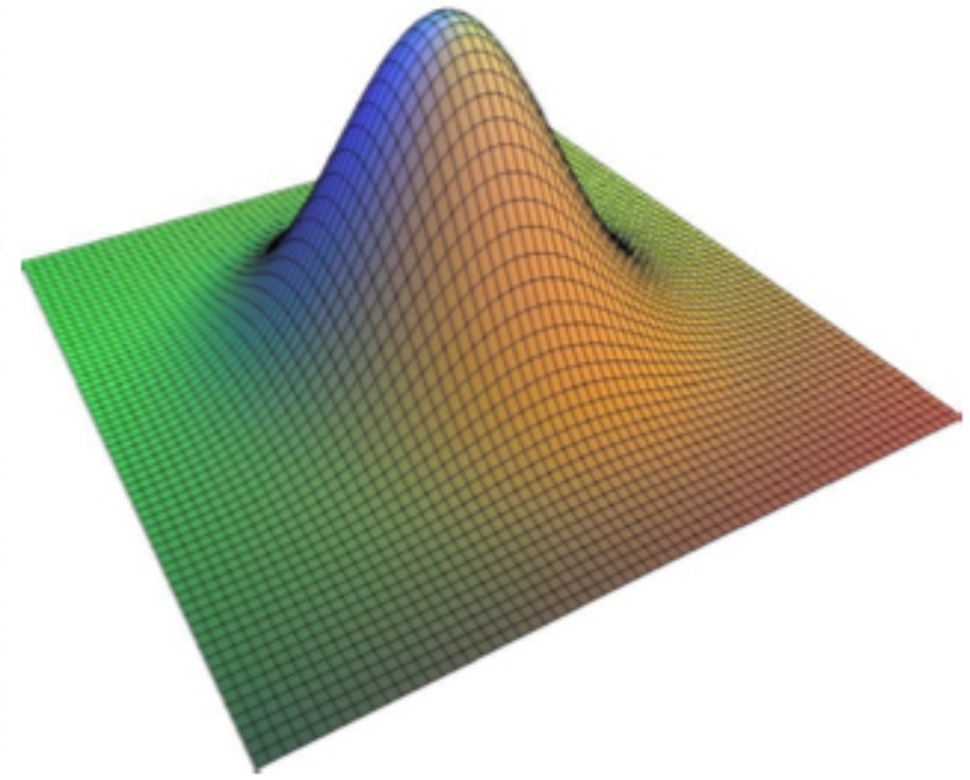
Bilateral filter

Input pixel p

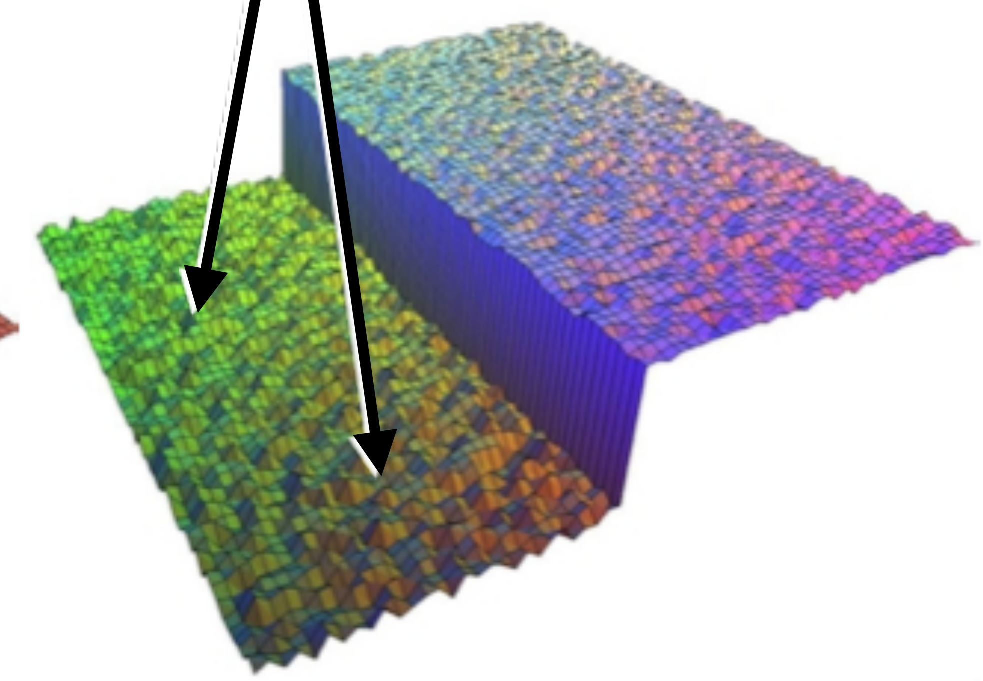
Pixels with significantly different intensity as p contribute little to filtered result (they are “on the “other side of the edge”



