Computer Vision

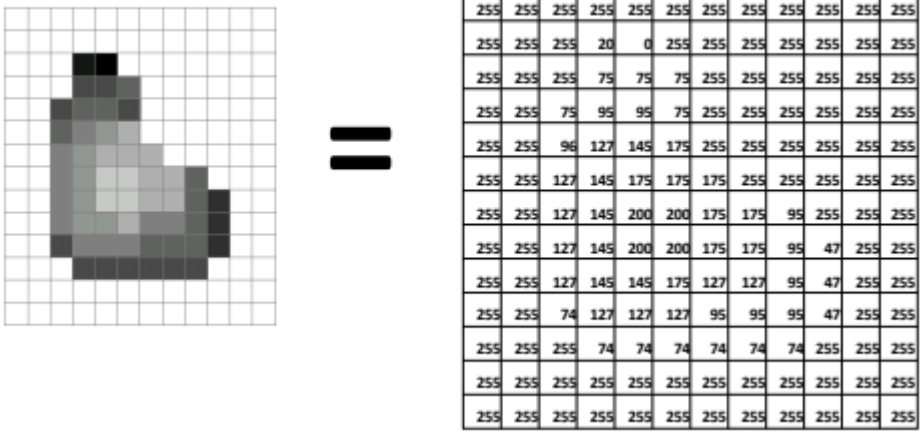
Simone Palumbo's Notes 2023/2024
MSc in Artificial Intelligence and Robotics

Image Processing

28/02/2024, 5/03/2024

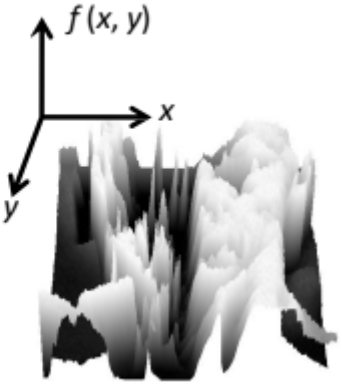
Images as Functions

A grayscale image is a matrix of integer values that represent intensity. For each pixel (e.g. cell of the matrix) we use a byte: 0 is black, 255 is white.



255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	20	0	255	255	255	255	255	255	255
255	255	255	75	75	75	255	255	255	255	255	255
255	255	75	95	95	75	255	255	255	255	255	255
255	255	96	127	145	175	255	255	255	255	255	255
255	255	127	145	175	175	175	255	255	255	255	255
255	255	127	145	200	200	175	175	95	255	255	255
255	255	127	145	200	200	175	175	95	47	255	255
255	255	127	145	145	175	127	127	95	47	255	255
255	255	74	127	127	127	95	95	95	47	255	255
255	255	255	74	74	74	74	74	74	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255

A grayscale image is a function f(x,y): R^2 -> R that gives the intensity of pixel in position (x,y).



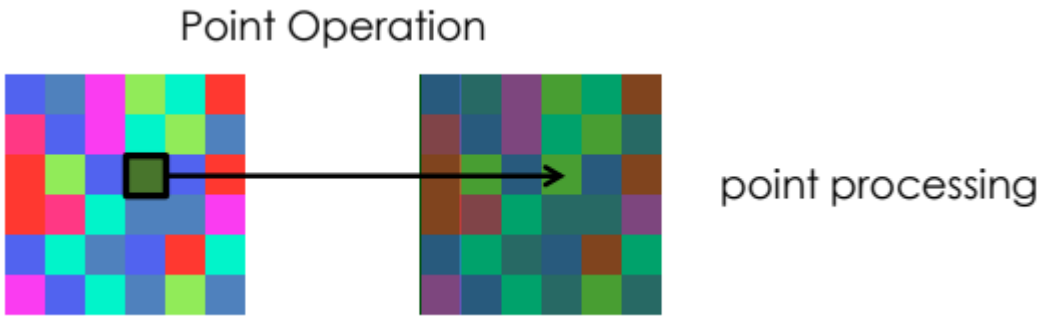
A <u>digital</u> image is a discrete (sampled and quantized) version of this continuous function f(x,y): R^2 -> R.

For colored images we have a tensor of three matrices that represent the channels. E.g. the RGB model has three matrices, for red, green and blue.

Image Processing

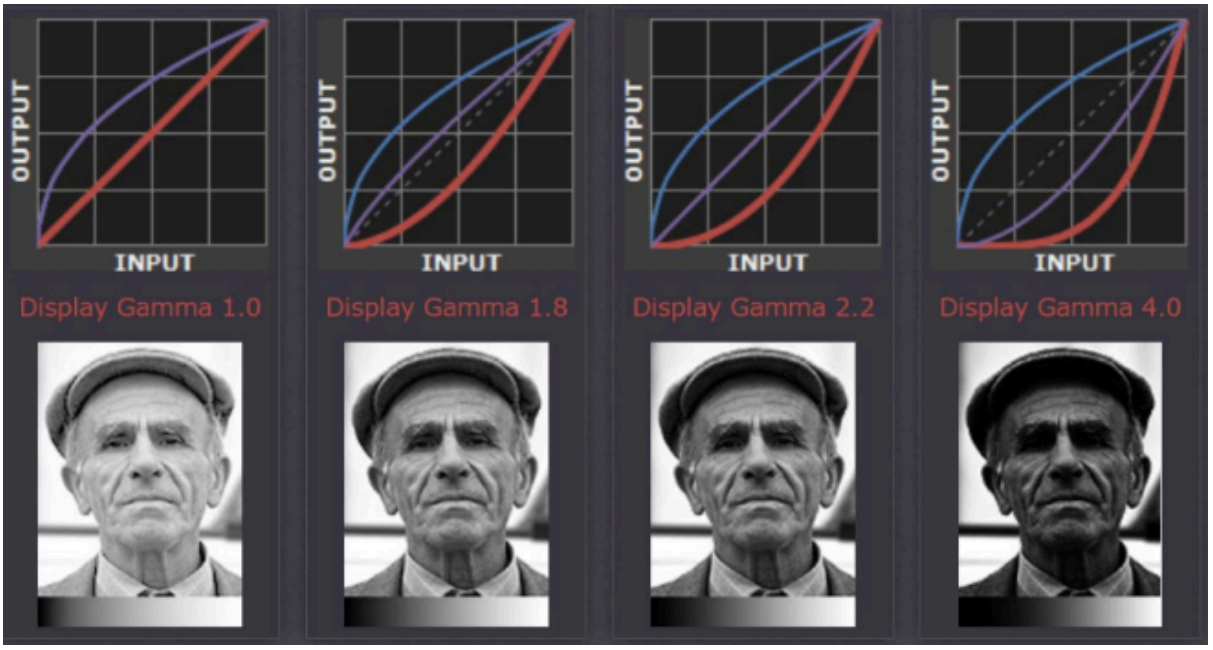
With an image we can do the following thing:

