Kevin Chen

12/19/18

Intro Computer Vision

Main Instructor: Dr. Aaron Bobick. Side Instructor: Dr. Iran Essa

\*Note\*: All comments in the code is Java-style with //, not MATLAB style with %.

**Introduction**

What This Course Covers

* This course covers the foundational aspects of how to analyze images and to extract content from images, i.e. how to see and interpret an image.
  + Foundational: mathematical and computational methods for interpreting images
  + Interpreting images: extract knowledge from images
* Examples of images: scene of the roads (autonomous vehicles), image of brain
  + Locate objects in an image and track them as they move

Difference between CV and CP (Computational Photography)

* CP is image processing, which is about capturing a light from a scene to record a scene into a photograph or related artifact that showcases the scene in novel ways
  + Image processing is needed to display the scene in novel ways
* CV is about interpreting an analysis of the scene: content of image (who/what in image and what is happening)

What is Computer Vision

* Steve Seitz: Goal of computer vision is to write computer programs that can interpret images.
  + Input: images. Output: some kind of meaning/interpretation/understanding of those images
  + Image processing is different b/c input is image and output is a modified image
* To recognize actions, computer vision may require a video instead of a single image.

Why Study Computer Vision

* Images (and movies) have become ubiquitous in both production and consumption
* Therefore, applications to manipulate images/movies are becoming core.
* Also applies for systems that extract information from imagery: surveillance, building 3D representations, motion capture assisted

OCR and Face Recognition (State of the Art CV)

* Optical character recognition (OCR): convert scanned docs to text
  + Considered difficult not too long ago, but now pretty standard.
  + Even works sometime for handwritten characters (e.g. address on envelope)
* Face detection
  + Blink detection.
  + Sony “smile shutter”: waits till it sees the person smile
  + Camera based login/authentication

Object Recognition

* Evolution Robotics Retail developed LaneHawk, a retail loss-prevention solution that helps turn bottom-of-basket (BOB) losses and in-cart losses into profits in real time.
  + In other words, when checking out in a store, it’ll detect if you tried keeping something in your cart (including on the bottom rack)
* Augmented reality: object recognition by mobile devices

Special Effects and 3D Modeling

* Special Effects: shape capture
  + Use image recognition to build a 3D model of a person in an image.
* Special Effects: motion capture
  + Knowing specific location of markers for placing objects on.
* 3D Modeling: Earth viewers (Microsoft’s Virtual Earth and Google Earth)
  + Use aerial imagery of Earth, figure out models of the buildings, reconstruct a map with only the models, and then fly around them virtually.

Smart Cars

* Mobileye for helping with driving safety
* Autonomous cars

Sports

* CV used in sports TV broadcasting
* Sportvision adds the down line in football.
  + If person is on the down line, the line goes behind the person. Doesn’t work so well on rainy/muddy days

Vision Based Interaction

* Computer vision in video games – Nintendo Wii controller
* Microsoft Kinect could product skeletal descriptions of people. (Rough bare-bone skeleton models from an image)
  + Can detect actions like driving or waving

Security and Surveillance

* Crowd monitoring, port monitoring
* Medical imaging: 3D imaging in MRI and CT, image guided surgery

Why is Computer Vision Hard?

* Pretty easy for humans generally, but there are optical illusions even for us, making computer vision hard.

Vision is NOT Image Processing

* Seeing is not the same as measuring properties in the image. Rather, seeing is building a percept (i.e. a description) of what is in the world based upon the measurements made by an imaging sensor.
* We, for example, build models from change in an object.
* Vision can lead to illusions, but image processing obviously cannot since it involves objectively measuring image properties.
  + Vision is the result of your brain making a story/description of what is seen

Course Overview

* Computer vision is the relationship between three ways of thinking of what goes on
  + Computational models (math). Example: geometry in stereo vision
  + Algorithms: e.g. stereo algorithms
  + Real images: comparing your result to the ground truth (right answer)

Topic Outline

* 10 overall units
  + Introduction
  + Image processing for computer vision
  + Camera models and views
  + Features and Matching
  + Lightness and Brightness
  + Image Motion
  + Motion and Tracking
  + Classification and Recognition
  + Miscellaneous operations
  + Human Vision

Software

* MATLAB with Image Processing Toolbox
  + img = imread(‘*image-name*.png’) creates a new image variable from a png image file.
  + imshow(img) will show the image, where img is an image variable.
  + img\_gray = rbg2gray(img) converts an image (passed in as img) to a gray scale image.
* Octave: basically open source version of MATLAB
  + A lot of code is compatible with MATLAB.
  + Install the image package, along with its dependencies (general, control, and signal) using the command: “pkg install -forge general control signal image;”
* Python + OpenCV

**Images as Functions**

Images as Functions Intro

* A (grayscale) image can be thought of a function: I(x, y).
  + Modifying the image is simply modifying this function.
  + Example: smoothing the function results in blurring the image.
* This function, f or I, maps from to R:
  + f(x, y) gives the intensity or value at position (x, y)
* Practically define the image over a rectangle, with a finite range:
  + f: [a, b] x [c, d] -> [min, max]. “min” is black, “max” is white
* Color images as functions: a color image is just three functions “stacked” together. We can write this as a “vector-valued” function:
* By convention, origin is top left of image. +x points right and +y points down.

Digital Images

* In CV, we typically operate on digital (discrete) images. 2 steps for discretization:
  + Sample the 2D space on a regular grid (at each point on the grid)
  + Quantize each sample (round to “nearest integer”)
* Image is thus represented as a matrix of integer values.