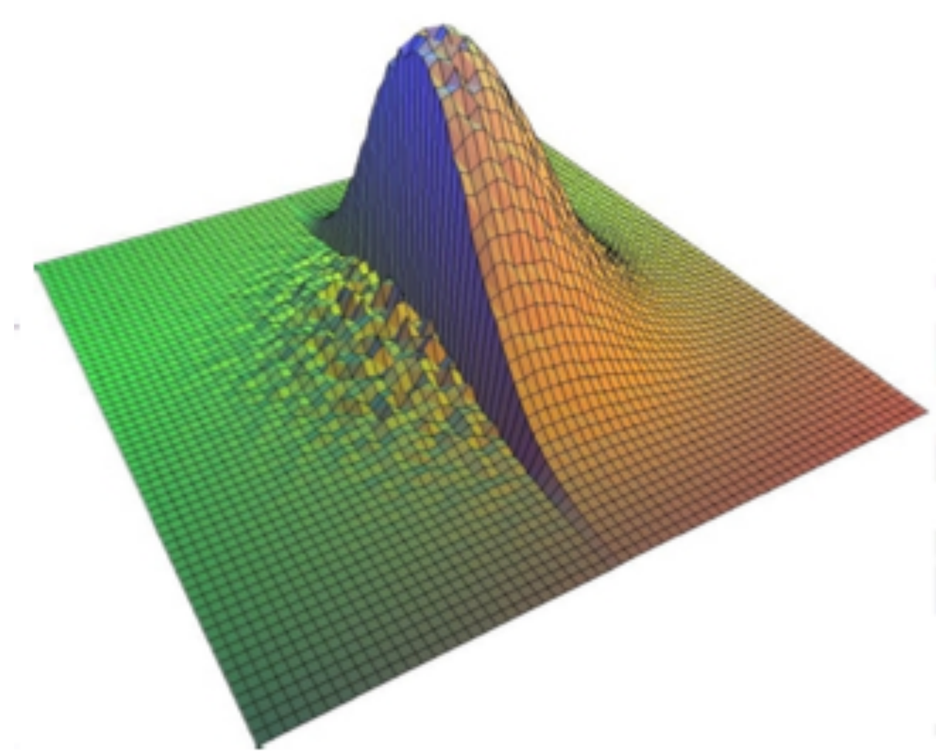
Input image



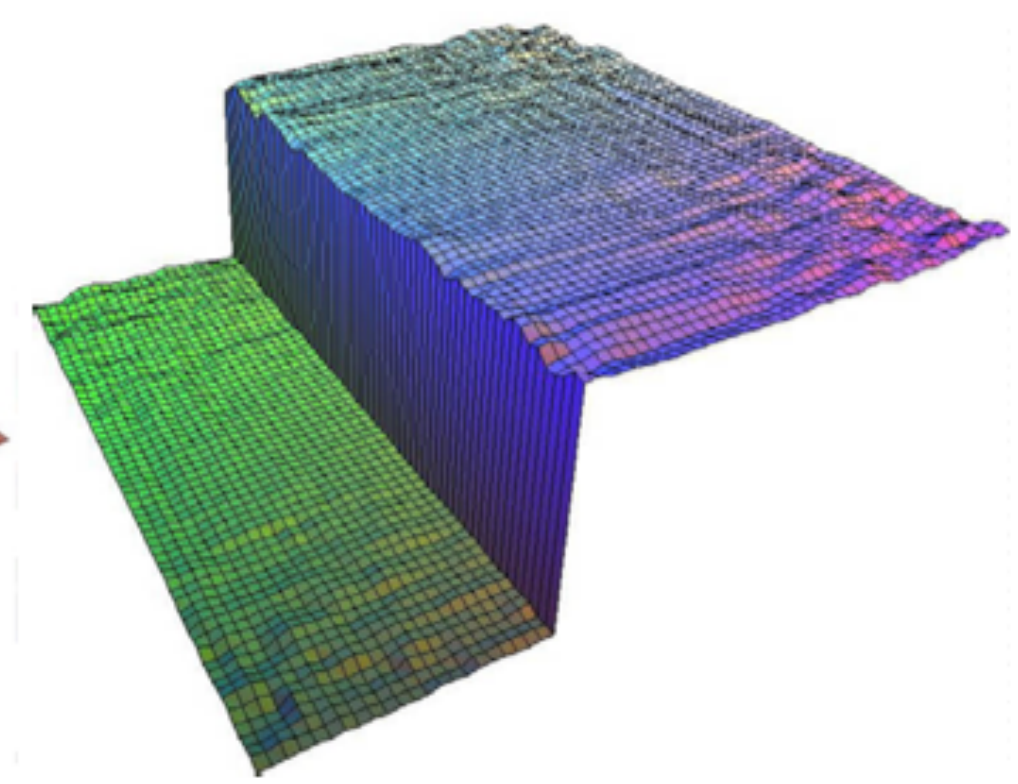
G(): Gaussian about input pixel p



f(): Influence of support region



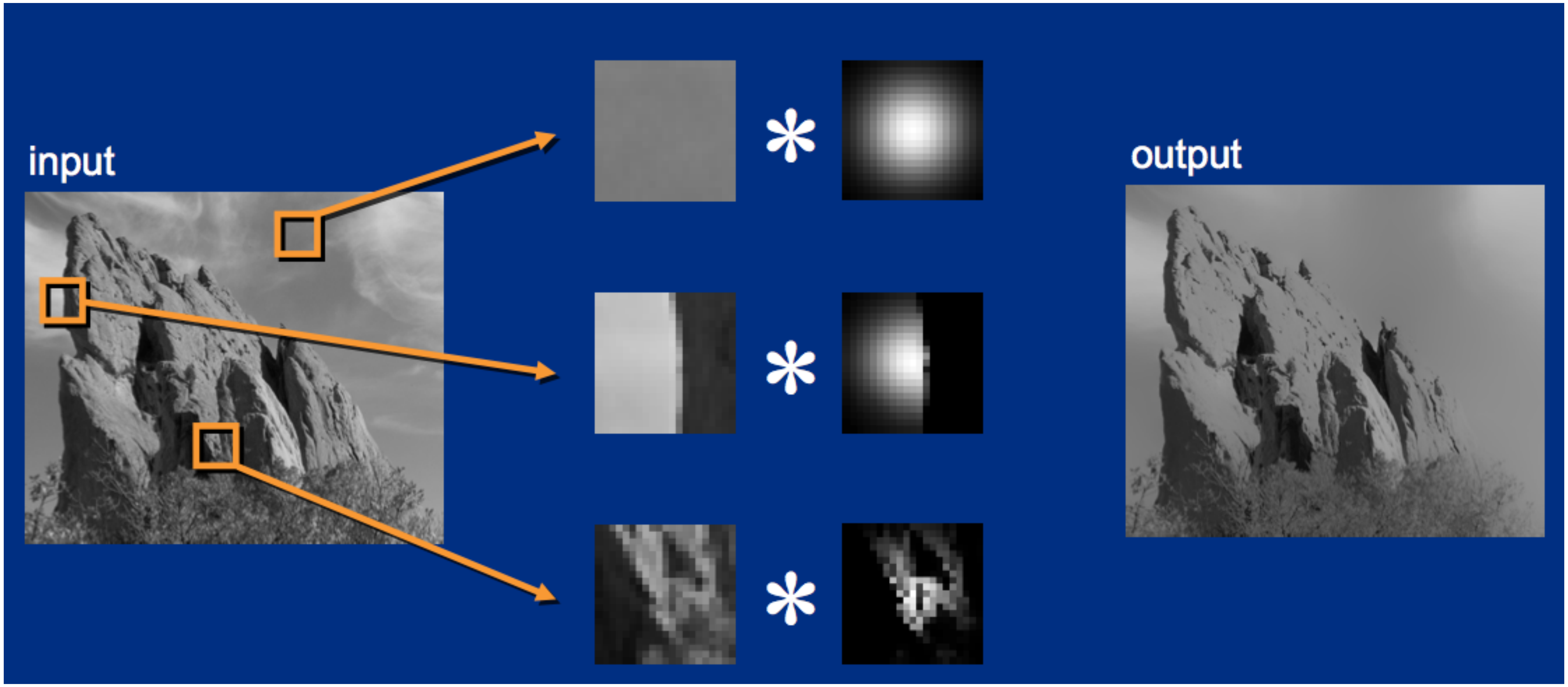
G x f: filter weights for pixel p



Filtered output image

Figure credit: Durand and Dorsey, “Fast Bilateral Filtering for the Display of High-Dynamic-Range Images”, SIGGRAPH 2002

Bilateral filter: kernel depends on image content



See Paris et al. [ECCV 2006] for a fast approximation to the bilateral filter

Question: describe a type of edge the bilateral filter will not respect (it will blur across).

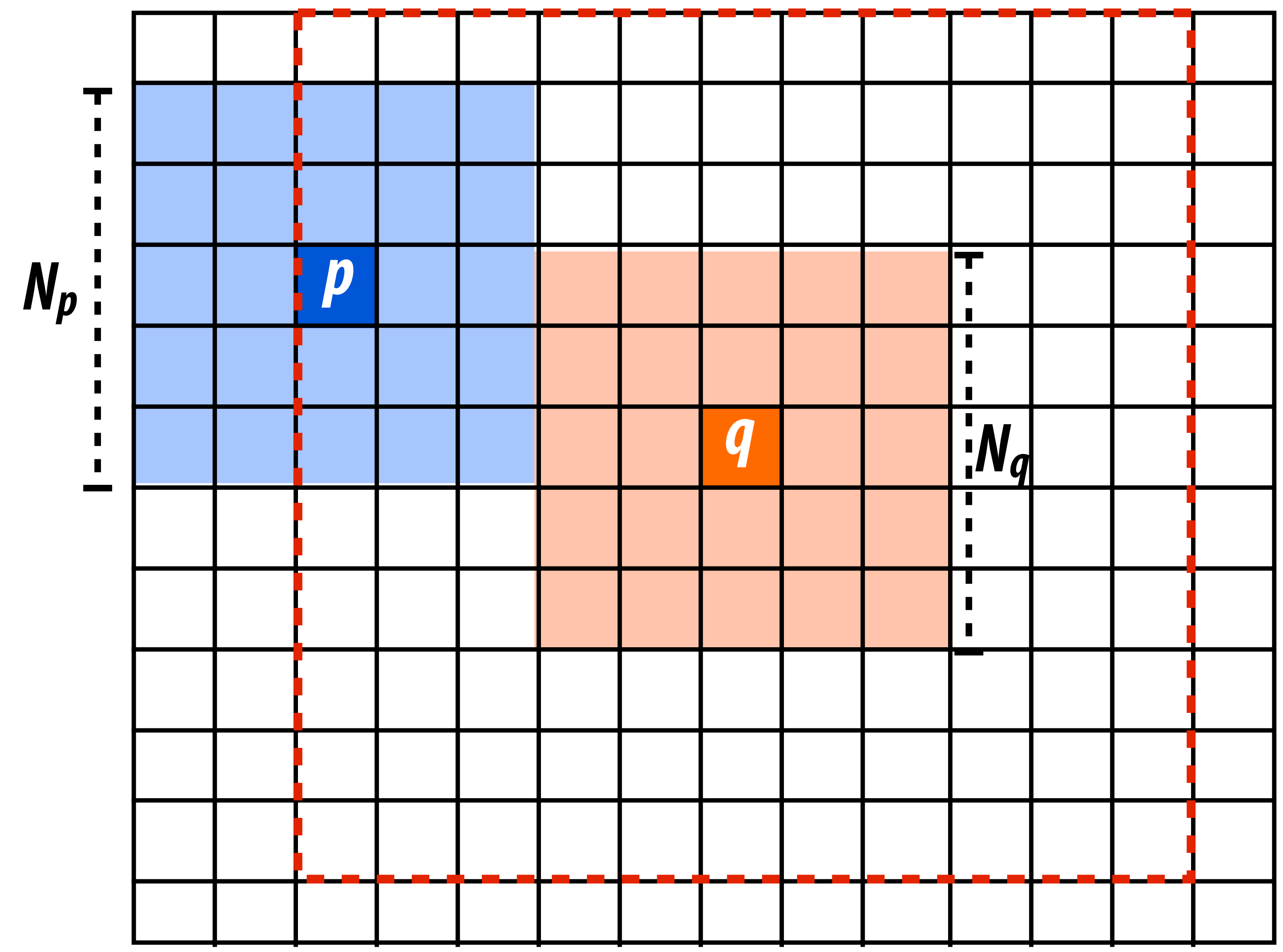
Figure credit: SIGGRAPH 2008 Course: "A Gentle Introduction to Bilateral Filtering and its Applications" Paris et al.

Denoising using non-local means

- **Main idea: replace pixel with average value of nearby pixels that have a similar surrounding region.**
  - **Assumption: images have repeating texture**