Point Processing Operations



Let's see some point processing examples

An important point processing operation is the **Gamma Correction**[1]:



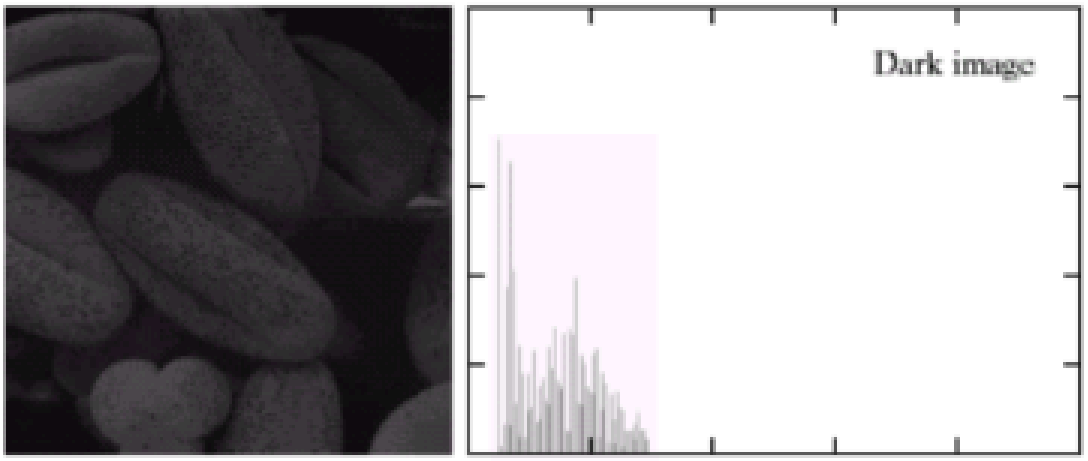
$$V_{\text{out}} = AV_{\text{in}}^{\gamma},$$

[1] As you can see from the image above, increasing the gamma gives more details. Standard value for gamma is 2.2

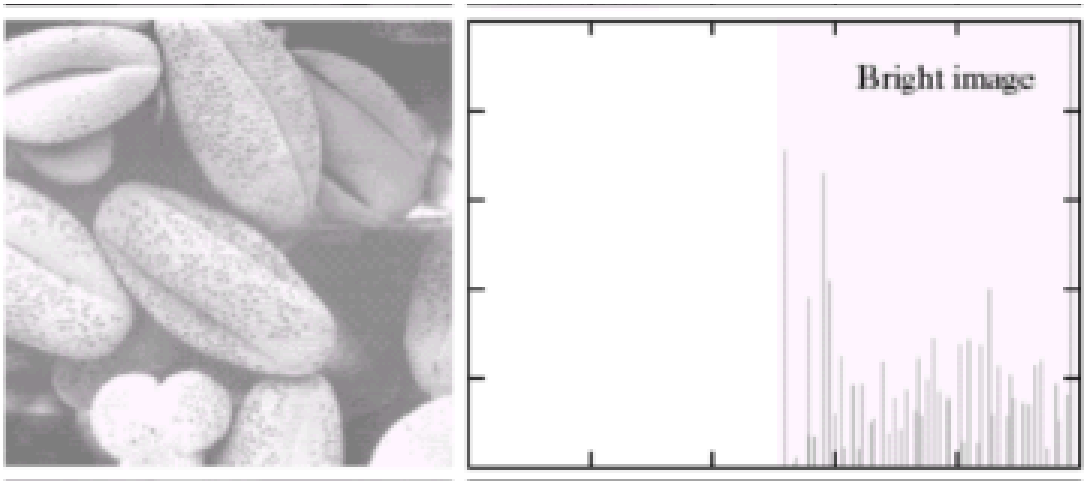
Global Histogram Equalization

First we need to build an intensity histogram of the image. For doing so, we do, for each intensity r:

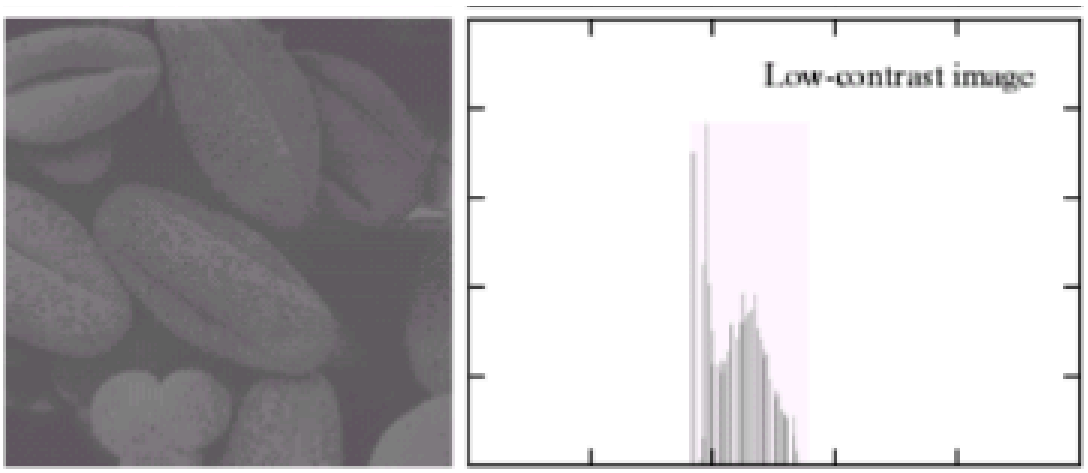
$$p(r) = \left(\frac{\text{Number of pixels with intensity } r}{\text{Total number of pixels}} \right)$$