Matlab Images are Matrices

* Program that stores the green values of an image:
  + im = imread(‘peppers.png’);
  + imgreen = im(:, :, 2); //Include all rows (first colon), include all colunns (second colon), include just green values (parameter “2”). For the third parameter 1 = red, 2 = green, 3 = blue b/c indexing starts from 1.
  + imshow(imgreen) //shows or displays the image
  + plot(imgreen(256, :)); //plots the 256th row of green intensity values, with x-coordinate as index and y-coordinate as intensity value.
* Load and Display an Image:
  + img = imread(‘dolphin.png’);
  + imshow(img);
  + disp(size(img)) //displays the size of the image. Prints height and then width.
  + disp(class(img)) //displays the image class or data type (e.g. uint8)
* Inspect Image Values:
  + disp(img(50, 100)) //display image value at row 50, column 100.
  + disp(img(101:103, 201:203)) //displays all values from rows 101-103 and columns 201-203, inclusive.
  + disp(img(50, :)) //displays all values from row 50
  + plot(img(50, :)); //plots the values from row 50
* Crop an image:
  + cropped = img(110:310, 10:160); //selects the rectangle formed by rows 110-310 and columns 10-160, inclusive, leaving all other pixels cropped out.
* Size of a colored image would return 3 values: # of rows, # of cols, # of color planes/channels
* Arithmetic Operations on 2 images (basically matrix operations):
  + average = dolphin / 2 + bicycle / 2; //Sums the images “dolphin” and “bicycle”. We divide by 2 to get the same range of values.
  + NOTE: This is NOT the same as “average = (dolphin + bicycle) / 2)” because adding the values first could cross the maximum threshold (e.g. 255 for uint8). If the value from dolphin is 183 and the value from bicycle is 152, the sum will be 255 for uint8.
  + The result of this operation is the blending of the two images.
* Multiply by a Scalar
  + halved = 0.5 \* dolphin // divides each intensity value by 2, rounded down.
* MATLAB function syntax:

function *return\_value* = *function\_name*(*param1*, *param2*, … , *paramN*)

//Body: Line 1

//Body: Line 2

…

endfunction

* Function that multiplies a matrix by a scalar:

function result = scale(img, value)

//The “.\*” makes sure we are doing a bitwise element operation (multiplication).

result = value .\* img;

endfunction

* + We can then call the function: imshow(scale(dolphin, 1.5));
* Alpha blending function:

function output = blend(a, b, alpha)

output = a .\* alpha + b .\* (1 - alpha);

endfunction

Common Types of Noise

* Noise is just another function that is combined with the original function to get a new function:
* Common Types of Noise:
  + Salt and pepper noise: random occurrences of black and white pixels
  + Impulse noise: random occurrences of white pixels
  + Gaussian noise: variations in intensity drawn from a Gaussian normal distribution
* Gaussian Noise in MATLAB:

noise = randn(size(im)) .\* sigma; //randn returns a number sampled from the normal distribution with mean 0 and standard deviation 1.

output = im + noise;

* + The lower sigma is, the lower the noise you add to your photo.
* Image Difference
  + The difference of two images shows how different each pixel is between two images. The whiter the difference, the larger the difference.
  + We often care only about the magnitude of the difference. The abs(*value*) function returns the magnitude.
  + We would like to do: abs\_diff = abs(bicycle – dolphin). However, bicycle – dolphin would be truncated to zero if the result is negative, and hence abs does nothing.
  + Two possible solutions (given images “a” and “b”):

1. Perform (a – b) + (b – a)
2. Convert them to floating point images
3. Use image package:

pkg load image;

abs\_diff = imabsdiff(a, b);

imshow(abs\_diff)

* Generating Gaussian Noise:
  + Return a matrix with 2 rows and 3 columns with values randomly sampled from a Gaussian Distribution with mean 0 and standard deviation 1.

noise = randn([2 3]);

* + MATLAB Histogram function example:

noise = randn([1, 100]);

[n, x] = hist(noise, [-3 -2 -1 0 1 2 3]); //Creates a histogram with the noise vector and bins with centers at -3, -2, -1, 0, 1, 2, 3. This returns the count of elements and the bin centers.

disp([x; n]);

plot(x, n);

* + hist(noise, [-3 -2 -1 0 1 2 3]); can be replaced with hist(noise, linspace(-3, 3, 7)). It starts from -3, goes to 3, with even spaces to get a total of 7 bins.
  + rand() samples numbers from a uniform distribution. randi() generates random integers.
* The amount of noise (sigma) must be relative to the overall range of intensity values in an image.
* Displaying images in Matlab with intensity value “*LOW*” (e.g. 0) as black and “*HIGH*” (e.g. 255) as white:

imshow(image, [*LOW HIGH*])

* + Passing an empty array scales the intensity values automatically. (Sets minimum value in the array to be black and maximum value to be white)
  + You only normalize the image to display it, not to compute with it.

**Filtering**

Smoothing an image with noise: Correlation Filtering

* First attempt: replace each pixel with an average of all the values in its neighborhood (e.g. all pixels within 3 pixels distance from the current pixel) – a moving average.
  + Key assumptions that this makes:

1. The “true” value of pixels are similar to the true value of pixels nearby.
2. The noise added to each pixel is done independently (e.g. the average of the noise would be zero for randn).

* If you know what the noise is, you generally can subtract it from the image. But the exception is if the noise causes a particular value to go beyond the threshold and get rounded to that threshold value.
* Weighted moving average: the weights of pixels further away but still within a neighborhood would be smaller.
  + A non-weighted average could use the weights [1, 1, 1, 1, 1] / 5 for the nearest 5 pixels (in a 1D case). A weighted average could use [1, 5, 6, 4, 1] / 16.
  + Notice that the above set of weights normalizes the sum of weights back to 1.
  + Generally, you use an odd number of neighbors (and weights) to be able to get the same number of pixels on both sides of current pixel being smoothed.
* We can easily extend smoothing a 1D vector into a 2D matrix by using a 2D set of weights centered around the pixel currently being smoothed.
  + Example: a 3 x 3 array.

Correlative Filtering

* Formalized formula for correlation filtering with uniform weights:
* Correlation filtering with nonuniform weights:
  + This is called correlation or cross-correlation, denoted as . We would say “G is the correlation/cross-correlation of H with F”.
  + The filter “kernel” or “mask” is the matrix of weights in the linear combination.
  + Example of a kernel (uniform weights):
* It’s not a good idea using a rectangle kernel with uniform weights to blur an image with Gaussian noise. This is because blurring a single pixel into a “blurry” spot would require a matrix that looks like a blurry spot – higher values in the middle, falling off to the edges.
  + You get hard boundaries by applying the rectangular kernel.