$$NL[I](p) = \sum_{q \in S} w(p,q) I(q)$$

$$w(p,q) = \frac{1}{C_p} e^{\frac{-\left\|N_p - N_q\right\|_2^2}{h^2}}$$

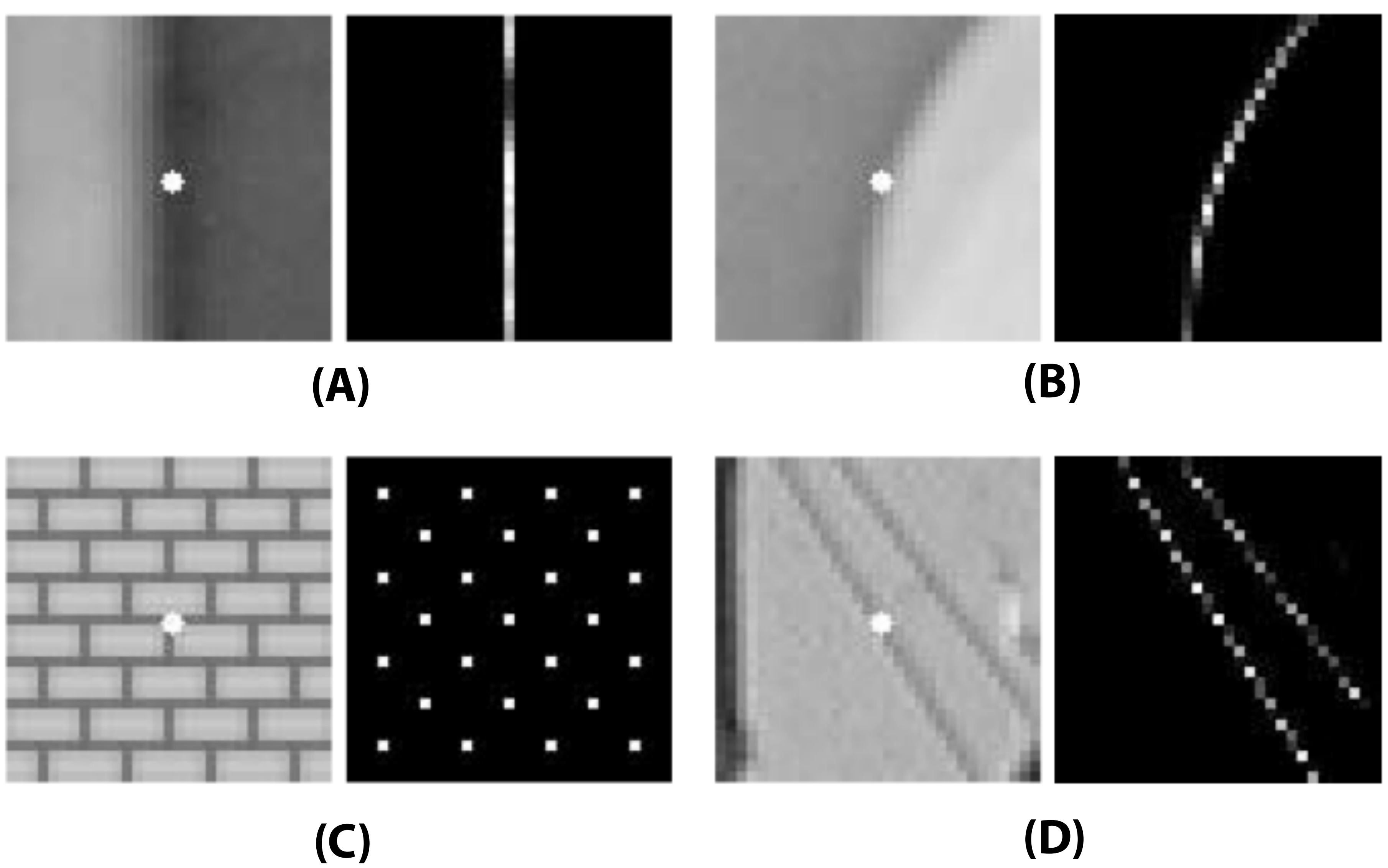


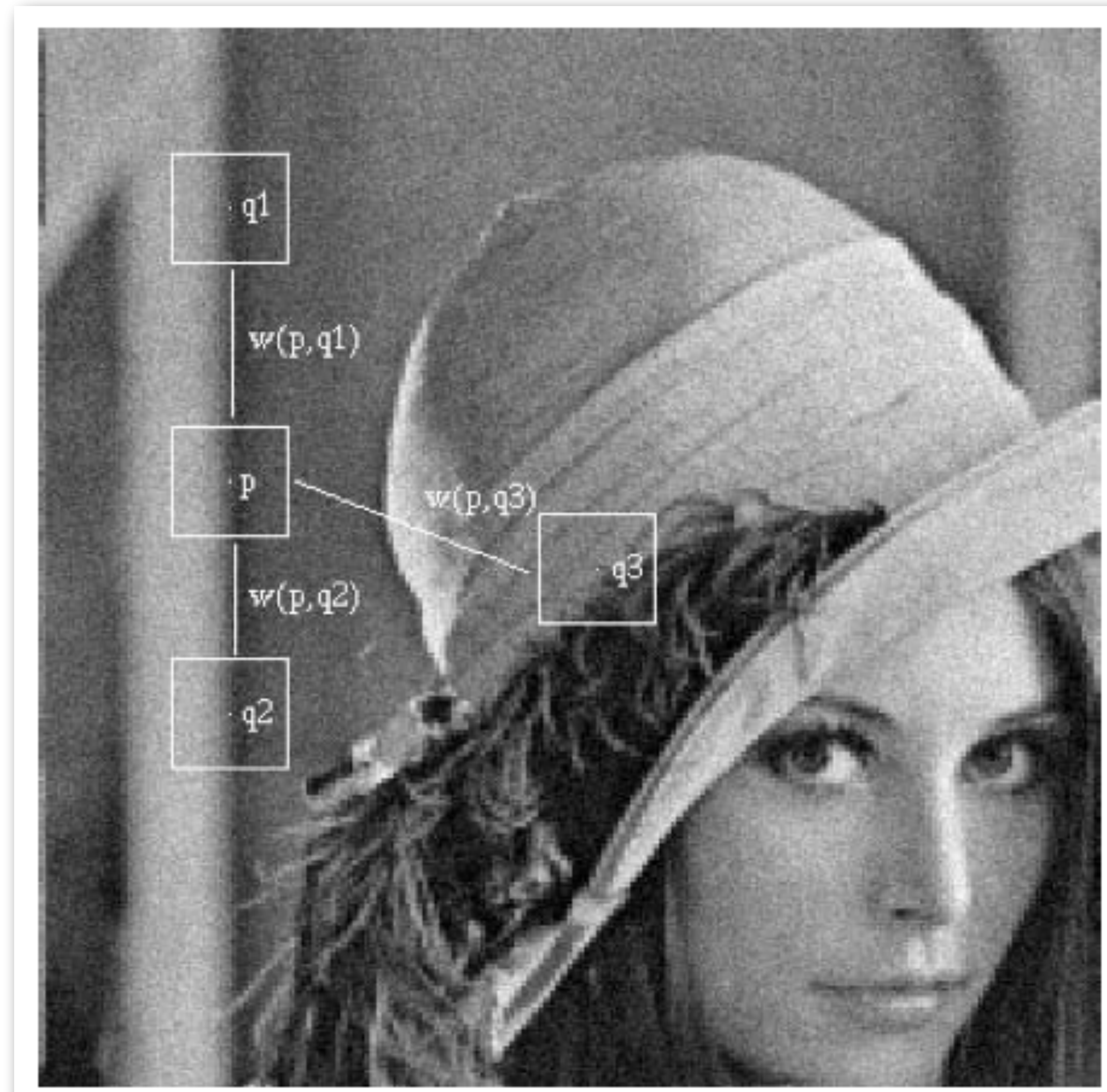
- N_p and P_q are vectors of pixel values in square window around pixels p and q (highlighted regions in figure)
- Difference N_p and P_q = "similarity" of surrounding regions
- Cp is just a normalization constant to ensure weights sum to one for pixel p.
- Set S is the search region (given by dotted red line in figure)

Denoising using non-local means

- Large weight for input pixels that have similar neighborhood as *p*
  - Intuition: "filtered result is the average of pixels "like" this one"
  - In example below-right: *q1* and *q2* have high weight, *q3* has low weight

In each image pair below:
- **Image at left shows the pixel to denoise.**
- **Image at right shows weights of pixels in 21x21-pixel kernel support window.**