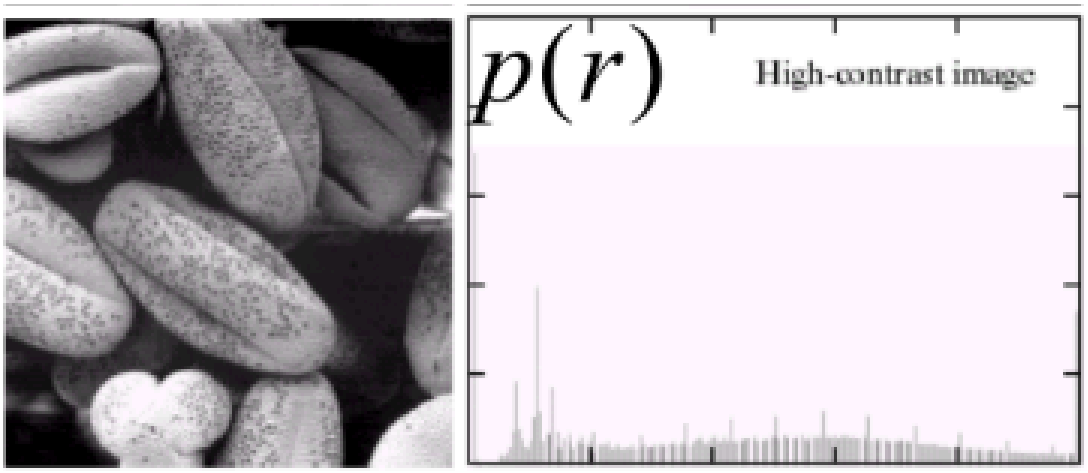
The intensity histogram of this image has higher bars for low intensity values, so it's a dark image



The intensity histogram of this image has higher bars for high intensity values, so it's a bright image



Here there is a small number of intensities r considered, so the pixels have almost the same intensities, and thus the image is low-contrast.



This image is equalized in terms of intensities considered. Thus, since almost every intensity is present in the image, we have a high contrast image.

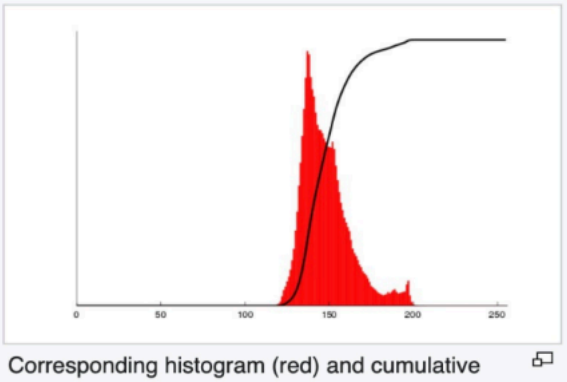
Now, if we want to equalize the histogram of an image, in order to change the intensity in such a way that more details are visible, we can do the following[2]

$$T(r) = round\left(255\frac{\text{Number of pixels with intensity } i \leq r}{\text{Total number of pixels}}\right)$$

$$= round\left(255\sum_{i=0}^{r}\frac{\text{Number of pixels with intensity } i}{\text{Total number of pixels}}\right)$$

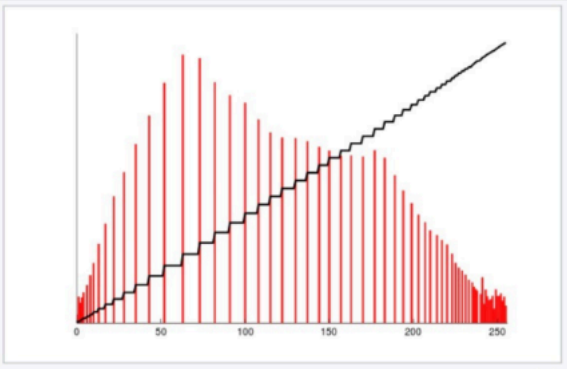
$$= round\left(255\sum_{i=0}^{r}p(i)\right)$$

Before Histogram Equalization



Corresponding histogram (red) and cumulative histogram (black)

After Histogram Equalization



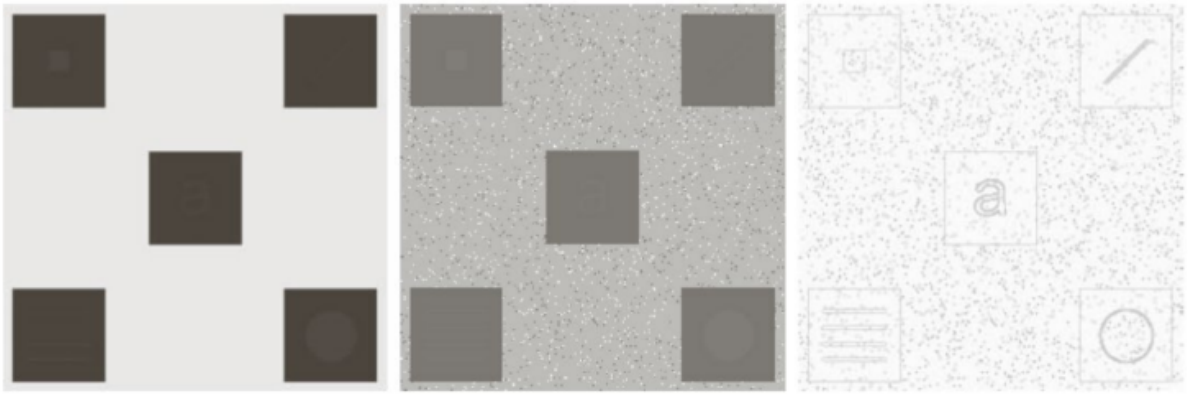
Corresponding histogram (red) and cumulative histogram (black)

The upper histogram is a cumulative histogram (in black), obtained by counting every intensity, which is not nice since it shows that the intensity is only in a particular part of the image, so it's pretty tight. The histogram below, the equalized one shows that after the equalization the cumulative histogram has a nice, linear, behavior.

[2] with r from 0 to (number of bins - 1). The maximum number of bins is of course 255.

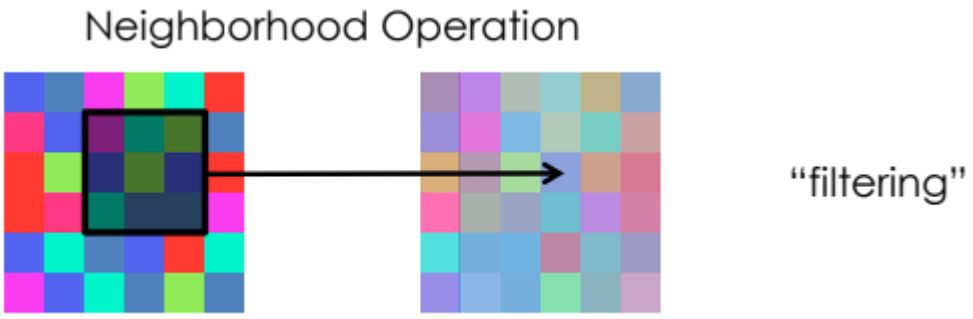
Local Histogram Processing

For each pixel we select a suitable neighborhood on which the histogram equalization is computed. So we calculate T(r) as the sum of p(i) (as in global histogram processing) but this time p(i) is computed on a subset of the entire set of pixels. The local approach is better than the global one: it is more computationally intensive than the global, but with the global approach, you also raise noise, since you are considering, for the computation of T(r), pixels that are far away, whereas generally a pixel in the image is not correlated to pixels that are too far away, but only to pixel in a locality



The first is the original image. In the center you have the (globally) equalized image, while on the right there is the locally equalized image with 3x3 neighborhood.