Gaussian Filter

* Nearest neighboring pixels having the most influence:
* A true Gaussian function (Isotropic Gaussian) would be:
  + Isotropic means circularly symmetric
* This produces a smoother blur if the image has Gaussian noise.
* Variance or the standard deviation determines the extent of smoothing. The larger the variance/standard deviation, the more smoothing.
  + When we say the size of the kernel, we do not mean the variance. We instead mean the dimensions of the kernel.
* fspecial() is a function that builds image filters for you. Example:

hsize = 31; // Kernel size of 31 x 31.

sigma = 5; // Gaussian standard deviation of 5.

h = fspecial(‘gaussian’, hsize, sigma); //Create a gaussian filter

surf(h); // Plots a 3D surface of the filter

imagesc(h); //Creates a birds-eye view of the 3D filter

outim = imfilter(im, h); //Applies the filter “h” to the image “im”

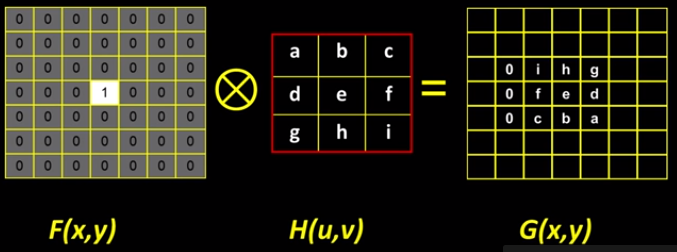
imshow(outim); // Show the image.

* Keeping the Two Gaussians (Noise Gaussian and Filter Gaussian) straight:
  + The sigma in noise affects the noise intensity, whereas the sigma in the filter affects the amount you blur. The sigma in noise is over intensity, whereas the sigma in the filter is over intensity values.

**Linearity and Convolution**

Linearity, Impulse, Correlation, and Convolution

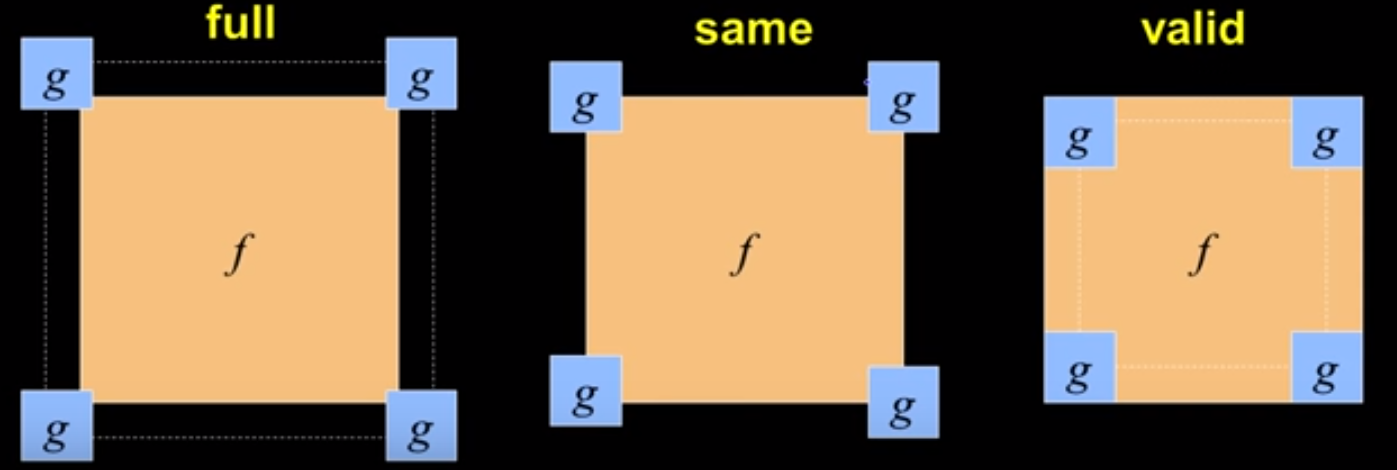
* An operation (or system) is linear if two properties hold ( and are some functions, is a constant):
  + Additivity (things sum):
  + Multiplicative scaling (homogeneity of degree 1 – constant scales):
  + Because it is sums and multiples, the filtering operation we have done is linear.
* Linearity allows us to say how a linear operator affects a function/image by creating a piece of the image at a time, as applying a linear operator to the whole image is the same as applying the linear operator to each of the pieces of the image and summing them together. This can be done by the “building block” of functions known as an impulse.
* Impulse function
  + In the discrete world, an impulse is a value of 1 for a single input, and 0 otherwise.
  + In the continuous world, an impulse is an idealized function that is very narrow and very tall so that it has a unit area (area of 1), i.e. for a very small set of input values, the height is infinite so that the area is 1, 0 otherwise. You take the limit as the width approaches zero and the height approaches infinite so that the area is 1.
* Let’s say you have a “black box”, or unknown system, labelled H. If you put in an impulse into H, the output is an impulse response (or h(t)).
  + If the black box is linear, you can describe H by h(x). This is because any set of impulses (i.e. an entire image) can be created by a scaled and shifted set the single impulse, so our knowledge of how the black box affects the single impulse can be applied to the entire image.
* Example of using an impulse signal to discover the values of the unknown kernel, H:



* + (Assumes the center coordinate is the “reference point” of the filter.)
  + The values in G tell us the values in H, as shown above. Notice that they are flipped.
* Correlation vs. Convolution:
  + (Cross-)Correlation:
  + Convolution (flip F in both dimensions):
  + is defines as the .
* There is no difference between correlation and convolution for symmetric filters (e.g. Gaussian).
  + Correlation filter is denoted as , convolution filter is denoted as
* If you apply the convolution to an impulse, you get the original filter directly because, as shown above, the correlation flips the filter, and so flipping it again via the convolution would achieve the original filter.
* If you convolve or correlate an image with an impulse, you get the original image.
* Properties of Convolution
  + Shift invariance: operator behaves the same everywhere (i.e. the value of the output depends on the pattern in the image neighborhood, not the position of the neighborhood).
  + Commutative:
  + Associative:
  + Identity: unit impulse (for 3 x 3).
  + Differentiation: because the kernel is considered constant, and the derivative is a linear operator and because of associativity.
* The computational complexity of applying a kernel/filter of size W x W to an image of size N x N is equal to
* In some cases, a filter is separable (in which case it is a “linearly separable kernel”), meaning you can get the square kernel H by convolving a single column vector (c) by some row vector (r). Example:
  + So G = H \* F = (C \* R) \* F = C \* (R \* F). So we can do two convolutions with each being , making it a total of . So this is more computationally efficient.