Buades et al. CVPR 2005

Optical flow

Optical flow

- **Goal: determine 2D screen-space velocity of visible objects in image**

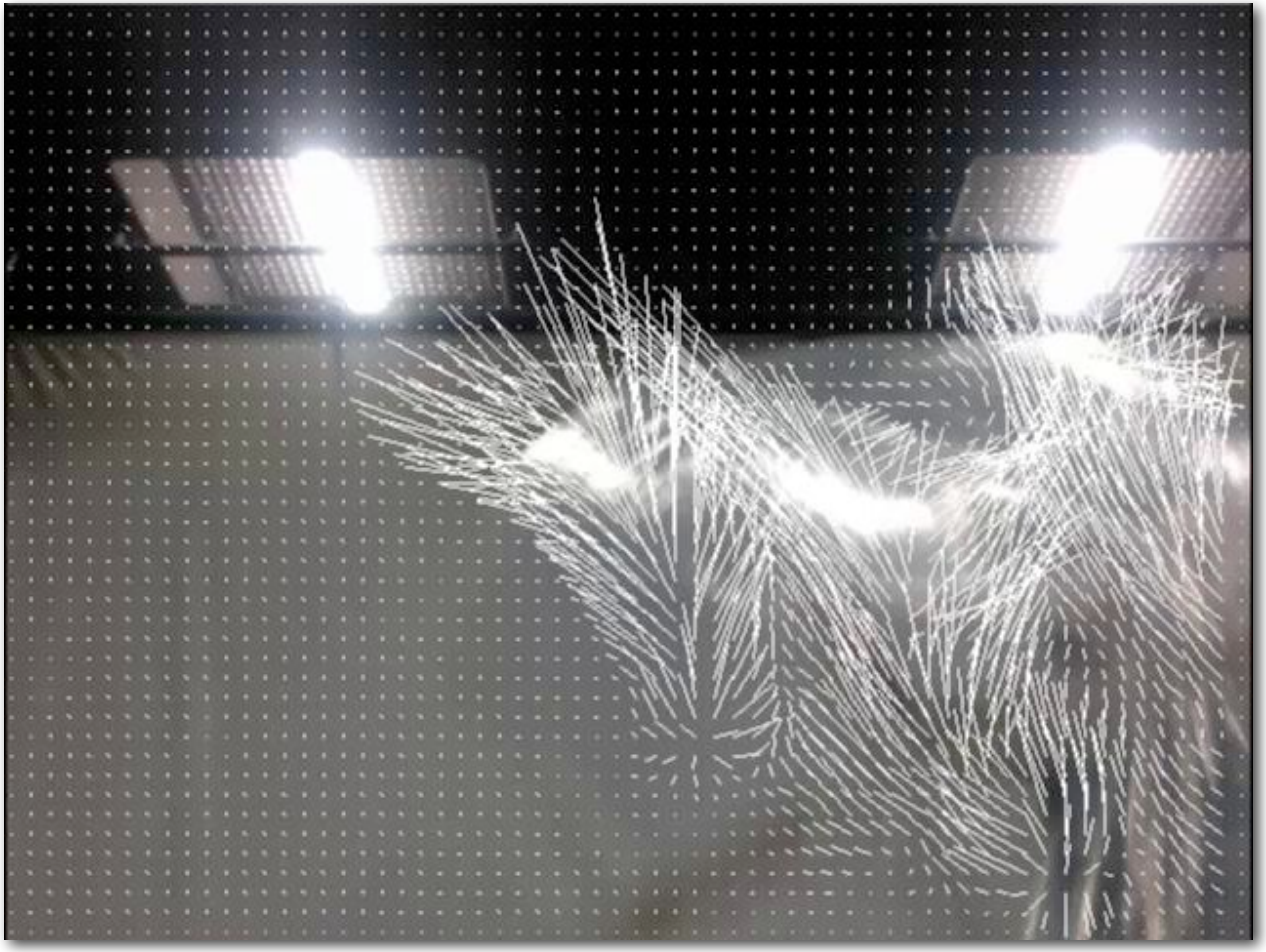


Image source: https://vimeo.com/28395792

Optical flow

- Given image A (at time *t*) and image B (at time *t* + Δ*t*) compute optical flow between the two images
- Major assumption 1: "brightness constancy"
  - The appearance of a scene surface point that is visible in both images A and B is the same in both images

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$$

The point observed at (*x*,*y*) at time *t* moves to (*x*+Δ, *y*+Δ) at *t*+Δ*t*, (and has a constant appearance in both situations)

Tailor expansion

$$I(x + \Delta x, y + \Delta y, t + \Delta t) = I(x, y, t) + I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y + I_t(x, y, t)\Delta t + \text{higher order terms}$$

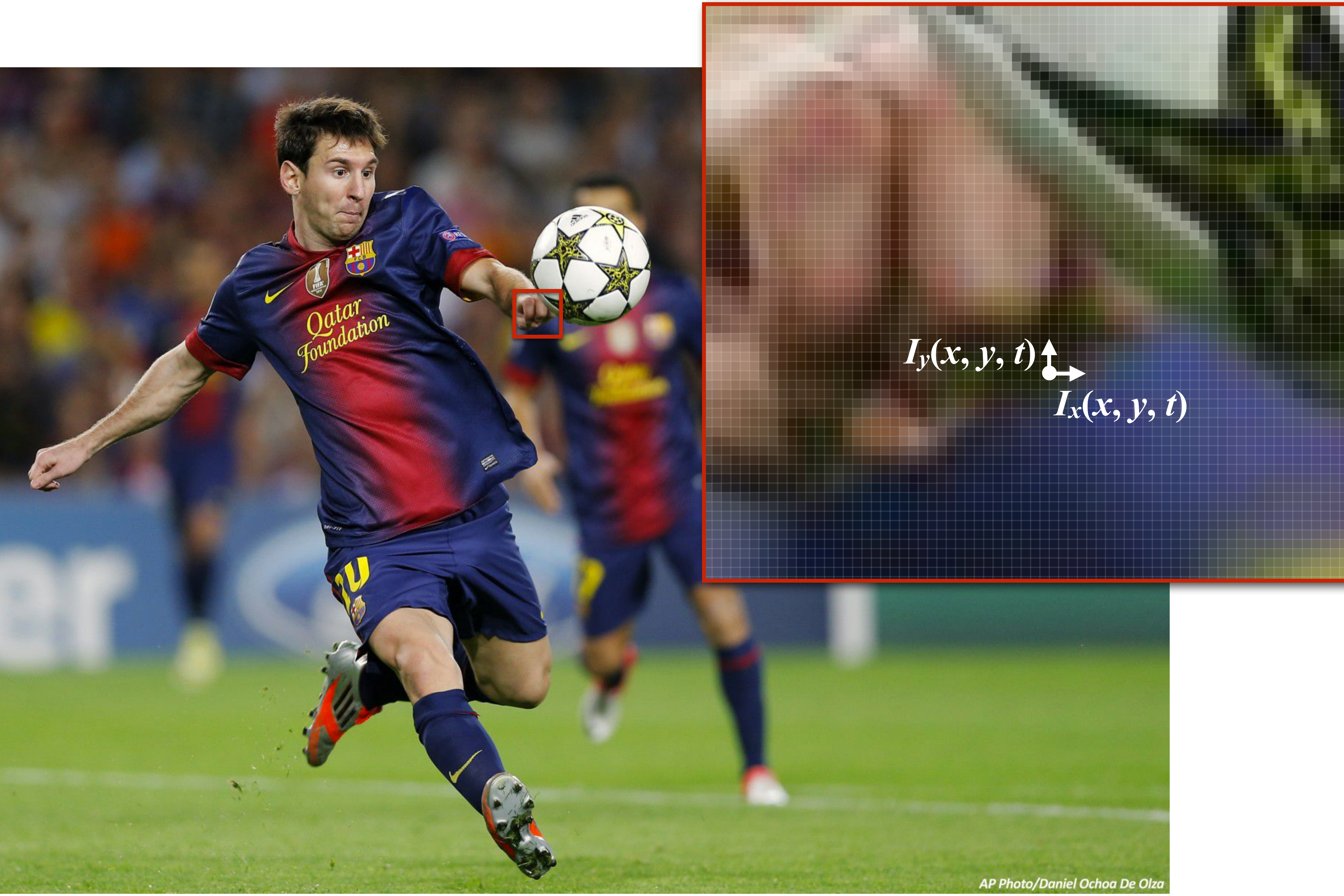
So...

$$I(x, y, t) \approx I(x, y, t) + I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y + I_t(x, y, t)\Delta t$$

$$\boxed{I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y} + \boxed{I_t(x, y, t)\Delta t} = 0$$

The observed change in pixel (x,y)

Is due to object motion at point by (Δx, Δy)



$I_y(x, y, t)$
$I_x(x, y, t)$
AP Photo/Daniel Ochoa De Olza

Gradient-constraint equation for a pixel is underconstrained

Gradient-constraint equation is insufficient to solve for motion

One equation, two unknowns: (Δx, Δy)

$$\boxed{I_x(x, y, t)}\Delta x + \boxed{I_y(x, y, t)}\Delta y + \boxed{I_t(x, y, t)\Delta t} = 0$$

Known: observed change in pixel (x,y) over consecutive frames

Known: spatial image gradients in image A

Major assumption 2: nearby pixels have similar motion (Lucas-Kanade)

$$I_x(x_0, y_0, t)\Delta x + I_y(x_0, y_0, t)\Delta y + I_t(x_0, y_0, t)\Delta t = 0$$

$$I_x(x_1, y_1, t)\Delta x + I_y(x_1, y_1, t)\Delta y + I_t(x_1, y_1, t)\Delta t = 0$$

$$I_x(x_2, y_2, t)\Delta x + I_y(x_2, y_2, t)\Delta y + I_t(x_2, y_2, t)\Delta t = 0$$

$$\vdots$$

Now we have a overconstrained system, compute least squares solution

Weighted least-squares solution

$$I_x(x_0, y_0, t)\Delta x + I_y(x_0, y_0, t)\Delta y + I_t(x_0, y_0, t)\Delta t = 0$$

$$I_x(x_1, y_1, t)\Delta x + I_y(x_1, y_1, t)\Delta y + I_t(x_1, y_1, t)\Delta t = 0$$

$$I_x(x_2, y_2, t)\Delta x + I_y(x_2, y_2, t)\Delta y + I_t(x_2, y_2, t)\Delta t = 0$$

$$\vdots$$

Compute weighted least squares solution by minimizing:

(x_i, y_i) are pixels in region around (x,y).