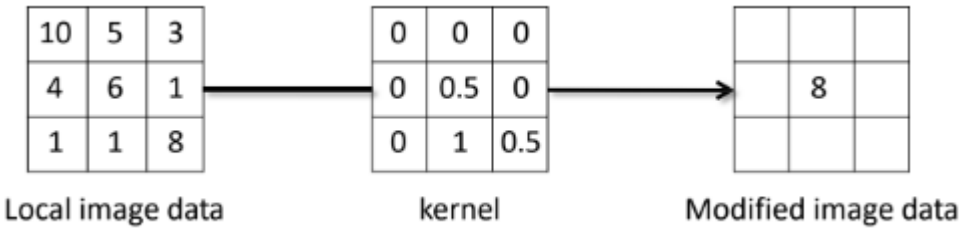
Filtering



With filtering we are forming a new image whose pixel values are a combination of the original pixel values of certain neighboring pixels in the original image. They are used for:

- extract useful information: e.g. edges
- enhancing images: e.g. remove noise, sharpening…
- key operator in CNNs

Linear filters, such as the Gaussian filter, can be implemented as convolutions, while non-linear ones cannot. To do linear filtering, we do cross-correlation or convolution between it and a filter[3], doing a linear combination (weighted sum) of the neighbor of the pixel



Examples of non-linear filters are the Bilateral and the Thresholding filters.

Cross-Correlation

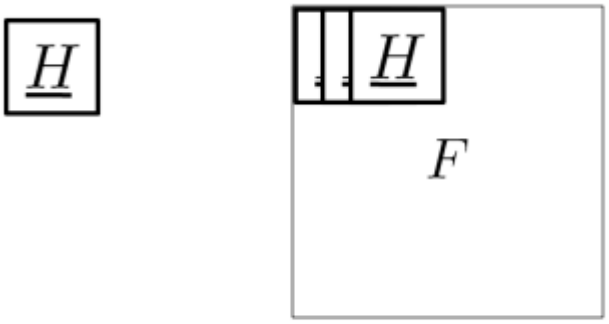
Given an image F and a kernel H of size (2k+1) x (2k+1), the output image G will be:

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i+u,j+v]$$

You can think of G as a dot product between the local neighborhood and the kernel, done for each pixel.

$$G = H \otimes F$$

An animation to see the cross-correlation is this one:



Cross-Correlation is neither associative nor commutative. In literature, it is often called just correlation or even “convolution”.

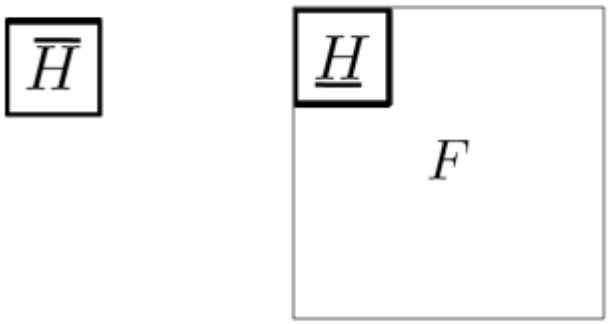
[3] also called mask or kernel

Convolution

Convolution is the following operation:

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i-u, j-v]$$

so in convolution the kernel is flipped horizontally and vertically.



Since the filters are generally symmetric, it does not matter and for this reason in literature “convolution” is synonym of “correlation”. The convolution has the following properties:

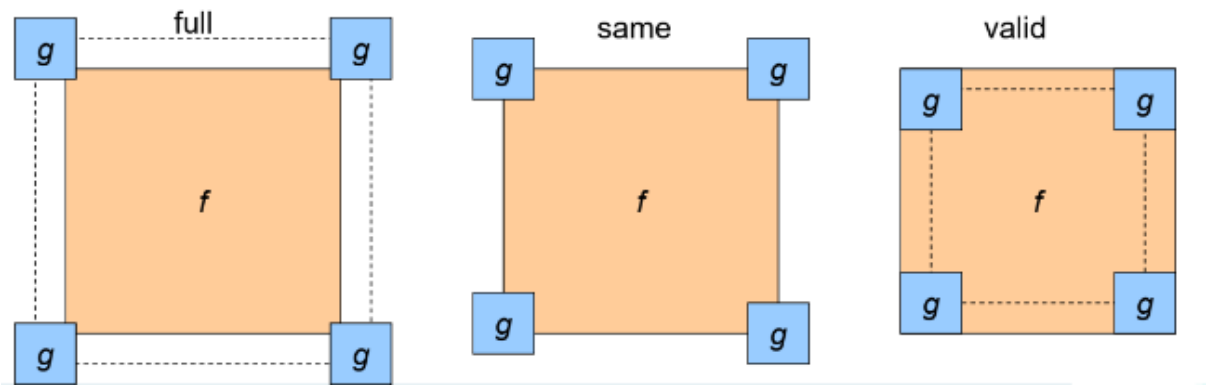
- commutative $a \star b = b \star a$
- associative $a \star (b \star c) = (a \star b) \star c$
- distributes over addition $a \star (b + c) = a \star b + a \star c$
- scalars factor out $\alpha a \star b = a \star \alpha b = \alpha(a \star b)$
- identity: unit impulse *e* = [..., 0, 0, 1, 0, 0, ...] $a \star e = a$

Associativity is really useful since we often apply several filters, since instead of applying several filters one after another, we can apply one filter that is obtained by convolving all the filters.

