# What Images are Like

D.A. Forsyth, University of Illinois at Urbana Champaign

Clean up around dictionary, and shorten

# Important facts

- Pixels are like their neighbors, so
  - you can denoise
  - Large changes are interesting, and you can find them
  - You can summarize images with interest points
    - which have very strong matching properties
- Image patches are repetitious
  - so you can denoise, inpaint, classify, regress
- Knowing whether something is an image is powerful and hard

# A crucial property of images

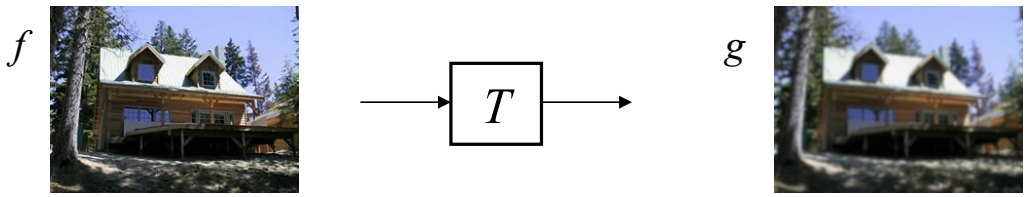
- Pixels are like their neighbors
  (mostly, for most pixels)

- Imagine you wish to denoise an image. You could do so by averaging neighbors (a filter!). Weighting the neighbors so nearby neighbors get heavier weights is a good move.

# Image filtering

- Roughly speaking, replace image value at $x$ with some function of values in its spatial neighborhood $N(x)$:

$$g(x) = T(f(N(x)))$$




- Examples: smoothing, sharpening, edge detection, etc.

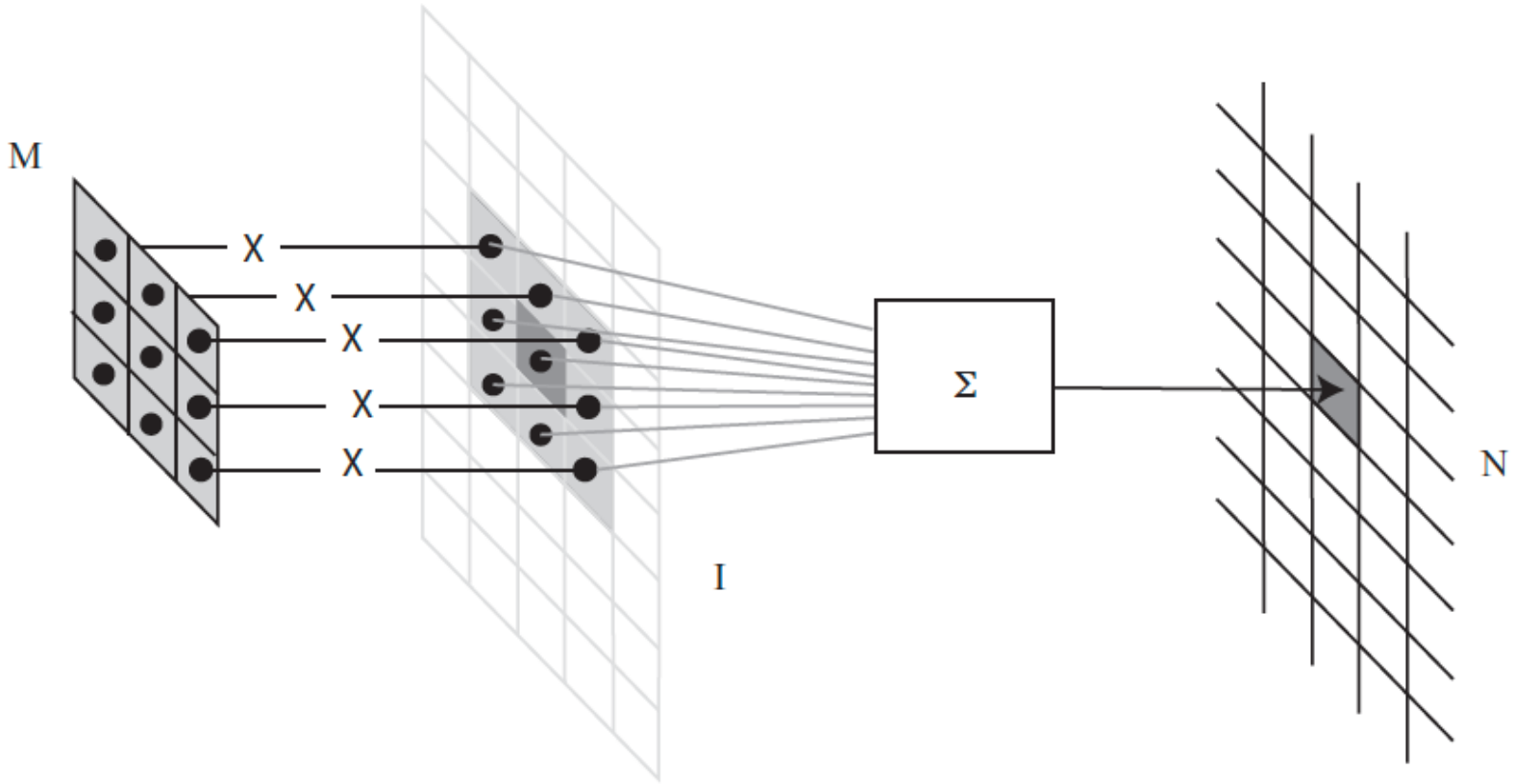
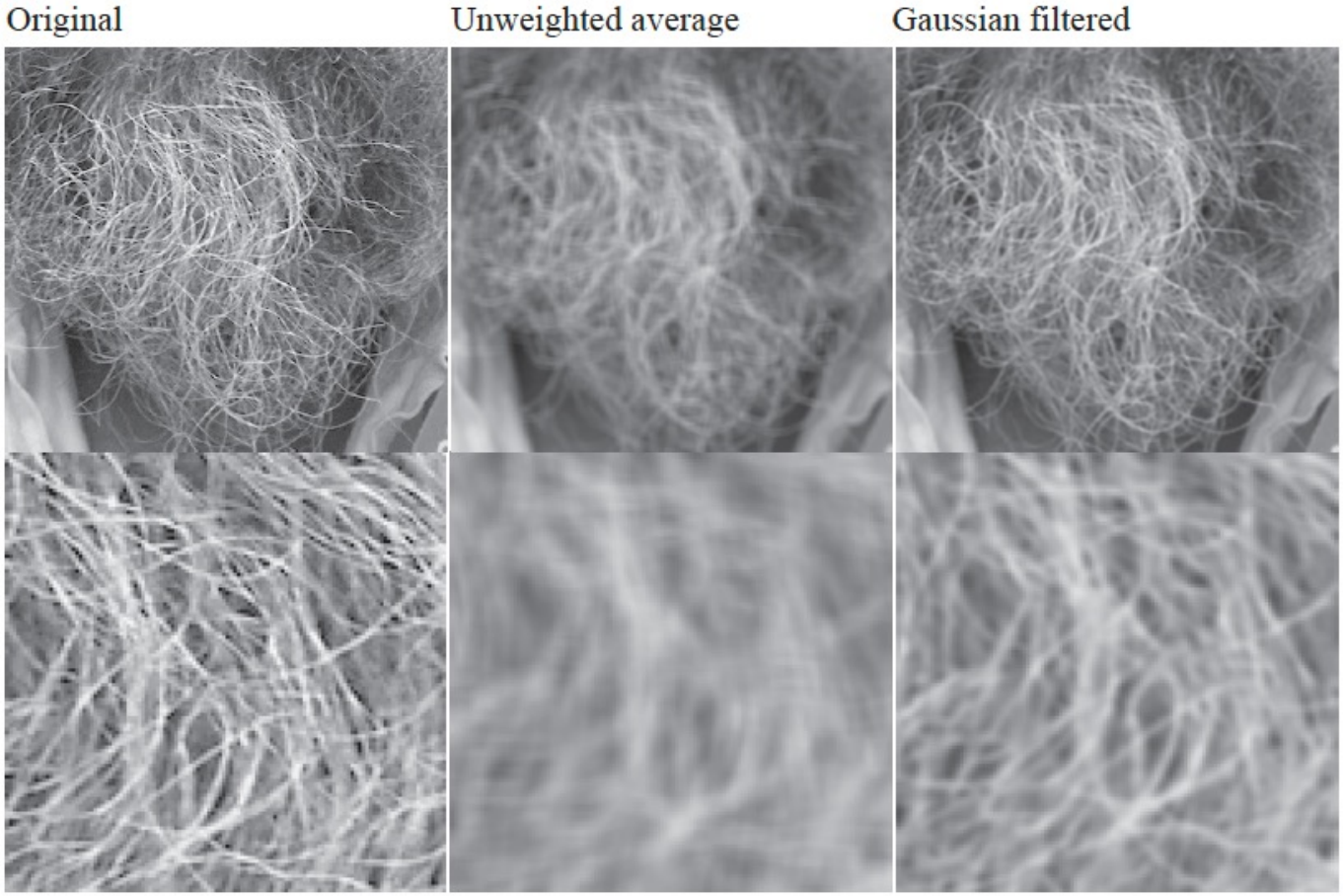