Weighting function w() weights error contribution based on distance between (x_i, y_i) and (x, y). e.g., Gaussian fall-off.

$$E(\Delta x, \Delta y) = \sum_{x_i, y_i} w(x_i, y_i, x, y)\left[I_x(x_i, y_i, t)\Delta x + I_y(x_i, y_i, t)\Delta y + I_t(x_i, y_i, t)\Delta t \right]^2$$

Solving for motion

E (Δx, Δy) minimized when derivatives are zero:

$$\frac{dE(\Delta x,\Delta y)}{d(\Delta x)} = \sum_{x_i,y_i} w(x_i,y_i,x,y)\left[I_x^2\Delta x + I_xI_y\Delta y + I_xI_t\right] = 0$$

$$\frac{dE(\Delta x,\Delta y)}{d(\Delta y)} = \sum_{x_i,y_i} w(x_i,y_i,x,y)\left[I_y^2\Delta y + I_xI_y\Delta x + I_yI_t\right] = 0$$

Rewrite, now solve the following linear system for Δx, Δy:

$$\Delta x \underbrace{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_x^2}}_{A0} + \Delta y \underbrace{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_xI_y}}_{B0} + \underbrace{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_xI_t}}_{C0} = 0$$

$$\Delta x \underbrace{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_xI_y}}_{A1} + \Delta y \underbrace{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_y^2}}_{B1} + \underbrace{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_yI_t}}_{C1} = 0$$

Precompute partial derivatives I_x, I_y, I_t from original images A and B

For each pixel (*x*,*y*): evaluate A0, B0, C0, A1, B1, C1, then solve for (Δ*x*, Δ*y*) at (*x*,*y*)

Optical flow, implemented in practice

Gradient-constraint equation makes a linear motion assumption

$$I(x, y, t) \approx I(x, y, t) + I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y + I_t(x, y, t)\Delta t$$

$$\boxed{I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y} + \boxed{I_t(x, y, t)\Delta t} = 0$$

The observed change in pixel (x,y)

Is due to object motion at point by (Δx, Δy)

- **Improvement: iterative techniques use this original flow field to compute higher order residuals (to estimate non-linear motion)**

- **Question: why is it important for optical flow implementation to be very efficient?**
  - **Hint: consider linear-motion assumption**

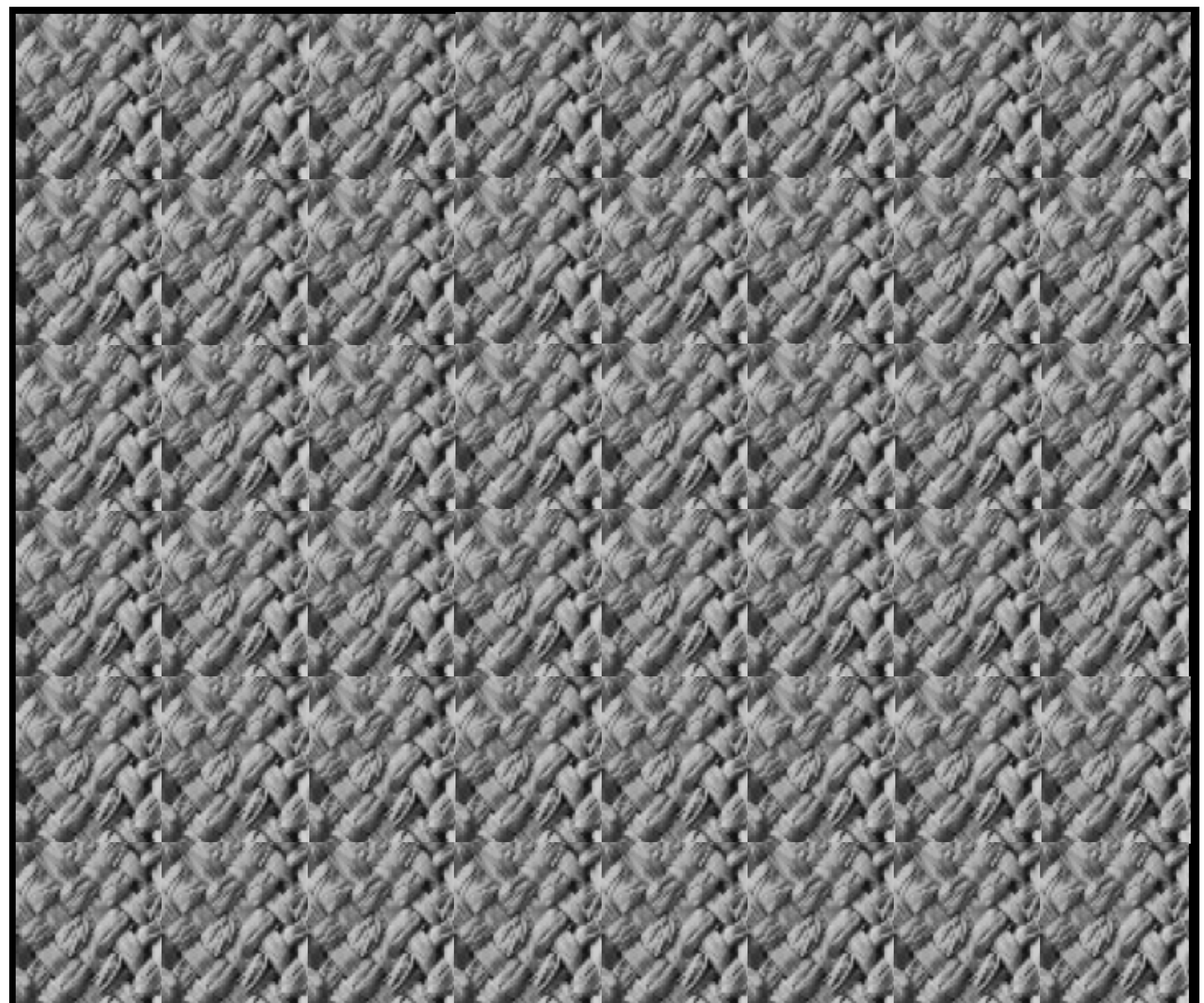
Image manipulation by example

Data-driven texture synthesis

- Input: low resolution texture image
- Desired output: high resolution texture that appears "like" the input

Source texture (low resolution)

High-resolution texture generated by tiling low-resolution texture

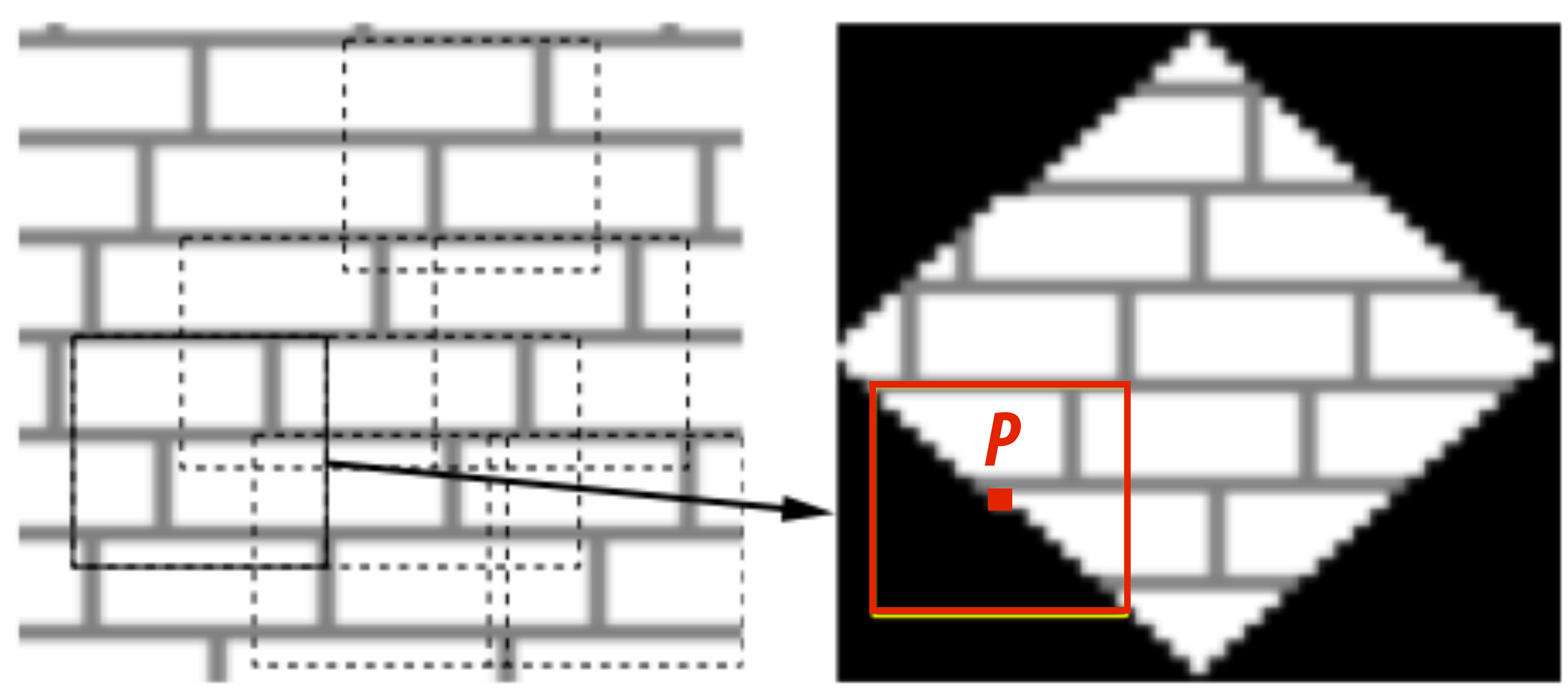


Algorithm: non-parametric texture synthesis

Main idea: given NxN neighborhood *w*(*p*) around unknown pixel *p*, want probability distribution function for possible values of *p*, given values of neighborhood *w*(*p*) around *p*:

P(p=X | *w*(*p*))