Boundary Issues

* Boundary Issues: issue of what happens when your filter falls off the edge. Three choices:
  + Full: Apply the filter to the image if any part of the filter overlaps with the image. This results in a larger output image.
  + Same: Apply the filter to the image if the center of the filter overlaps with the image. This results in an image of the same size.
  + Valid: Apply the filter to the image only if the entire filter is overlapped by the image. This results in a smaller image.



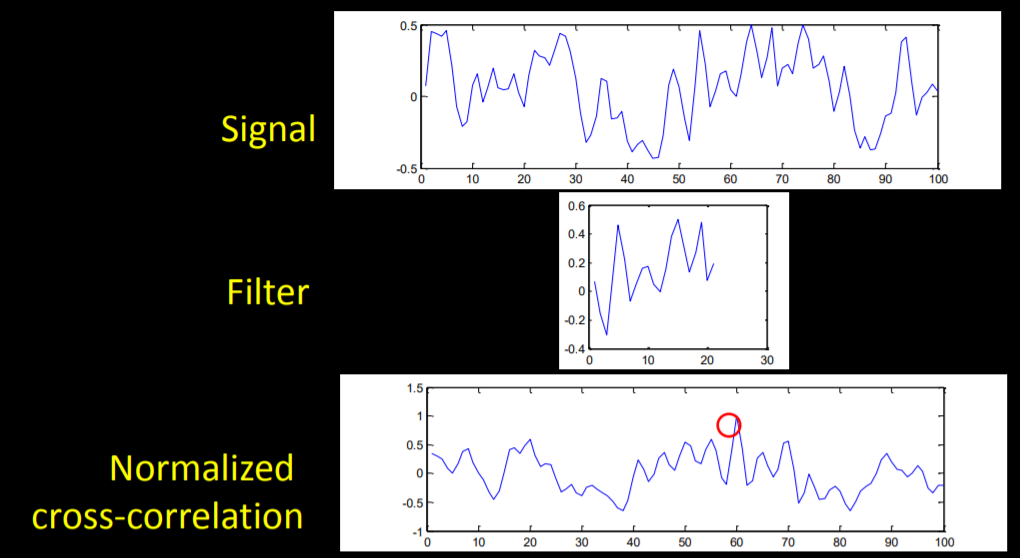
* But what method do we use near the edge (i.e. the filter is not entirely on the image)? How can we extend the image so that the filter always has values from the original image?
  + Clip: assume that the image is black for the pixels that are outside of the image. MATLAB command: imfilter(f, g, 0)
  + Wrap around: assumes the picture continues and wraps around. Not very good…only good for periodic signals. MATLAB command: imfilter(f, g, ‘circular’)
  + Copy edge/replicate: extend out the same value. MATLAB command: imfilter(f, g, ‘replicate’)
  + Reflect across edge: reflect a small portion of the edge to extend the image. This is a good choice because the created imagery has the same statistics as the original image. MATLAB command: imfilter(f, g, ‘symmetric’).
* Special examples of Linear Filters:
  + preserves the original image
  + shifts the image by 1 pixel left with correlation.
  + blurs the image with a box filter.
  + is a sharpening filter: accentuates differences with local average. This is also called the unsharp mask. This works because we effectively subtract off a blurry version of the picture when performing .
* Recall additive Gaussian noise. Another type of noise is the salt and pepper noise (introduced above). The values being injected in this case are totally random, unlike Gaussian noise. Recall from prob/stats that the median instead of the mean is a better choice for this scenario.
  + Median filter: center the filter on the current pixel. Set the current pixel value to be the median of the pixel values that the filter encloses.
  + Good for removing spikes.
  + This is not a linear operator.
  + The median filter is edge preserving (keeps edges sharp). The mean filter blurs/smooths the edges.
  + MATLAB code for adding salt and pepper noise to an image, and then applying a median filter

noisy\_img = imnoise(img, ‘salt & pepper’, 0.02);

median\_filtered = medfilt2(noisy\_img);

**Filters as Templates**

* We will now talk about storing an image not just based on a representation that keeps track of all the intensity values, but rather other image properties (e.g. the similarity of the image with some template at different locations in the image).
  + When comparing the difference between filters, we must keep the multiplicative scale of the filters the same.
* High level idea: filters can be used as templates of what we want to find in the image (i.e. template matching). In other words, we high a high value where the filter matches a portion of the image well, and low value for where the filter does not match the portion of the image well. This can be accomplished via a normalized correlation.
* Two steps for the method of normalized correlation: normalize the filter (e.g. set standard deviation of all pixel to be 1) and then scale the patch of pixels that the filter is currently on so that the standard deviation of the pixels is also one. Now we compute the correlation.
* 1D Correlation: applying a 1D filter to a 1D signal using cross-correlation. The highest value in the cross-correlation is where the filter best matches signal. Example:



* + This applies to 2D as well.
* MATLAB code to find where the filter best matches the signal for 1D:

pkg load image;

function index = find\_template\_1D(t, s) // t = template, s = signal/image

//computes the normalized cross-correlation of template t with s

c = normxcorr2(t, s)

[maxValue rawIndex] = max(c); //Find the max value and its index in c

index = rawIndex – size(t, 2) + 1; //Account for offset

endfunction;

// Test code:

s = [-1 0 0 1 1 1 1 0 -1 -1 0 1 0 0 -1];

t = [1, 1, 0];

index = find\_template\_1D(t, s);

* Mathlab code inding the indices where the filter best matches the 2D image:

Function [yIndex xIndex] = find\_template\_2D(template, img)

c = normxcorr2(template, img);

[yRaw, xRaw] = find(c == max(c(:)));

yIndex = yRaw – size(template, 1) + 1;