FIGURE 3.1: *To compute the value of $\mathcal{N}$ at some location, you shift a copy of $\mathcal{M}$ (the flipped version of $\mathcal{W}$) to lie over that location in $\mathcal{I}$; you multiply together the non-zero elements of $\mathcal{M}$ and $\mathcal{I}$ that lie on top of one another; and you sum the results.*

# Smoothing



# Filtering is a (very rough) pattern detector

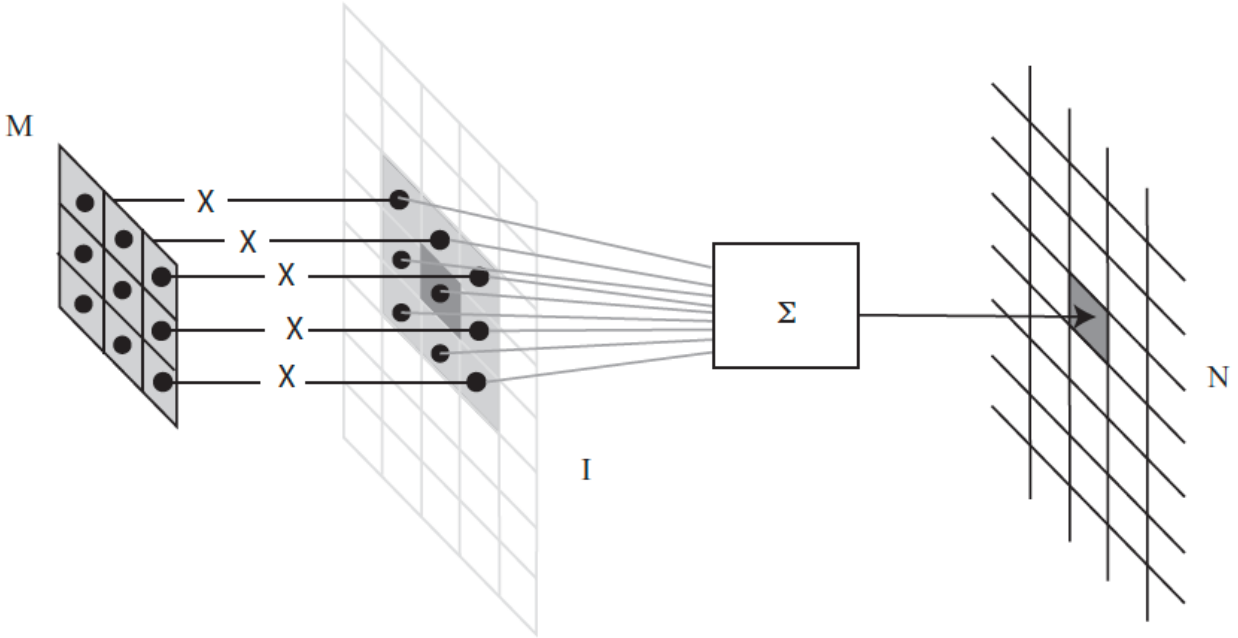



FIGURE 3.1: *To compute the value of $\mathcal{N}$ at some location, you shift a copy of $\mathcal{M}$ (the flipped version of $\mathcal{W}$) to lie over that location in $\mathcal{I}$; you multiply together the non-zero elements of $\mathcal{M}$ and $\mathcal{I}$ that lie on top of one another; and you sum the results.*

# Gradients are patterns

For an image $\mathcal{I}$, the gradient is

$$\nabla \mathcal{I} = (\frac{\partial \mathcal{I}}{\partial x}, \frac{\partial \mathcal{I}}{\partial y})^T,$$