Padding and Stride in Convolution

What is the size of the output of the convolution? It depends on the padding techniques used during the convolution:

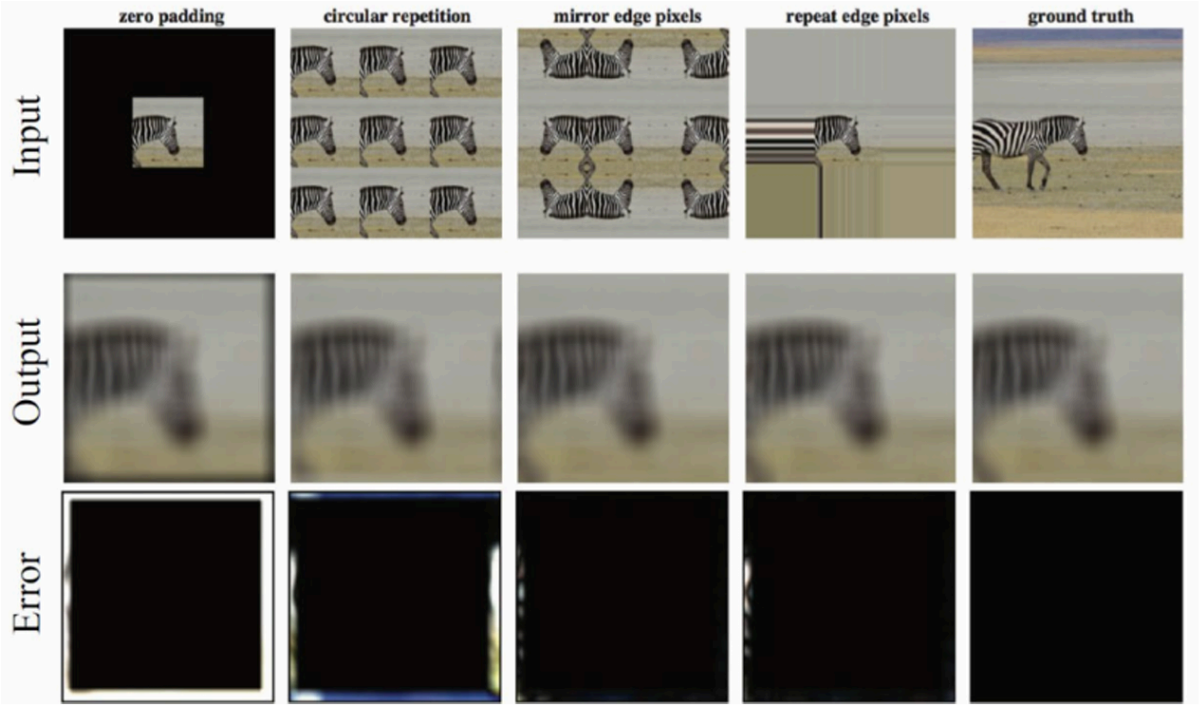
- Same Padding: the input images is padded in order for the output feature map to have the same spatial dimensions as the input
- Valid Padding: no padding is added. As a result, the output feature map may have smaller dimension that the input
- Full Padding: the input is padded as much as possible for the size of the filter. That results in a larger output feature map compared to the input.



You can see a visualization of this concept here.

What value to use for the padding pixels? We can use:

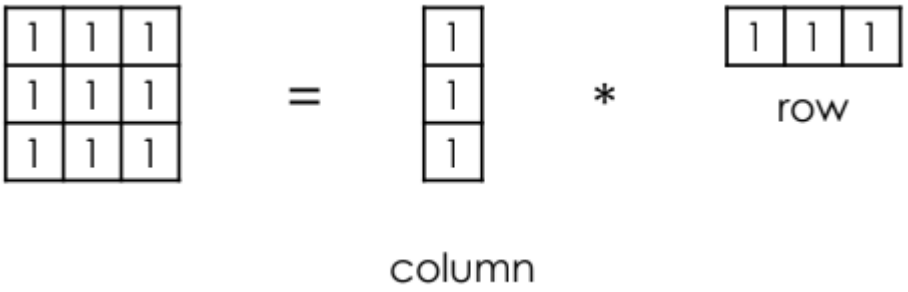
1. Zero Padding: set all pixels outside the source image to 0.
2. Circular Repetition
3. Mirror Edge Pixels: reflect pixels across the image edge
4. Repeat Edge Pixel: repeat a certain pixel



Besides the padding, another important option for the convolution is the stride. The stride tells us how much we shift after applying the operation.

Separable Filter

A 2D filter is Separable if it can be written as the product of a column and a row

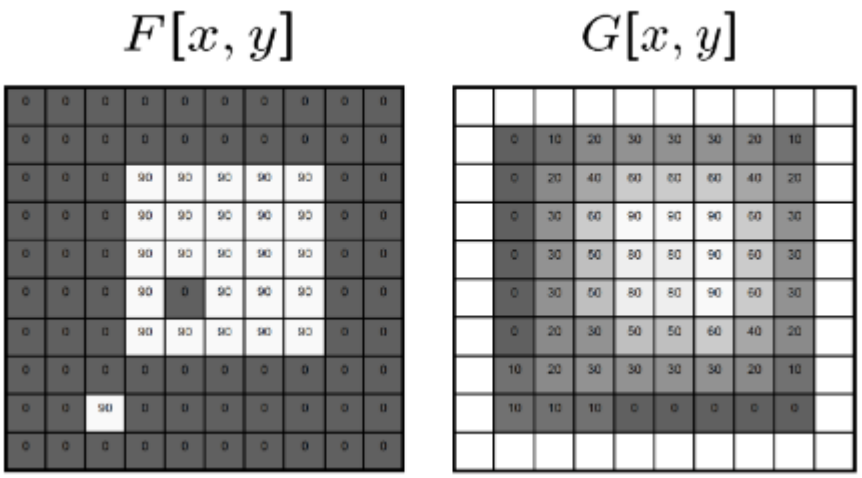


A 2D convolution with a separable filter is equivalent to 2 1D convolution. Why is it useful? If the image is M x N and the filter is L x L:

- The cost of convolution with a non-separable filter is L^2 x M x N
- The cost of convolution with a separable filter is 2 x (L x M x N)

Box Filter

One of the simplest filter to implement is the box filtering that just does the average of the pixels in the window.

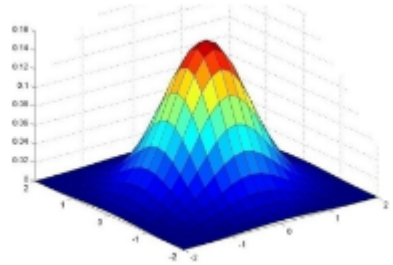


Gaussian Filter

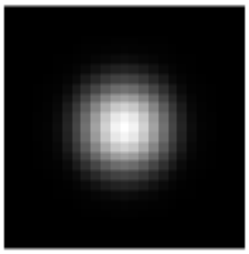
An important linear filter.