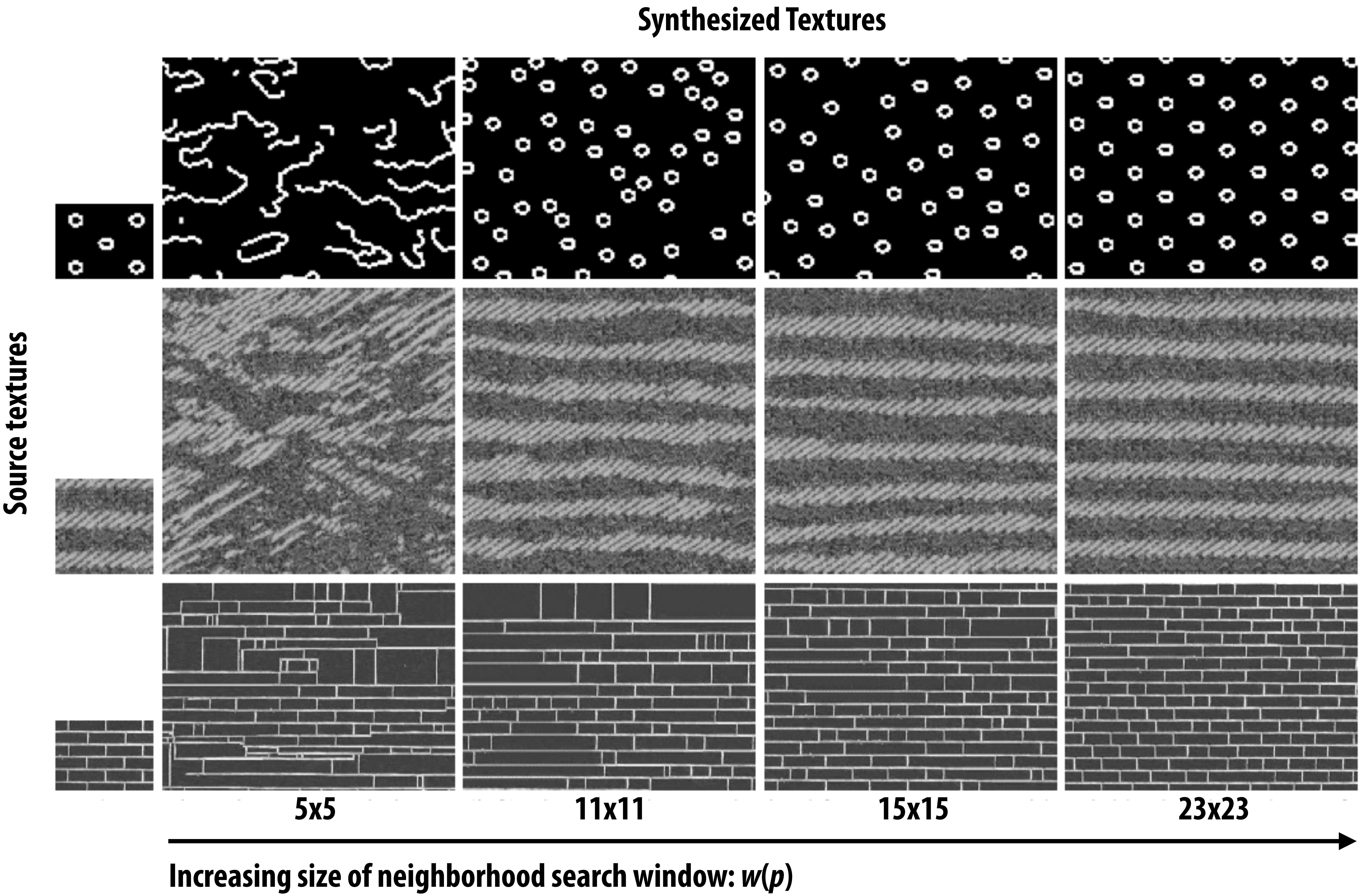
```
For each pixel p to synthesize:

    1.  Find other patches in the image that are similar to the NxN
        neighborhood around p (use gaussian weighted sum-of-squared-
        differences as the patch distance function)

    2.  Center pixels of closest patches are candidates for p

    3.  Randomly sample from candidates weighted by distance d
```



[Efros and Leung 99]

Non-parametric texture synthesis

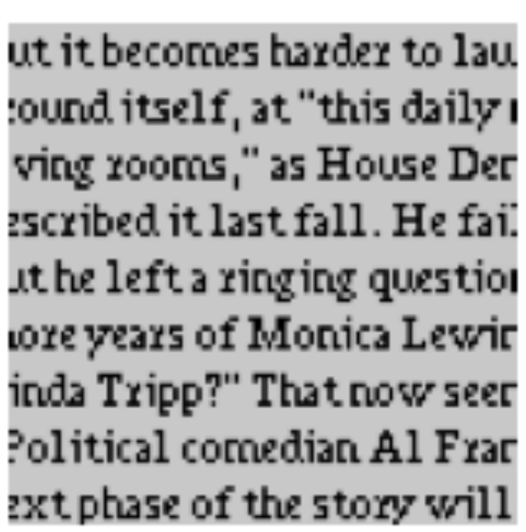
[Efros and Leung 99]



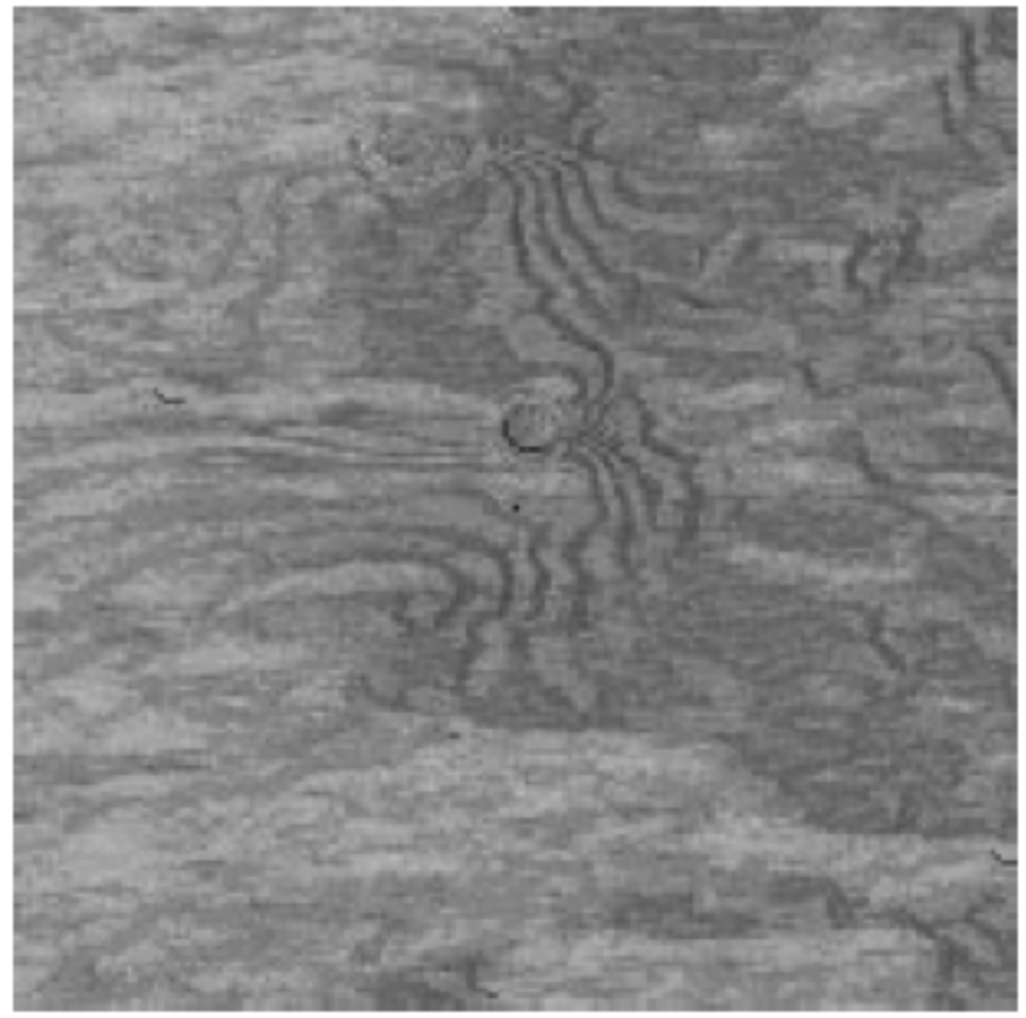
More texture synthesis examples

[Efros and Leung 99]

Source textures



Synthesized Textures



Naive tiling solution

Image completion example

Image credit: [Barnes et al. 2009]

Original Image

Masked Region

Completion Result

Problem: high computational cost

- **Large patch windows + full image search = slow**
  - Want large patch windows: preserve image structure
  - Want full-image search: highly relevant examples are rare
- **Must perform search process for all pixels to fill in**
  - Naive algorithm:

```
For each pixel p to fill in:
   For each pixel pi in image:
      Compute distance between neighborhoods of p and pi.
```

- **Possible acceleration techniques?**
  - Limit search window (reduces output quality — may miss relevant examples)
  - Use acceleration structure for search (e.g., k-d tree)
  - Reduce dimensionality of patches + approximate nearest neighbor search (ANN)
  - Exploit spatial coherence of pixel values in images