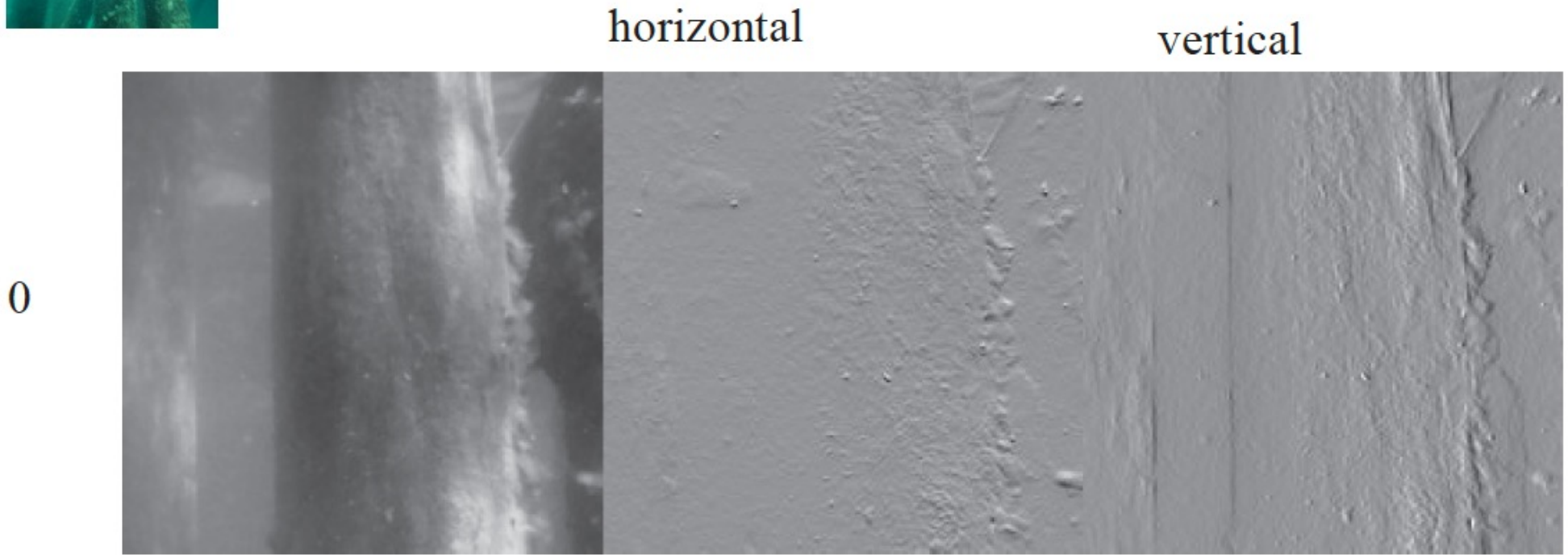
which we could estimate by observing that

$$\frac{\partial \mathcal{I}}{\partial x} = \lim_{\delta x \to 0} \frac{\mathcal{I}(x + \delta x, y) - \mathcal{I}(x, y)}{\delta x} \approx \mathcal{I}_{i+1,j} - \mathcal{I}_{i,j}.$$

This means a convolution with

$$-1 \quad 1$$

Spot the typo!

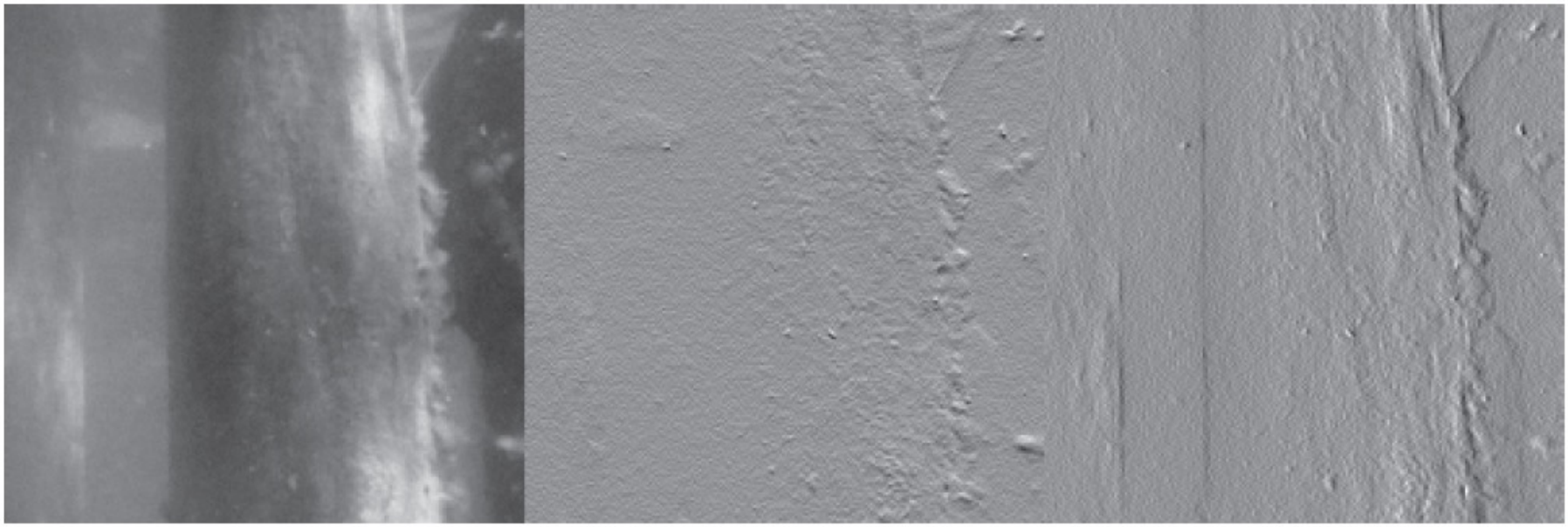


Spot the typo!