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

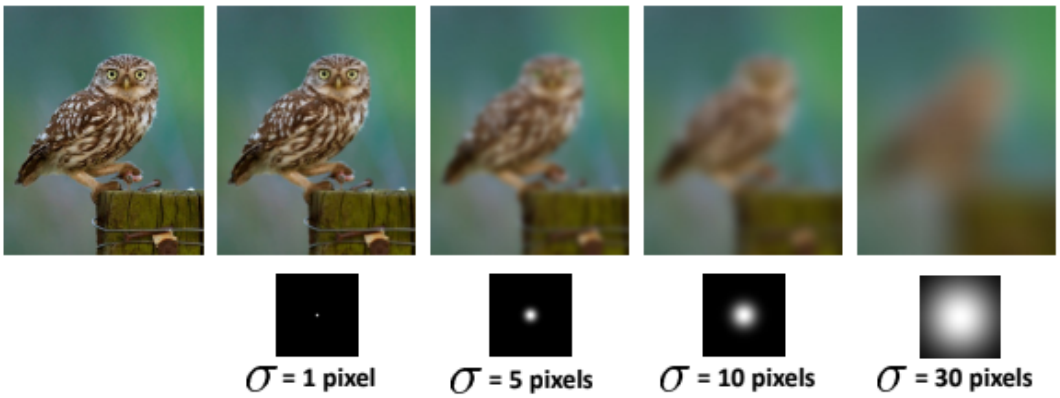
Computed with the gaussian formula above. In a 3 dimensional way is represented like this:



While in 2D is:



So we are seeing the gaussian from above. Example:



When sigma increases the image becomes more blurred.

Properties of the Gaussian Kernel

1. Convolving two Gaussian filters results in another Gaussian.

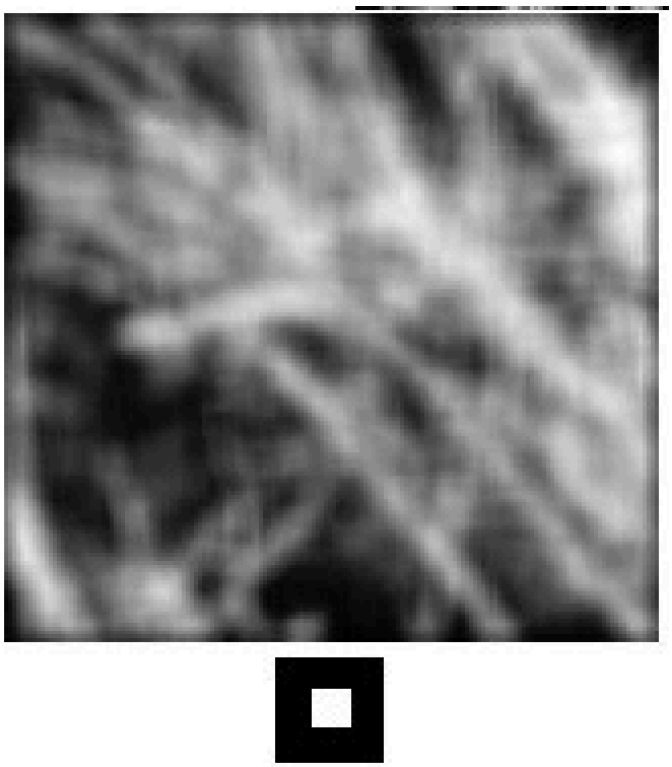
Specifically, performing two convolutions with a Gaussian kernel of width σ is equivalent to a single convolution with a kernel width of σ√2.

2. The Gaussian kernel is separable in two 1D Gaussian Kernel

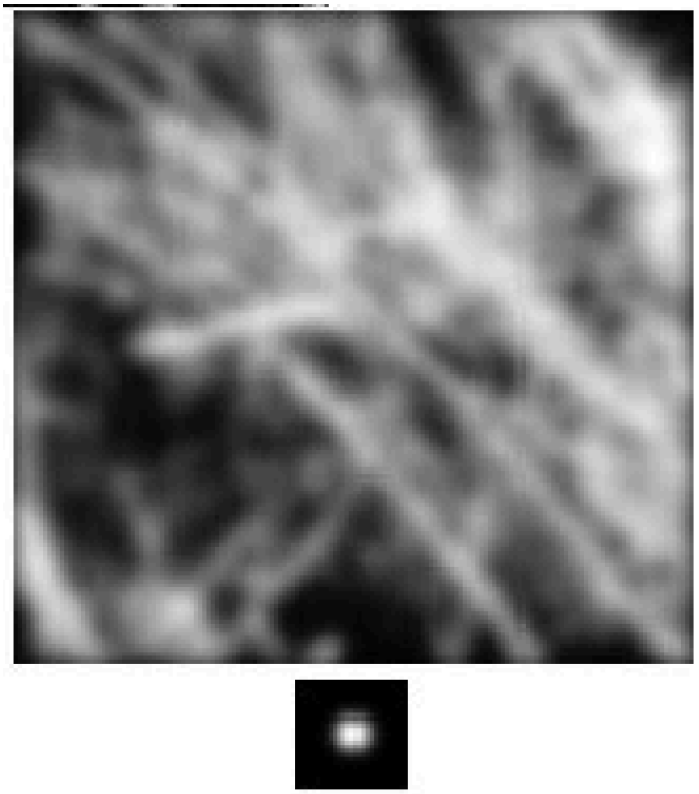
Gaussian vs Box Filter

Comparison between box filtering and gaussian filtering. This is the original image:

This is after applying a box filtering:



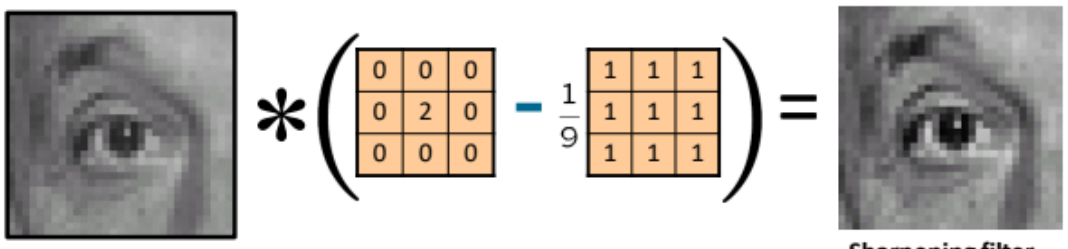
This is after applying a gaussian filter:



Is not really visible, but the box filtering is “blocky” while the gaussian is smooth.