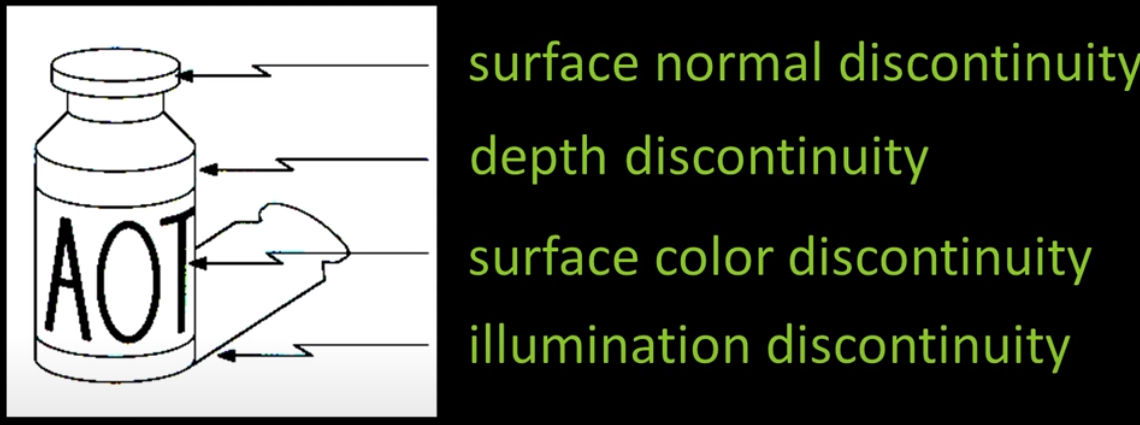
xIndex = xRaw – size(template, 2) + 1;

endfunction

* Template matching is suitable for finding something whose appearance in an image does not vary.
* Template matching between two non-identical objects can be meaningful if the scale, orientation, and general appearance is right.
  + Ex: template matching might work well for finding a car in an image even if the car in the template is slightly different than the car in the image.
  + If your template is somewhat close to what you’re trying to find, then you may be able to narrow down the image to a few candidate choices.

**Edge Detection**

* Edges inside an image are a very important feature. Edges alone can be used to deduce the significance of many images.
* Types of edges:



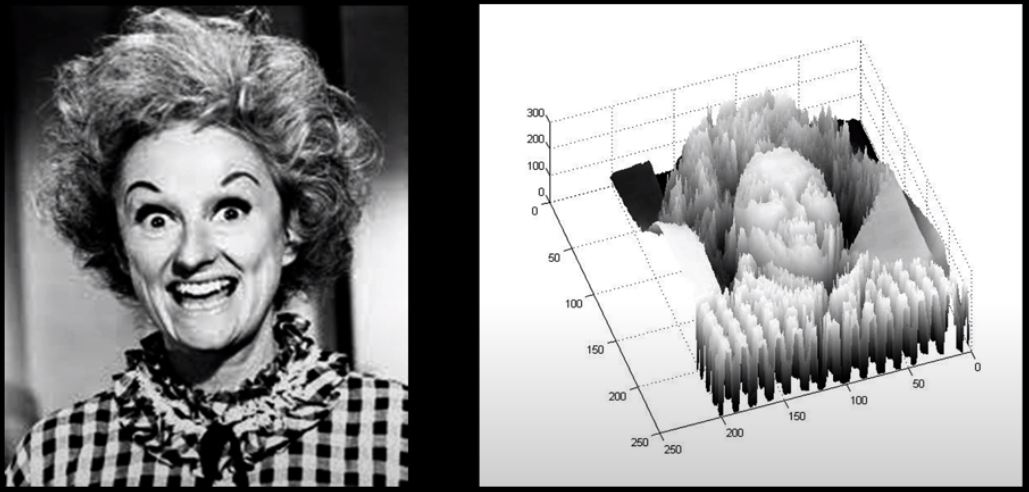
* + Depth discontinuity: the depth of the bottle is different than the depth of the background
  + Illumination discontinuity: there’s a difference between the illumination of the shadow and the non-shadow region
  + Surface normal discontinuity: two surfaces at different angles come together abruptly, forming an angle.
  + Surface color discontinuity: the color of the surface is different.
* Goal of edge detection: convert an image into a reduced set of pixels that are the important elements of the picture.



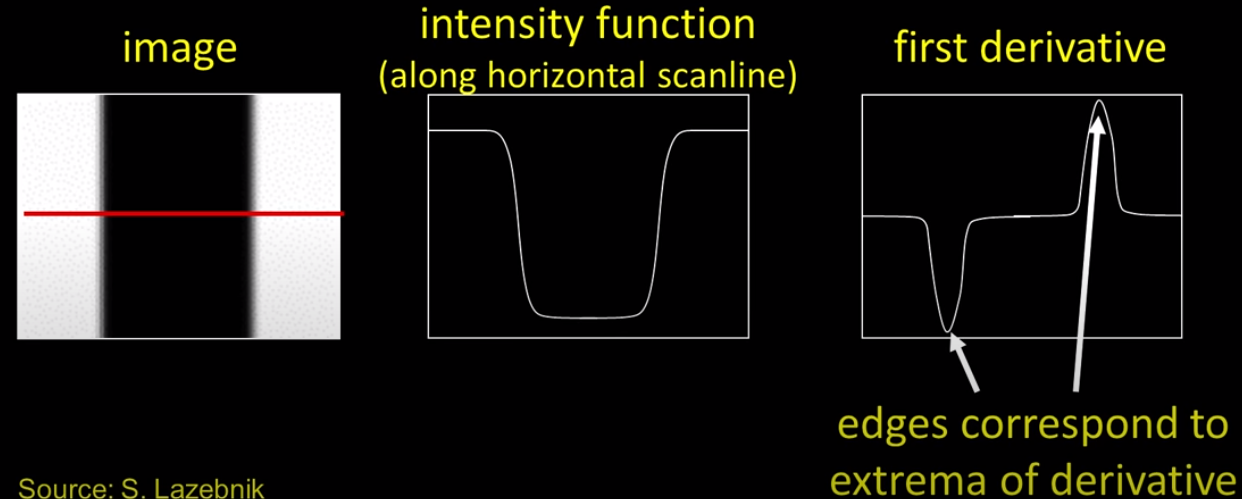
* + We care about boundaries related to shape or illumination, not color discontinuity (i.e. changes in the color of the surface)

Gradients

* Edges look like steep cliffs when we plot the image as a function I(x, y) (i.e. a 3D graph of the image with axes x-coordinate of image as x-axis of the graph, y-coordinate of image as y-axis, and illumination value as z-axis).