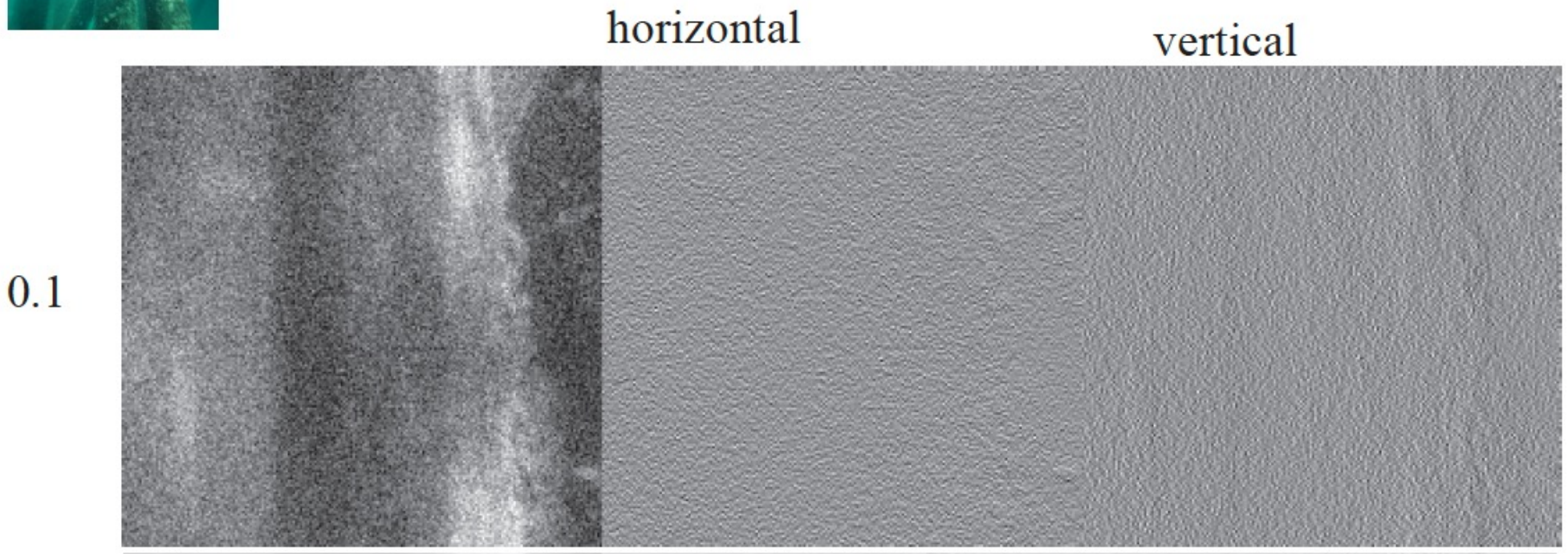
horizontal

vertical

0.01



Spot the typo!



# Pixels are like their neighbors

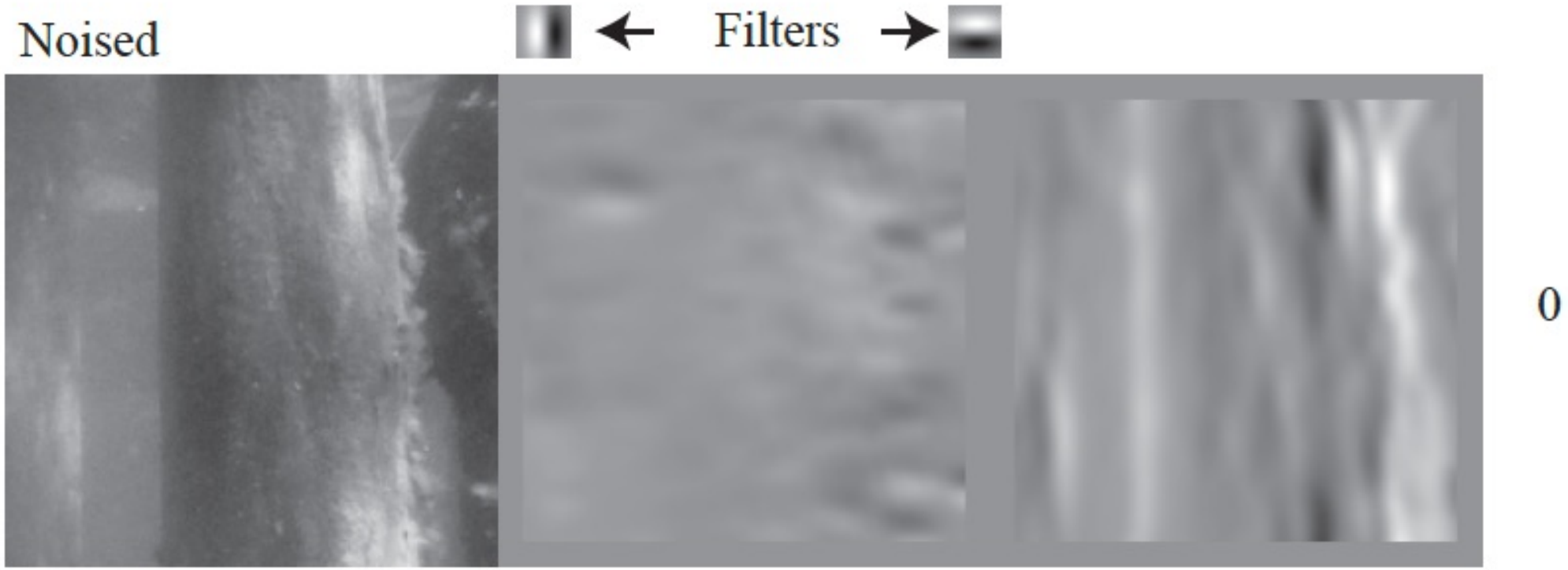
- so you can denoise images
- and gradients are important