Sharpening

As we said, we can combine multiple kernels for doing convolution with the image. For instance:

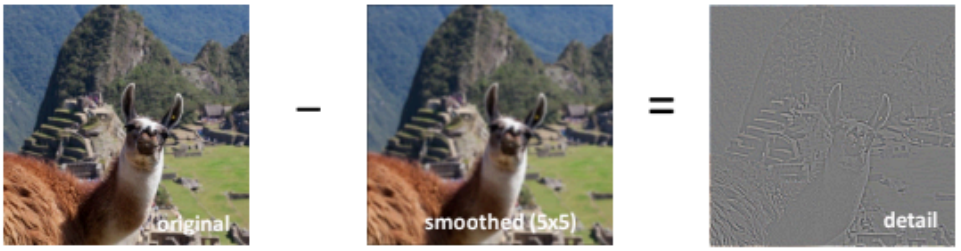


Original

Sharpening filter
(accentuates edges)

This is the sharpening filter, which accentuates edges. We can sharpen an image in two ways:

1. doing the operations singularly one after the other:



(This "detail extraction" operation is also called a ***high-pass filter***)

Let's add it back:

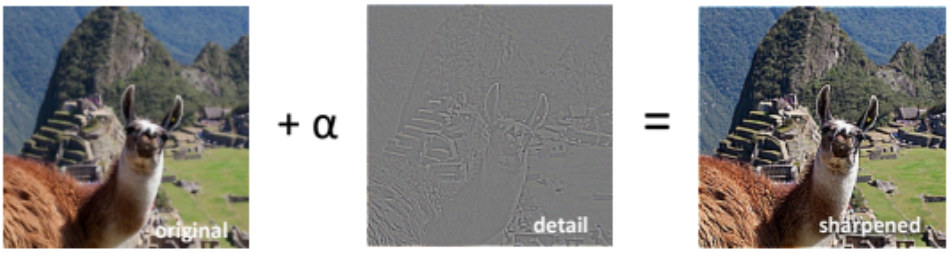


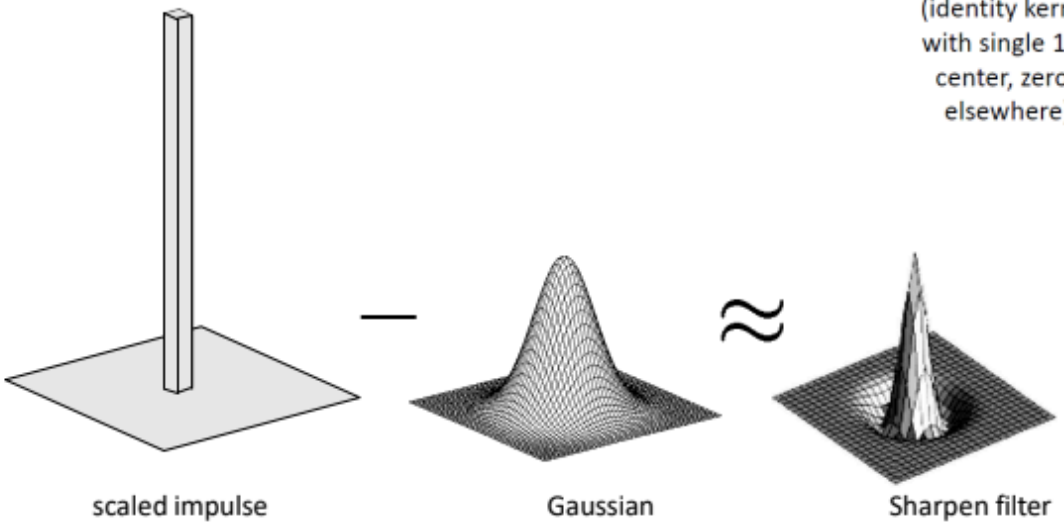
Photo credit: https://www.flickr.com/photos/geezaweezer/16089096376/

so we remove a smoothed version of the image from the original, basically doing a "detail extraction", also called high-pass filtering. Now we can add the details back to the original image, but scaled by a factor alpha, getting a sharpened image.

2. Or we can apply directly the sharpen filter:

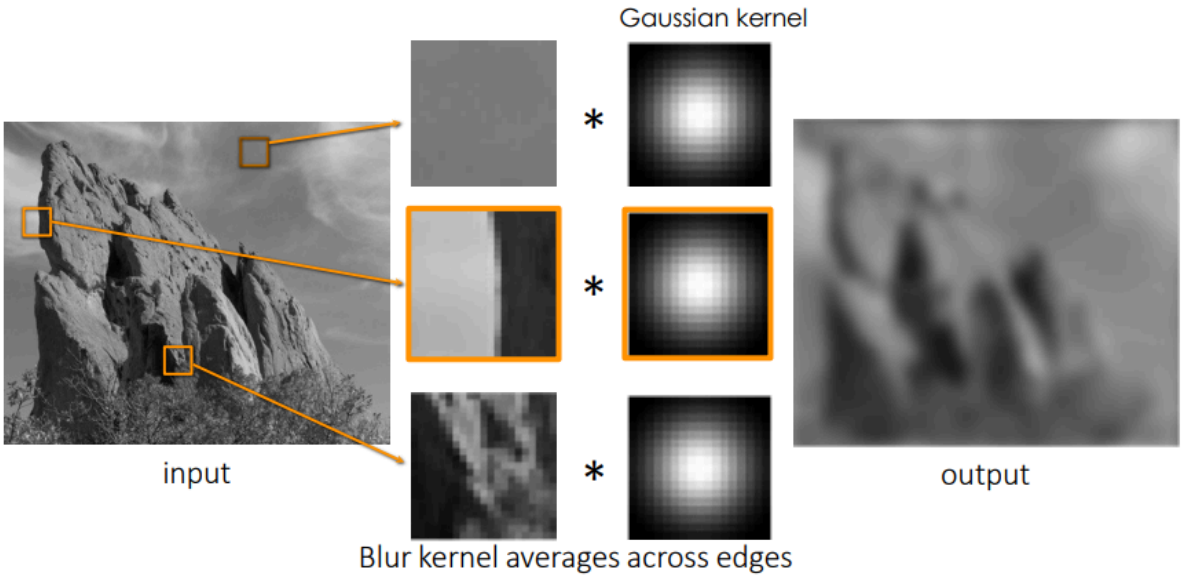
$$F + \alpha\,(F - \underbrace{F * H}_{\text{blurred image}}) = (1+\alpha)\,F - \alpha\,(F * H) = F * (\underbrace{[1+\alpha]\,e}_{\text{unit impulse (identity kernel with single 1 in center, zeros elsewhere)}} - \alpha H)$$

image



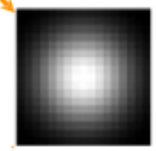
Bilateral Filter

There is a problem with Gaussian Filtering: the gaussian kernel blurs all the images, in the same way in all the parts.