PatchMatch

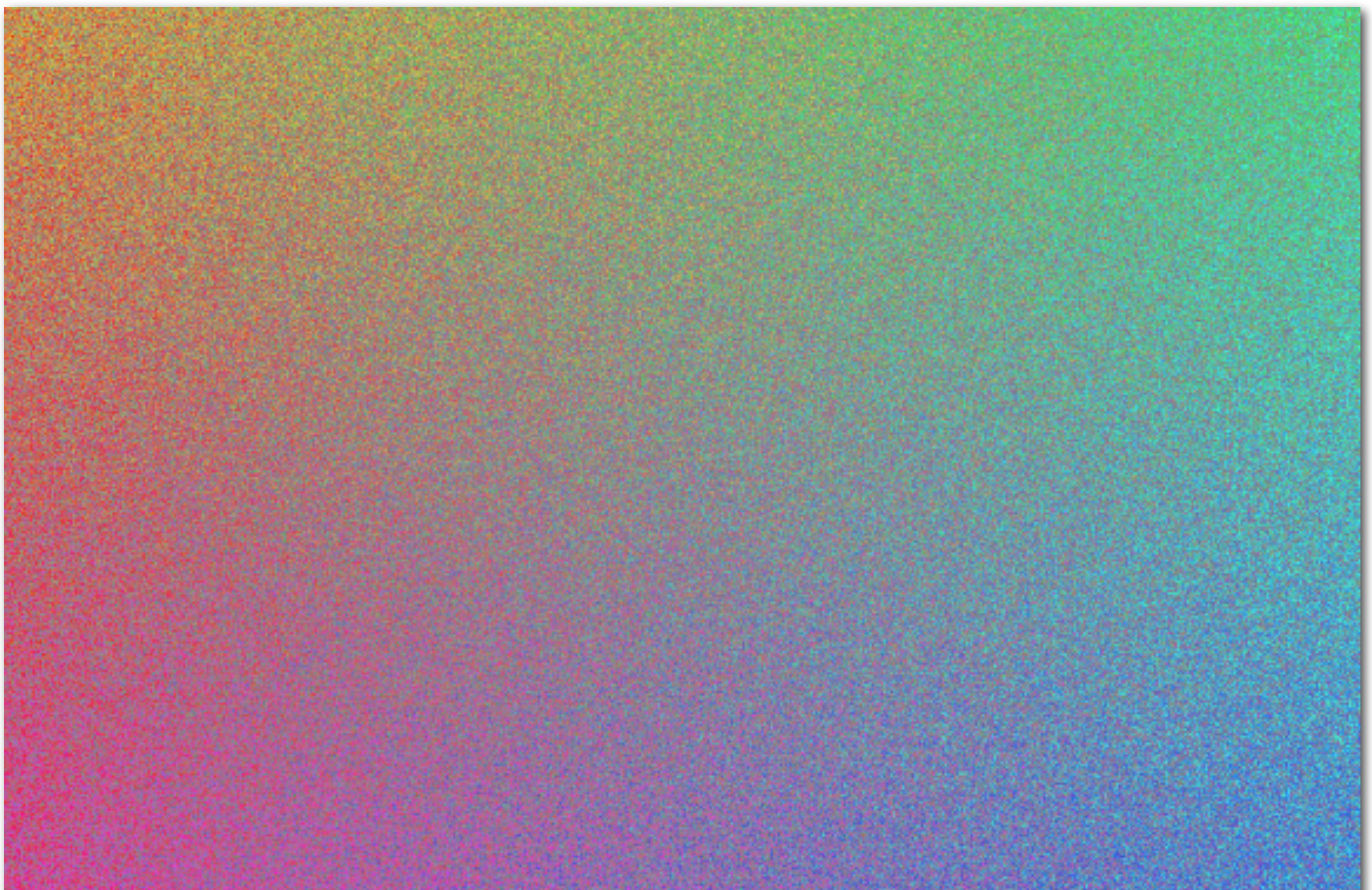
[Barnes et al. 2009]

- A <u>randomized</u> algorithm for rapidly finding correspondences between image patches

- Problem definition:
  - Given images A and B, for each patch in image A, compute the offset to the nearest neighbor patch in image B
    - Overlapping patches: each patch defined by its center pixel (ignoring boundary conditions, each MxN image consists of MxN patches)
  - PatchMatch computes nearest neighbor field (NNF)
    - NNF is function $f\colon \mathbb{R}^2 \rightarrow \mathbb{R}^2$ (maps patches in A to patches in B)
    - Example: if patch b=(x2,y2) in image B is NN of patch a=(x1,y1) in image A, then $f(a) = b$

PatchMatch: key idea #1

- **Law of large numbers: a non-trivial fraction of a large field of random offset assignments are likely to be good guesses**
- **Initialize *f* with random values**



Visualization of nearest neighbor offsets:

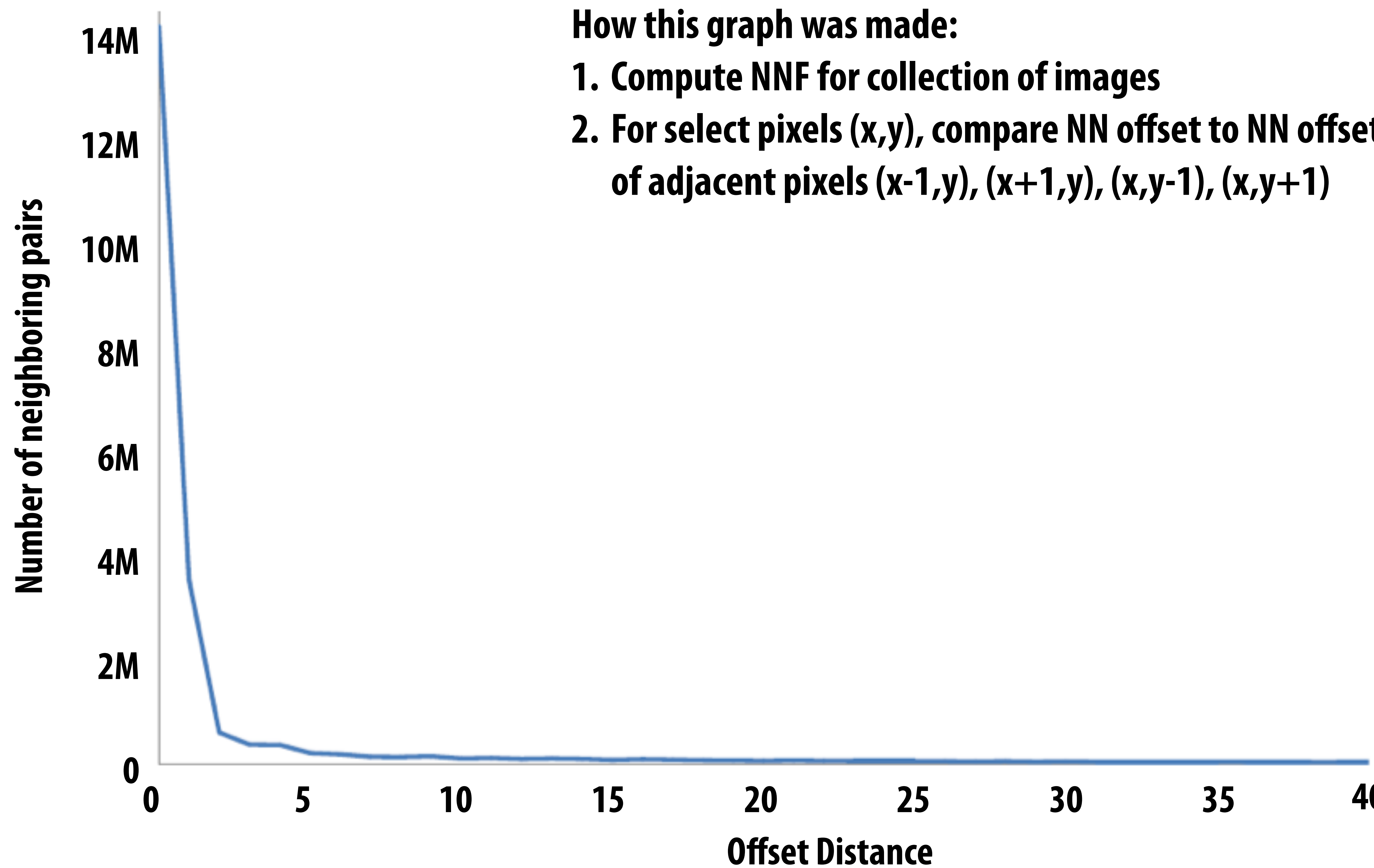
Saturation = magnitude of match offset
Gray = zero offset: best match patch in B is at same pixel location as query patch in A)

Hue = direction of offset
offset X = red-cyan axis
offset Y = blue-yellow axis

Image credit: [Barnes et al. 2009]

PatchMatch key idea #2: spatial coherence

- There will be high coherence of nearest neighbors in natural images
- Nearest neighbor of patch at (x,y) should be a strong hint for where to find nearest neighbor of patch at (x+1,y)



How this graph was made:

1. Compute NNF for collection of images
2. For select pixels (x,y), compare NN offset to NN offsets of adjacent pixels (x-1,y), (x+1,y), (x,y-1), (x,y+1)

Image credit: [Barnes et al. 2009]

Propagation: improving the NNF estimate

- **The NNF estimate provides a "best-so-far" NN for each patch in A**
  - $f(a)$ = nearest neighbor patch of a
  - $d(a,b)$ = distance between patch a and patch b (e.g., sum-of-squared differences over the patch)

- **Try to improve NNF estimate by exploiting spatial coherence with left and top neighbor:**
  - Let $a=(x,y)$, then candidate matches for a are:
    - $f(x\text{-}1, y) + (1,0)$
    - $f(x, y\text{-}1) + (0,1)$
  - Replace $f(a)$ with candidate patch $b=f(x,y\text{-}1)+(0,1)$ if $d(a, b) < d(a, f(a))$

- **Next iteration, use bottom and right neighbors as candidates**
  - Propagate down-right in first pass
  - Propagate up-left in second pass, etc.