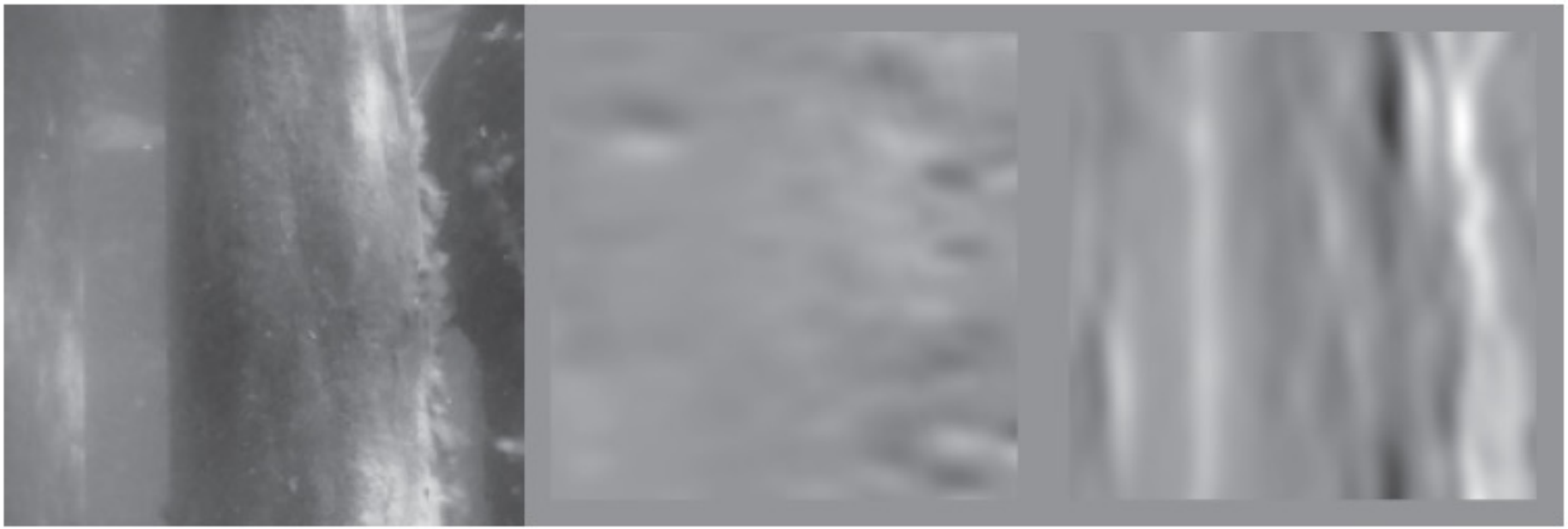
# You can find gradients reliably

- and edges
- and other good stuff

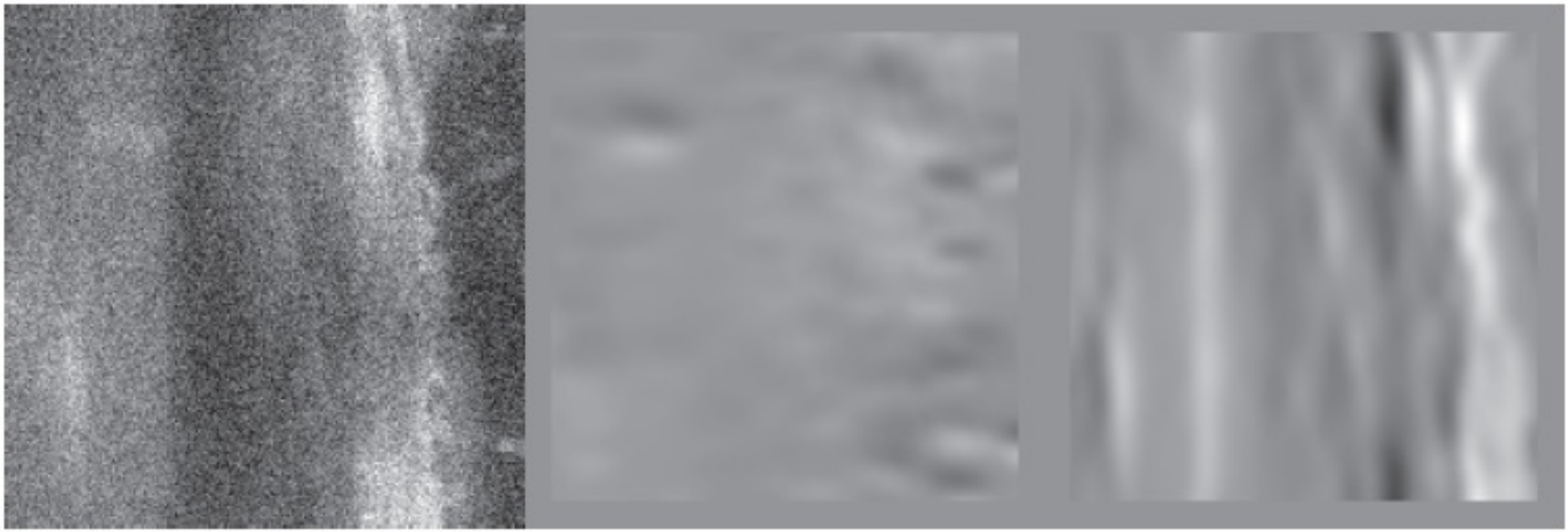
- Denoise with a smoothing filter
  - then find the gradient

SPOT THE TYPO!





0.01



0.1

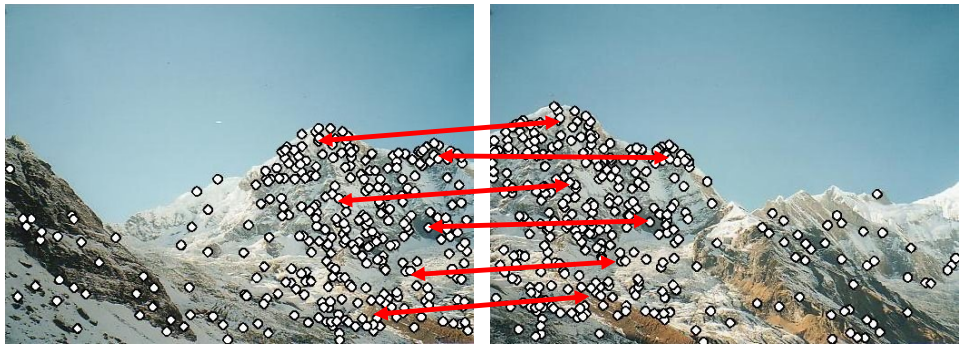
# You can build well behaved point descriptors

- Pixels are like their neighbors => many pixels are redundant
  - Meaning you could “leave them out”

- Some pixels are much more important than others
  - Keypoints, interest points

# Why extract keypoints?

- Motivation: image alignment
  - We have two images – how do we combine them?