* Basic idea of edge detection: look for a neighborhood with strong signs of change:
  + But how large of a neighborhood do we consider, and how much is a strong sign of change?
* An edge is a place of rapid change in the image intensity function (i.e. there is a large derivative).



* + Now we will filter our image with the appropriate operator to find those peaks.
* The partial derivative of a continuous 2D function f(x,y) is . But for discrete image, we can approximate the derivative using finite differences:

This is known as the right derivative, as it takes one step to the right (positive x) direction. Here is an example of applying the right derivative filter:



* + Notice how vertical edges are much more pronounced (as we step right horizontally) than horizontal edges.
  + If the original image sets 0 to be black and 255 to be white, the image of right derivatives has -255 being black and 255 being white.
* We can approximate the y-derivative using finite differences in y: . This makes horizontal edges much more pronounced, as we step down vertically.
* The filter for x-derivatives is as it performs and the filter or y-derivatives is (assuming positive goes down) as it performs . If positive goes up, then the filter is .
* The issue with the above filters is that they are not symmetric around the image point. Hence, a better filter for the x-derivative (vertical edges) would be which takes the average of the “right” and “left” derivative. Similarly, a better filter for the y-derivative (horizontal edges) would be .
* The Sobel operator is an edge detector similar to the operators above, except that they also factor in corner values:
  + Vertical edges :
  + Horizontal edges (here positive is up):
  + Sobel Gradient is , where / is the result of applying / onto the image using correlation. The gradient magnitude is and the gradient direction is .
* Prewitt uses and
* Roberts uses and
* Matlab code for applying the Sobel operator:

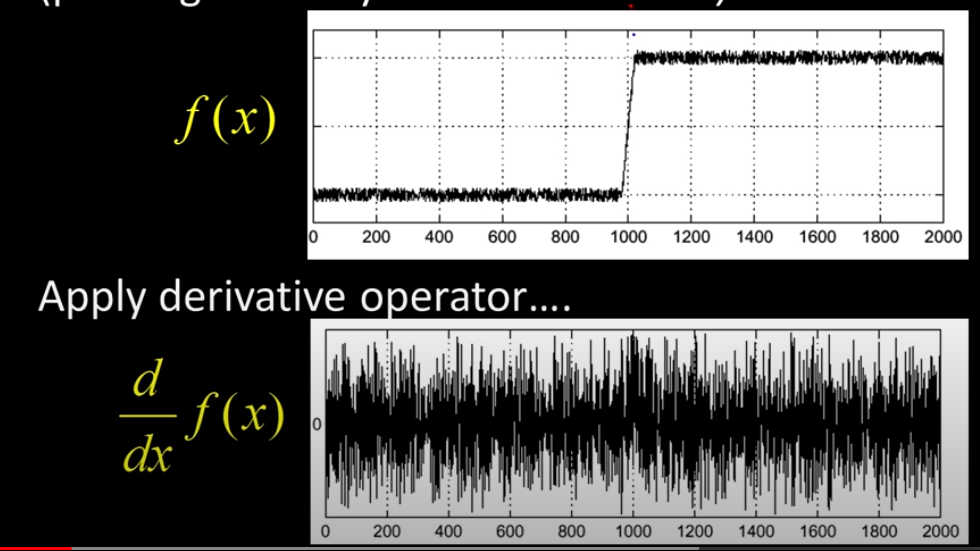
filter = fspecial(‘sobel’);

outim = imfilter(double(im), filter); // by default, does correlation

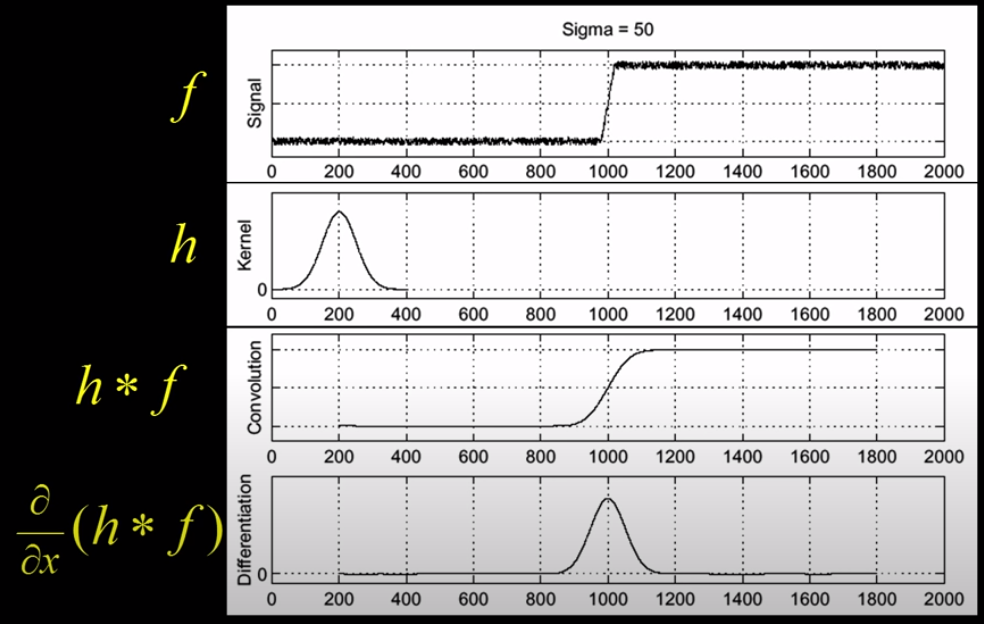
imagesc(outim);

colormap gray;