PatchMatch iterative improvement

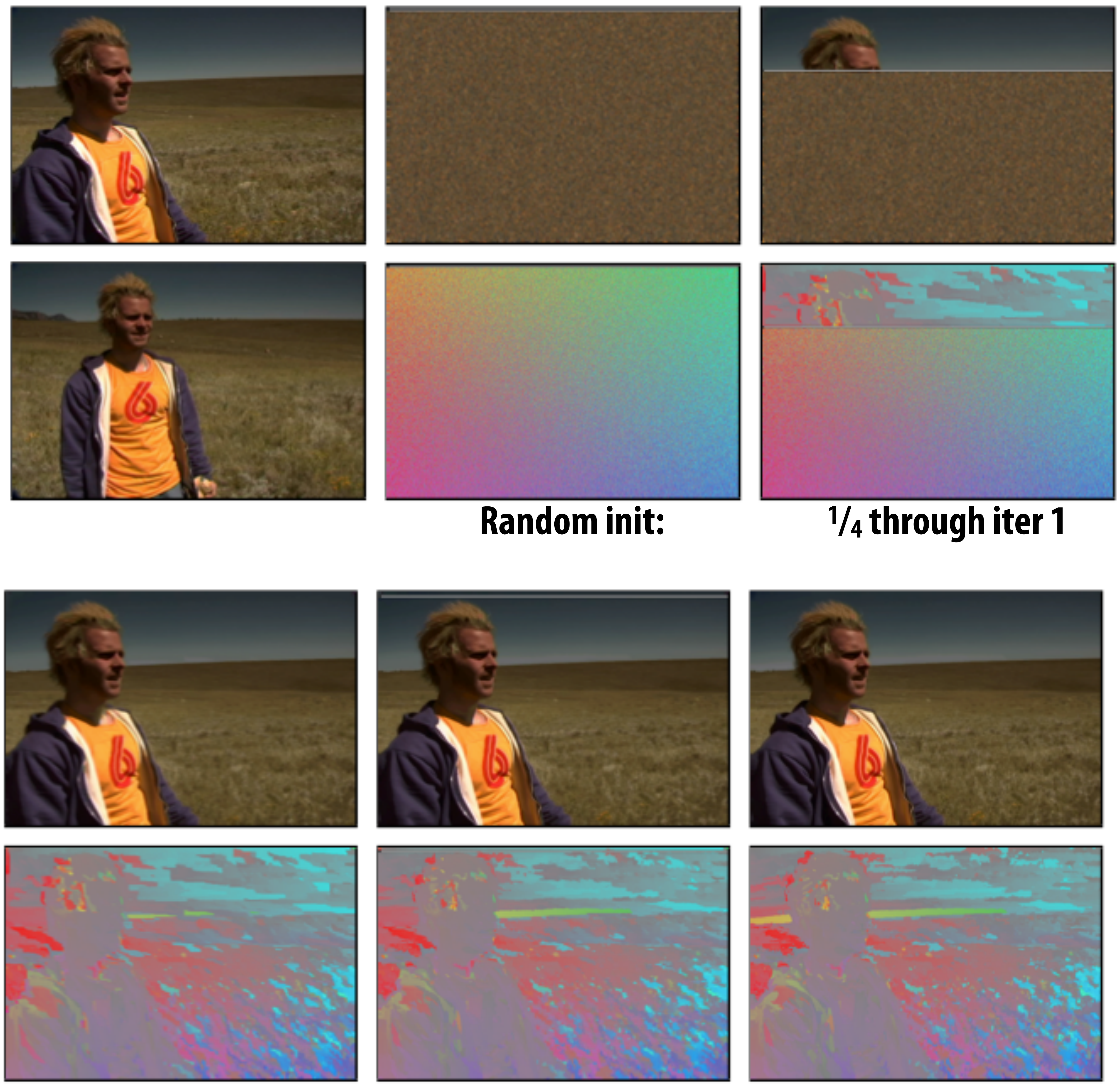


Image A

Image B (source of patches)

Random init:

$^1/_4$ through iter 1

End of iter 1

Iter 2

Iter 5

Experiment:
Reconstruct image A using patches from image B

Image credit: [Barnes et al. 2009]

Random search: avoiding local minima

- **Propagation can cause PatchMatch to get stuck in local minima**

- **Sample random sequence of candidates from exponential distribution**
  - **Let *a*=(*x*,*y*), then candidate matches for *a* are:** $(x,y) + w\alpha^i \mathrm{R}^i$
  - R^i **is uniform random offset in [-1,1] x [-1,1]**
  - w **is maximum search radius (e.g., width of entire image)**
  - α **is typically 1/2**
  - **Check all candidates where** $w\alpha^i \geq 1$

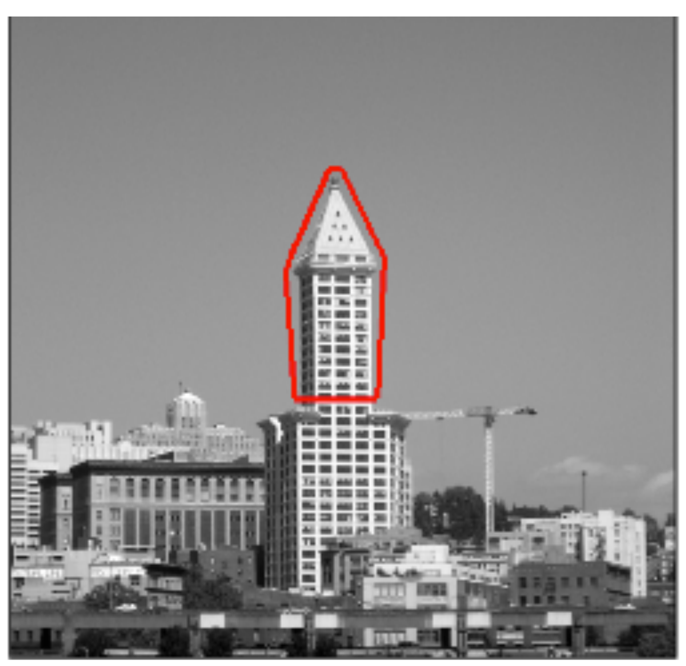
Optimization: enrichment

- Propagation step propagates good matches across spatial dimensions of image
- Can also propagate good matches across space of matches itself
- Idea: if $f(a) = b$, and $f(b) = c$, then c is a good candidate match for a
  - If you think of the NNF as a graph, then enrichment looks for nodes reachable in two steps
  - Note: enrichment assumes we're searching for matches in the same image as the image we are trying to complete

Example applications

Photoshop's Content Aware Fill

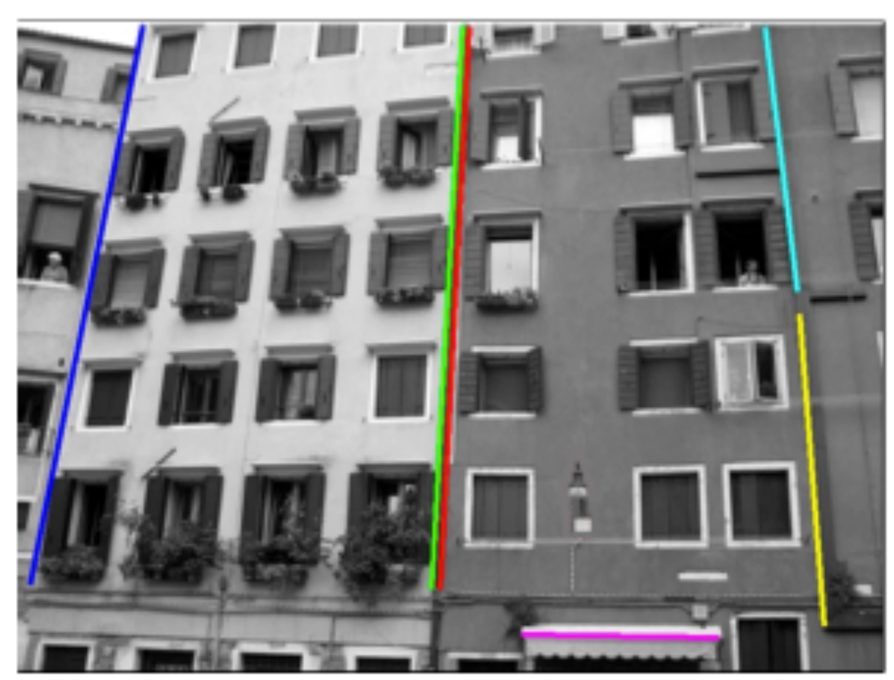
Object Manipulation



Building segment marked by user

Building scaled up, preserving texture

Image retargeting (changing aspect ratio)



Original image (with user-provided search constraints)

Retargeted (without constraints)

Retargeted (with constraints)

Image credits: [Barnes et al. 2009]

PatchMatch summary

- Randomized algorithm: converges rapidly in practice
- Main idea: coherence (largely spatial) of nearest neighbors
- Propagation step is inherently serial, but good parallel approximations exist
  - PatchMatch has been implemented efficiently on GPUs
- Data access caches well, but it is unpredictable (not a bounded window)
  - Different workload characteristics from many other image processing algorithms we have discussed

Class discussion

- Imagine the your final project is to architect a processor to handle image processing tasks for the widely anticipated kPhone. (like the iPhone, but better)

- How would you characterize image processing workloads?
  - Parallelism?
  - Data-access patterns?
  - Predictability? (of data access, of instruction stream)

- What are good characteristics of a processor for image processing tasks?
  - Programmable, or fixed-function?
    - If programmable, do we need: branch-prediction? out-of-order execution?
    - If fixed-function, in what ways can it be configured?
  - What forms of parallelism? (SIMD, multi-core)
  - Support for multi-threading, prefetching?
  - Data caches or on-chip buffers/scratchpads?

Readings

- **Adams et al. *The Frankencamera: An Experimental Platform for Computational Photography*. SIGGRAPH 2010**