Step 1: extract keypoints

Step 2: match keypoint features

# Keypoint representations support very fast methods

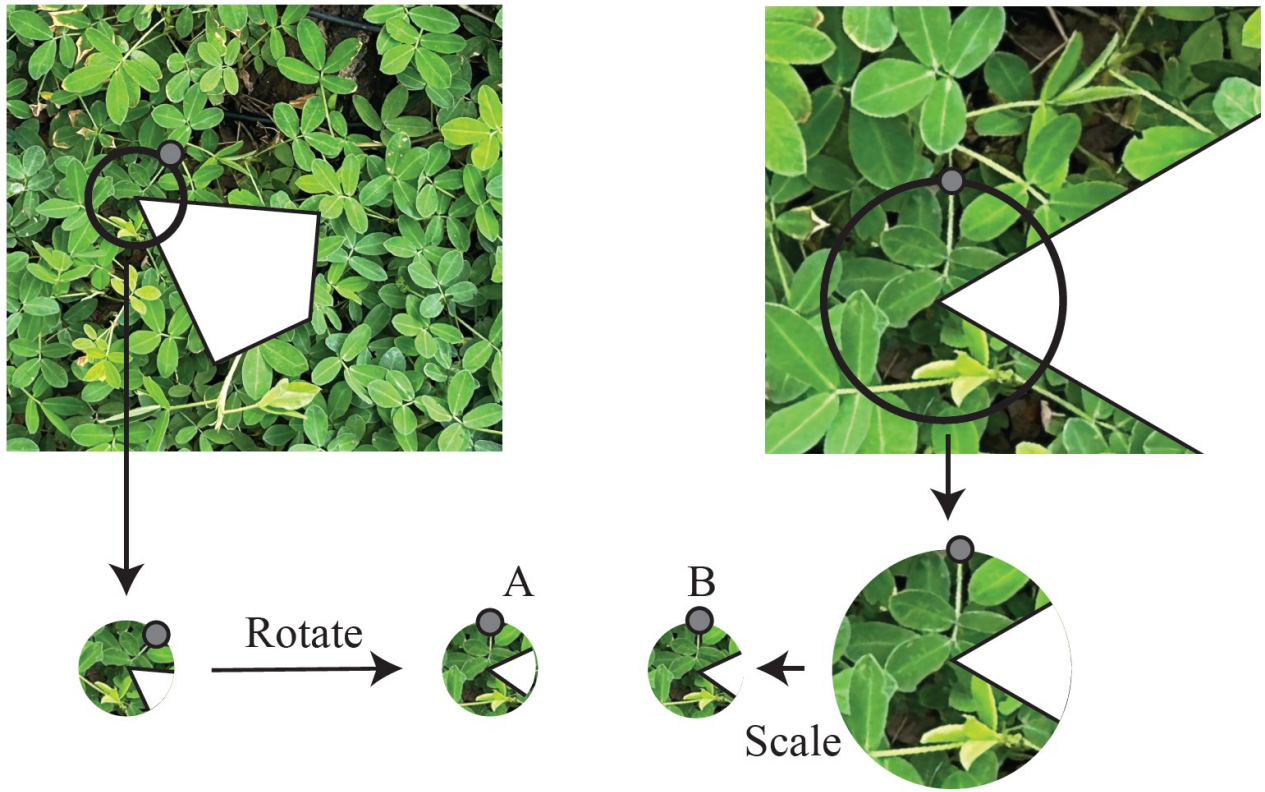
- In some applications, GPU just isn’t available
  - too heavy; too much power; etc
  - And there isn’t much CPU either

- Idea: reduce image to keypoints, work with those

# Need to find

- Where is it:
- What scale its at:
- What orientation its at:

In a way that is (mostly) unaffected by image transformations


Rotate
A
B
Scale

# Need to find

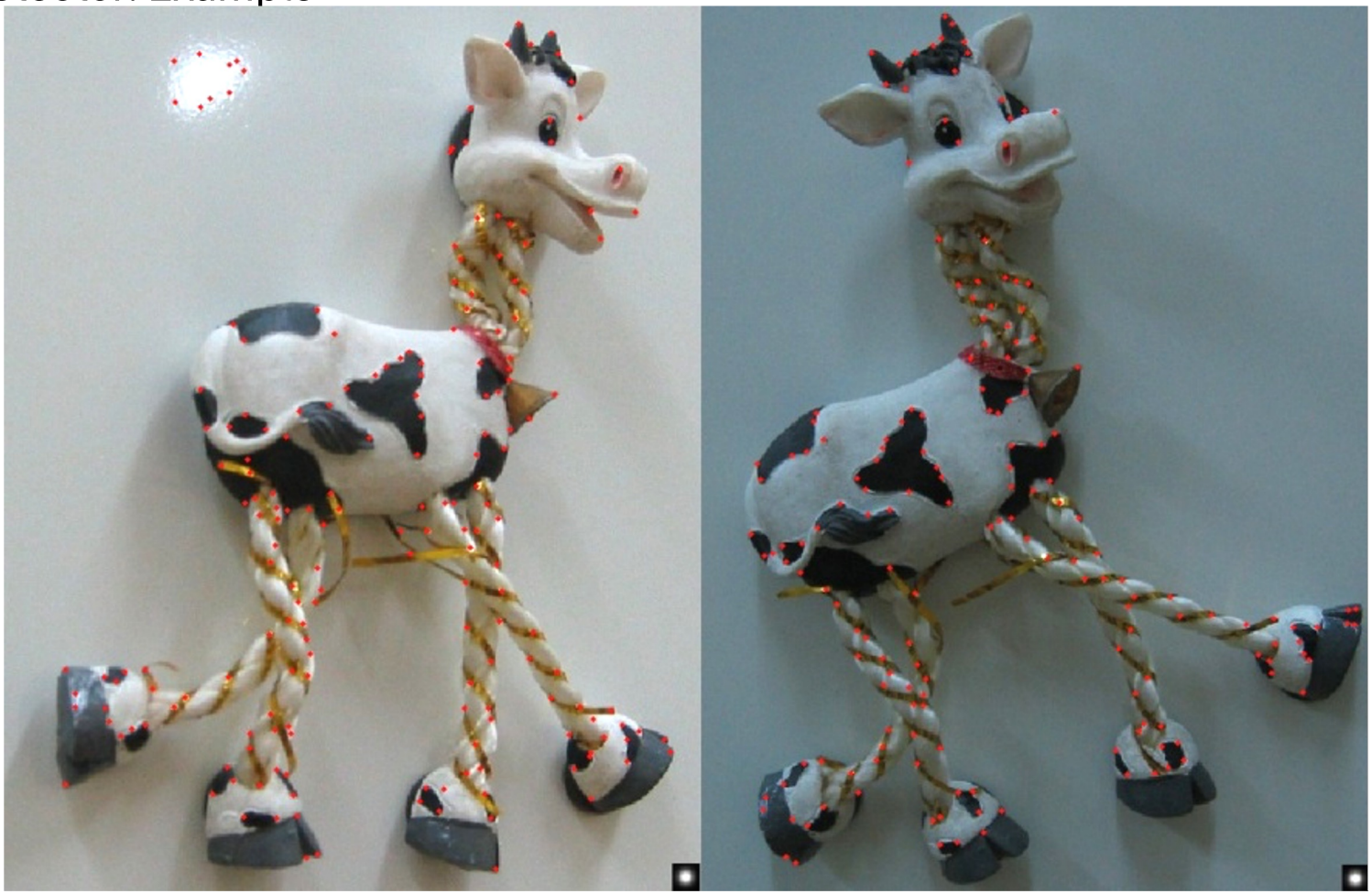
- Where is it:
- What scale its at:
- What orientation its at:
- Description of contents:

Corner detector

In a way that is (mostly) unaffected by image transformations

Harris Detector: Example

Harris Detector: Example



# Need to find

- Where is it:
- What scale its at:
- What orientation its at:
- Description of contents:

LoG filter peak

In a way that is (mostly) unaffected by image transformations

# Need to find

- Where is it:
- What scale its at:
- What orientation its at:
- Description of contents:

Direction of most common gradient in window

In a way that is (mostly) unaffected by image transformations

# Need to find

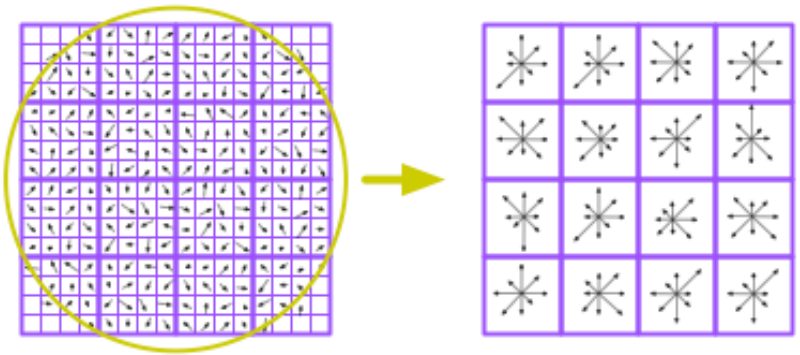
- Where is it:
- What scale its at:
- What orientation its at:
- Description of contents:

Sift features
(or some equivalent)

In a way that is (mostly) unaffected by image transformations

# Describing a window's contents – SIFT

- We want description to be:
  - Invariant to changes in image brightness: - use orientations
  - Robust to noise: - ignore orientations with small magnitude
  - Distinctive: - use lots of local orientations
  - Invariant to small errors in location:
    - "bucket" the orientations in image
    - Use a histogram for each bucket
- SIFT features behave very well
  - the nearest neighbor to a query patch
    - is usually a matching patch

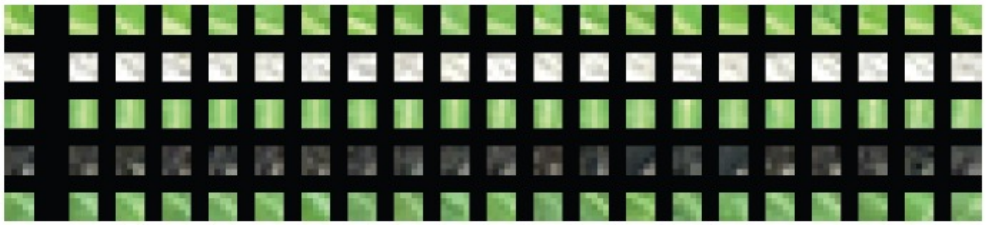


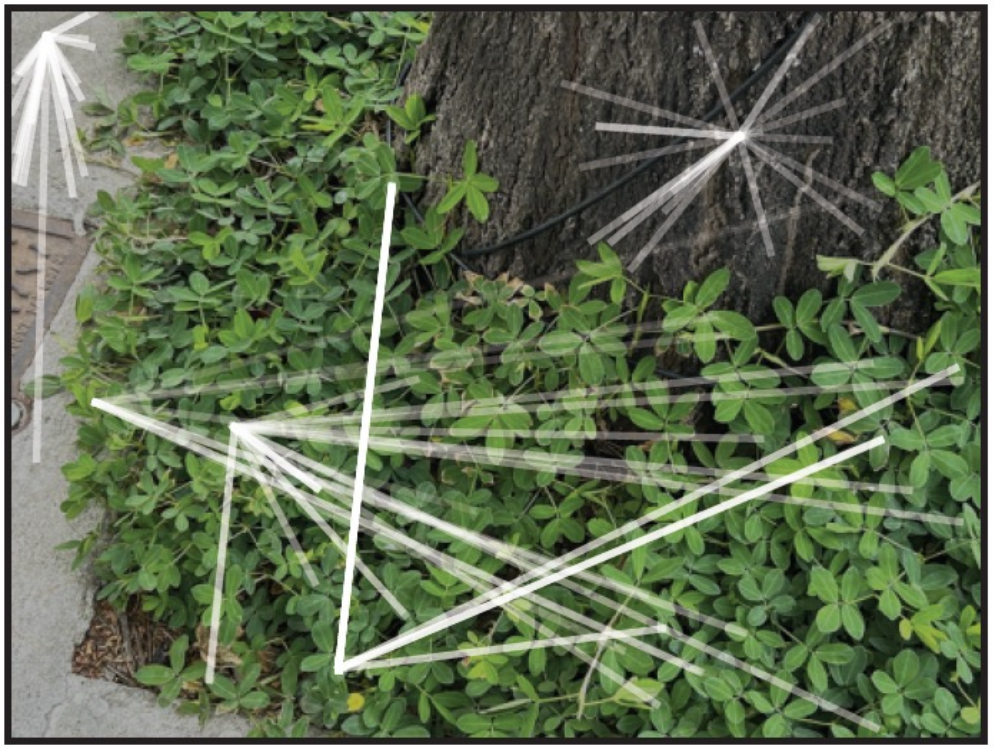
# Images are made of quite repetitious patches

- Because pixels are like their neighbors
- So there is often a patch some distance away that looks like the patch you're looking at
  - even for quite big patches
- This is hugely useful
  - Denoising
  - Inpainting
  - Classification
  - Regression

# Patch matches

5 x 5



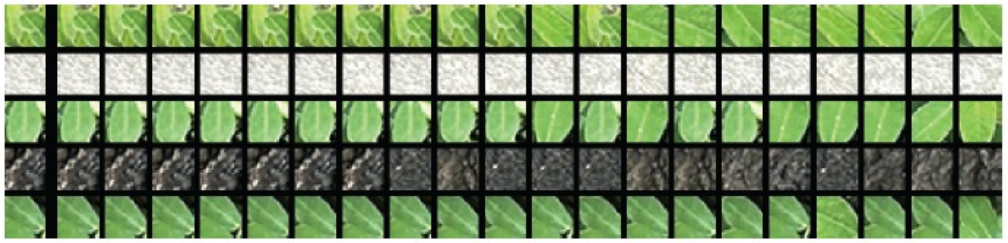


# Patch matches

11 x 11

# Patch matches

21 x 21



# Inpainting

- Simplest case:
  - some fraction of pixels has been “knocked out” at random
  - (perhaps set to zero)
  - and you know which pixels