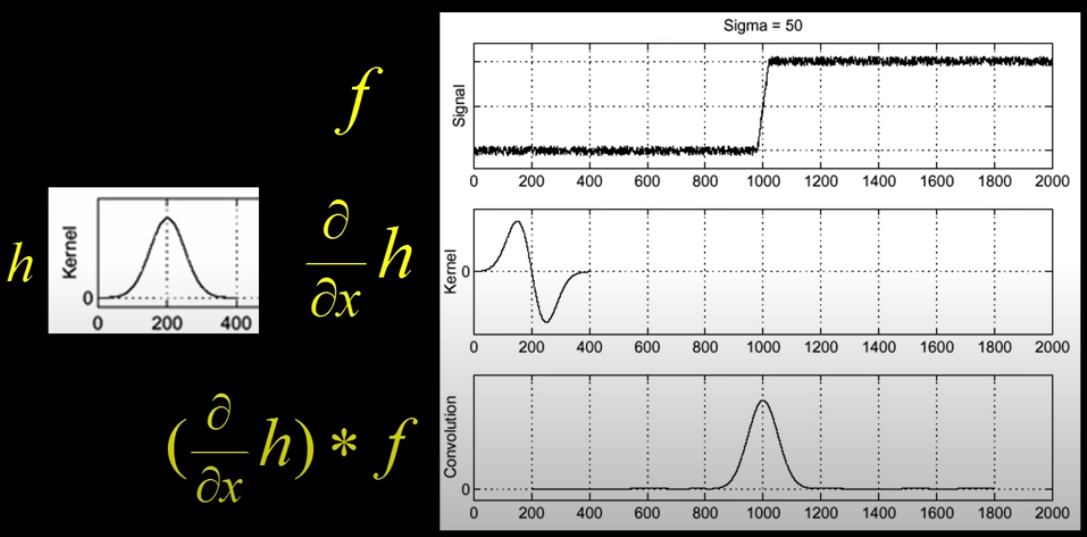
* Be sure to use correlation when applying the edge operators to preserve the intended directionality of gradients.
* The edge detection approach of using gradients won’t work well on images with noise, as the derivatives of noise are also large. See example below:



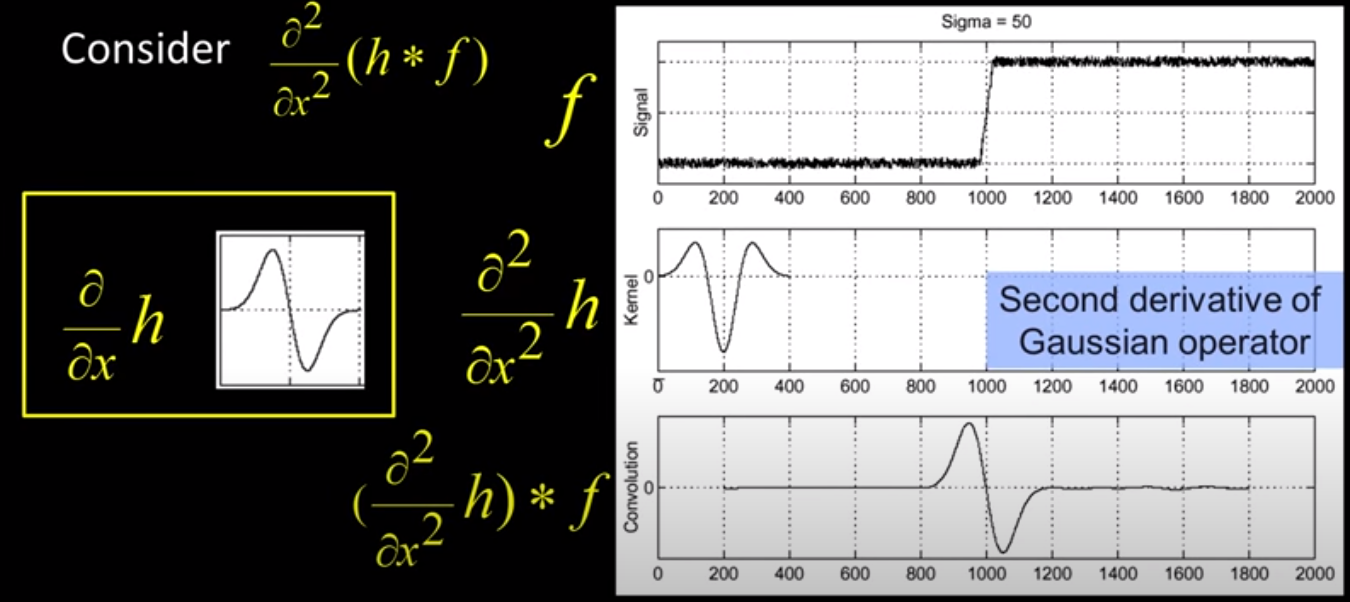
* + Clearly, the edge occurs at , but that’s not easy to tell from .
  + To fix this problem, first smooth the image by filtering the image (e.g. with a Gaussian filter).



* + In the above example, the edge is clearly at the peak of .
  + Due to the property that , we can speed up the computation by taking the derivative of just the kernel (h) instead of the entire filtered image (), as the kernel is often much smaller than the image:



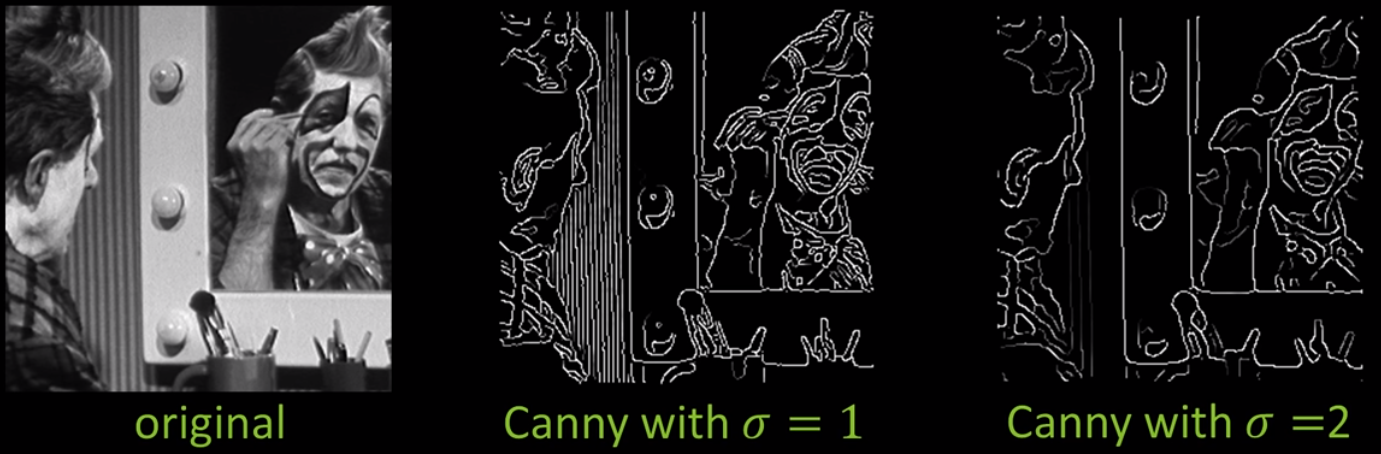
* To find edges, we need to find peaks in , which is equal to . To do so, we take the second derivative of the function: , which is equal to . We do so by taking the second derivative of the Gaussian function and correlating/convolving that with the image :



* + Wherever the second derivative is equal to zero and where the magnitude of the derivative of the second derivative (i.e. third derivative) is large, there is an edge.

2D Operators

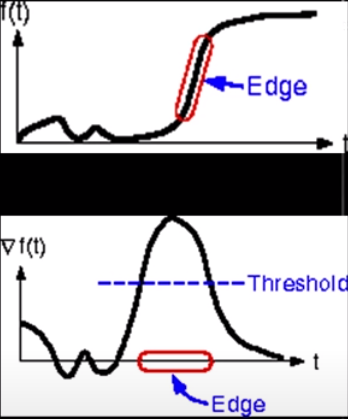
* When taking a 2D derivative, we also need to specify what direction we are taking the derivative in.
  + 2D Gaussian filter example: . is an operator in the x-direction (e.g. Sobel operator), is the Gaussian filter, and is the image.
* By increasing when smoothing the image using a Gaussian filter, the derivatives for edge detection will become smaller. Thus, increasing means that only larger scale (i.e. sharper) edges will be detected. The choice of depends on desired behavior.



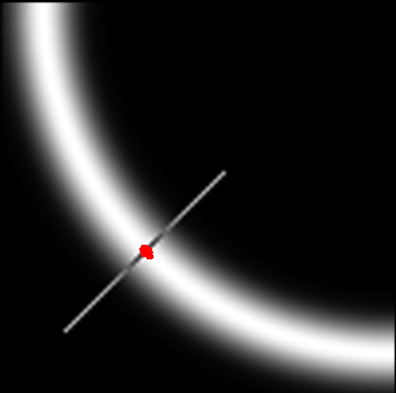
* High-level edge detection steps:
  + Smoothing derivatives (e.g. via a Gaussian filter) to suppress noise and compute gradients.
  + Threshold to find regions of “significant” gradient.
  + “Thin” to get localized edge pixels (as you would get regions of edge pixels from just doing the previous step)
  + Add link or connect edge pixels (as the edges are not necessarily connected from just performing the previous step).
* High-level description of the edge detector developed by John Canny:
  + Filter image with derivative of Gaussian
  + Find magnitude and orientation of gradient
  + Non-maximum suppression: thin multi-pixel wide “ridges” down to a single pixel width (before on left, after on right):

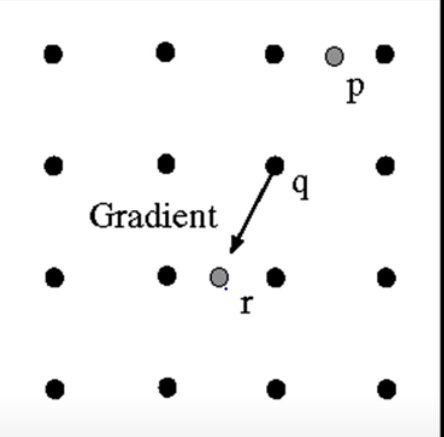
* + Linking and thresholding (hysteresis): define two thresholds – low and high. Use the high threshold to start edge curves and the low threshold to continue them.
* MATLAB code for Canny edge detector: edge(*image*, ‘canny’)
* The reason why non-maximum suppression is needed is because there may be a large region of pixels above the threshold gradient, thus making a multi-pixel wide edge:



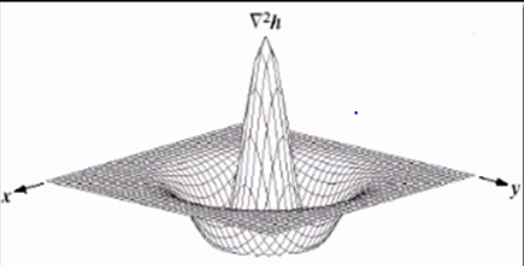
* + Non-maximum suppression looks at the direction of the gradient and finds the local maximum (marking the red dot as the thinned edge):



* + One detail is that you can interpolate sub-pixels and between pixels to get sub-pixel accuracy:



* However, performing just non-maximum suppression will drop weaker edges if they don’t cross the gradient threshold. Canny threshold hysteresis fixes this:
  + Apply a high threshold to detect strong edge pixels (as described above)
  + Link those strong edge pixels to form strong edges.
  + Apply a low threshold to find weak but plausible edge pixels.
  + Extend the strong edges to follow weak edge pixels.
* Single 2D Edge detection Filter
  + Recall above that we use the second derivative of the Gaussian filter to find edges in a 1D image, which is robust to noise.
  + To find edges in a 2D image, one complication is: which direction do we take the two derivatives in. The correct answer is to use the Laplacian of Gaussian operator:



* + Apply the Laplacian of Gaussian operator to the image, and take the zero-crossings (with significant gradient) to be the edges.
  + Matlab code for Laplacian of Gaussian: edge(*image*, ‘log’)

**Hough Transform**

Lines