- Fix:
  - take window around knocked out pixel
  - find closest match in the image
  - take the center pixel from matching window

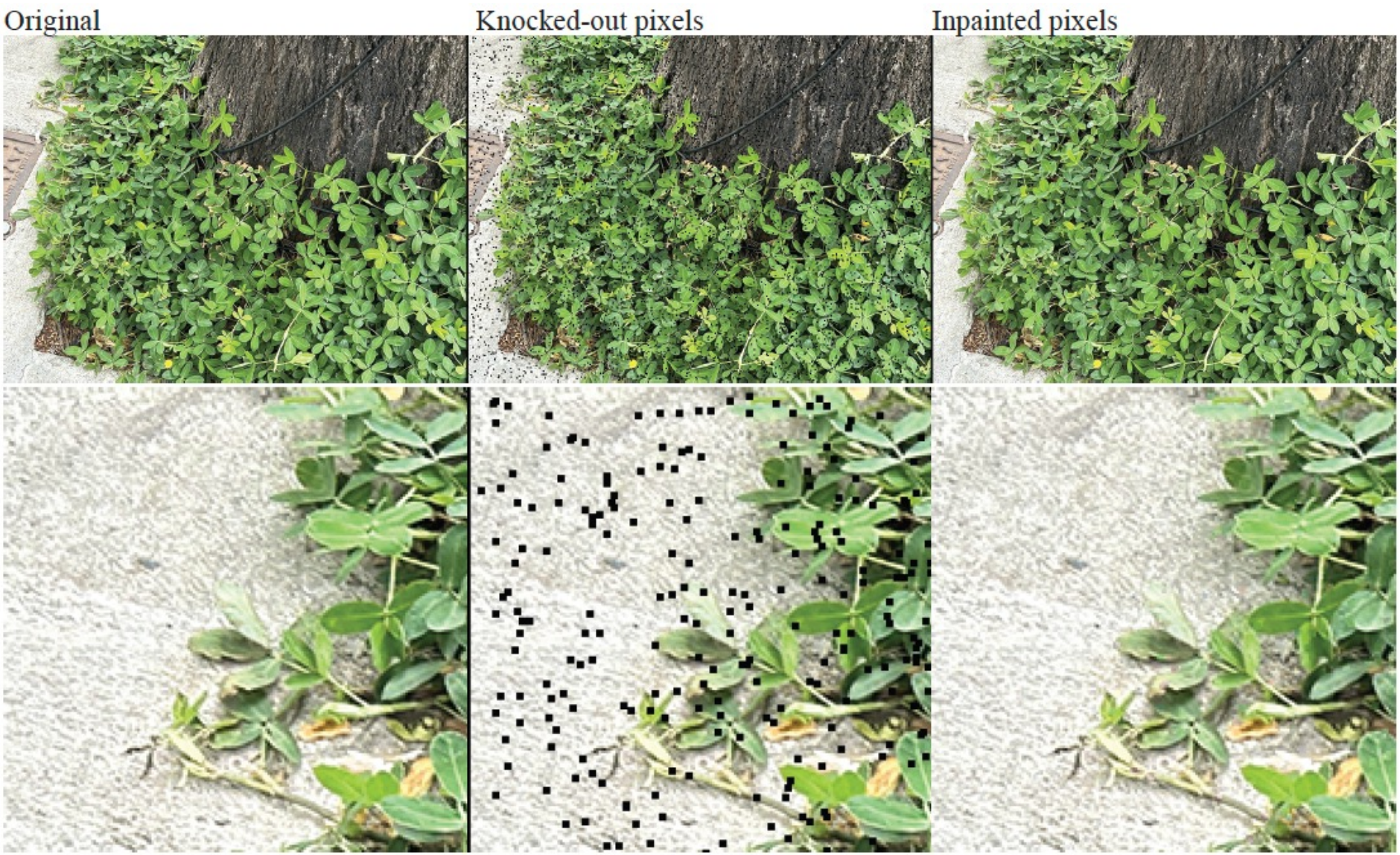
Original

Knocked-out pixels

Inpainted pixels

# Inpainting for bigger knockouts

- Assume a kxk window gets ”knocked out” (k small, but k>1)
- Procedure works if you are careful about matching
- - eg don’t match to windows that have knocked out pixels


Original
Knocked-out pixels
Inpainted pixels

# Incremental inpainting

Now imagine that the process that knocks out pixels doesn't just choose pixels at random, but has some some kind of spatial structure. For example, you might have an image with writing on it, and want to replace the writing. Alternatively, the image might have one more more large holes in it.

The pixel inpainting procedure above will work, but some details need to change. When isolated pixels are knocked out, you expect that the patch around the pixel is known. If the image has a large hole in it, this no longer applies. Fixing a pixel requires you have at least some known pixel values close to it. Choose such a pixel, and match the patch using the known pixels only. You can do this with a mask that zeros the contribution of knocked out pixels to the SSD. This produces a pool of matches. Now estimate the value of the pixel using this pool. For the moment, choose the center of the best match. Place this value in the image, and you now have an image with a slightly smaller hole in it, so you should be able to find more candidate pixels for replacing. In this *incremental reconstruction* approach, the order in which you visit pixels and the size of the patch becomes important and can quite strongly affect the result.

# Forming and using patch dictionaries

- Idea:
  - other patches in an image are good for denoising
  - why not use other patches in other images?

- Issues:
  - how to find the right patch in a very large number of patches?
  - redundancy
  - many patches are like one another, and so just make work

# Desirable outcome

- Procedure to
  - accept a patch
  - produce a patch/some patches that are similar
- Built out of very large number of patches

# A very simple image classifier

- Problem:
- you want to classify an image

- Strategy:
- learning:
  - get many labelled images
  - cut into patches
  - build tree
  - for each leaf, record the most common label
- classifying:
  - cut image into patches
  - pass patch down tree
  - vote for label

Now old-fashioned, but moderately effective and very good in its time

# Denoising from weird noise

- Imagine you have very strange noise, and you can simulate it
- You want to denoise

- Strategy:
  - Take many images, and create (noisy – clean) pairs
  - Carve into patches and build a HKM tree using only the noisy patches
    - (but keep the clean patch – so each data item is (noisy – clean) but the clusters, etc are formed using noisy alone)
- To denoise:
  - carve up noisy image into patches
  - walk the tree with the noisy patches
  - at leaf, substitute the clean version of the best match