Gaussian filtering

$$h[m,n] = \sum_{k,l} g[k,l] f[m+k, n+l]$$

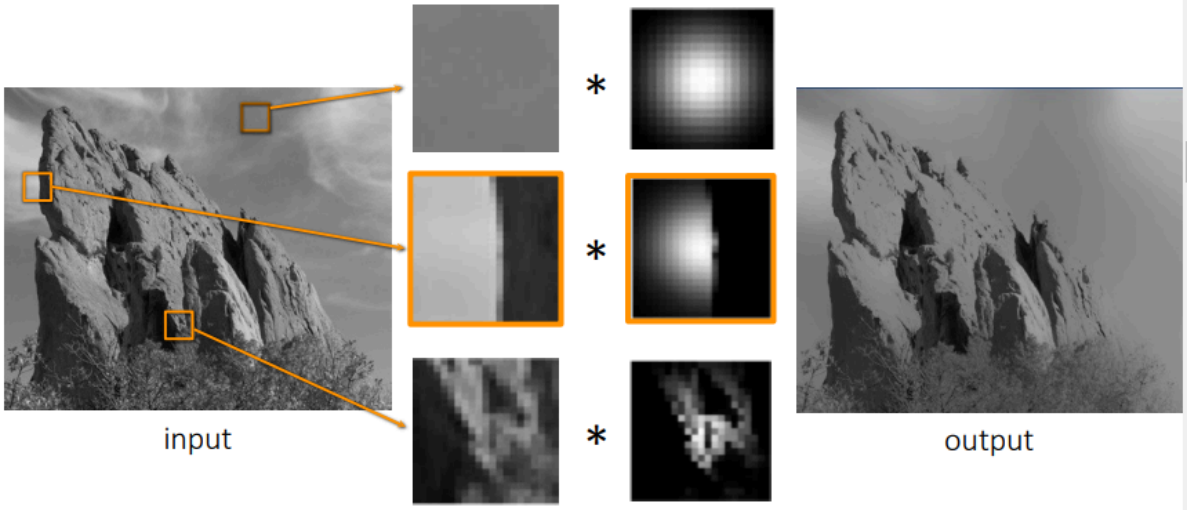


σ_s

Spatial weighting:
favor *nearby* pixels

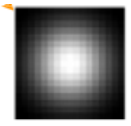
This is good if you want to just blur the image. If you want to preserve something in the image this is not good. σ_S determines how much we want to blur the image. **g[k,l]** is the gaussian kernel while **f[m + k, n + l]** is the image. g[k,l] smooths pixels that are nearby in spatial distance.

We can solve this problem using the bilateral filtering:



Do not blur if there is an edge! How does it do that?

Bilateral filtering

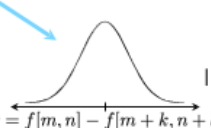


σ_s

Spatial weighting:
favor *nearby* pixels

$$h[m,n] = \frac{1}{W_{mn}} \sum_{k,l} g[k,l] r_{mn}[k,l] f[m+k, n+l]$$

Normalization factor



σ_r

$x = f[m,n] - f[m+k, n+l]$

Intensity range weighting:
favor *similar* pixels

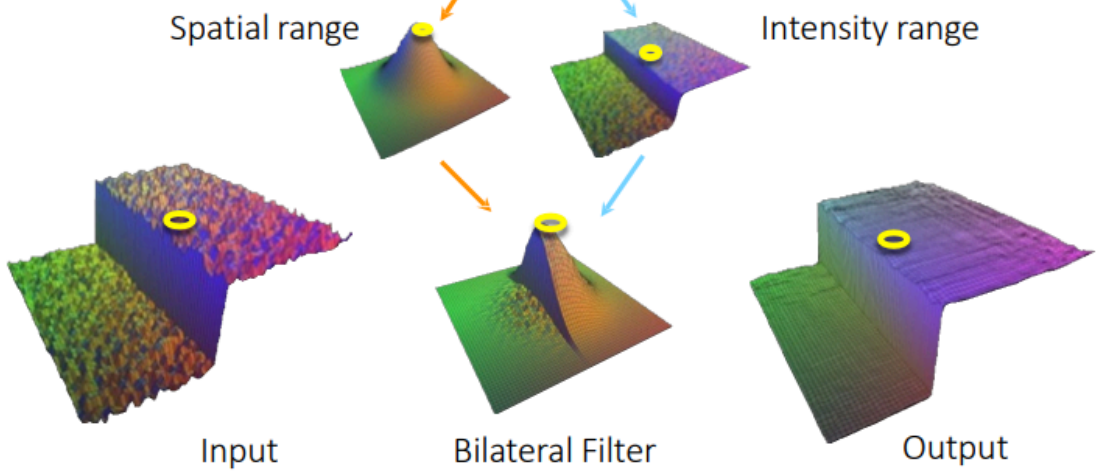
This applies the gaussian kernel only in the uniform part of the image, on the edges the gaussian filter is modified to preserve them. **r_{mn}[k, l]** is the intensity range weighting that favors similar pixels. If the intensity of two adjacent pixels is very different the region is not uniform, so there is an edge, and we want to treat it differently, as we said. This is recognized by r_{mn}[k, l]. Moreover, we have a normalization factor. While the gaussian kernel depends only on the spatial distance, the bilateral kernel depends on spatial and intensity

distance, so it smooths pixels that are nearby in space and intensity. *Note that this is a non-linear filter, and so it cannot be implemented as a convolution*.

Let's do an example:

$$h[m,n] = \sum_{k,l} g[k,l] f[m+k, n+l]$$

$$h[m,n] = \frac{1}{W_{mn}} \sum_{k,l} g[k,l] r_{mn}[k,l] f[m+k, n+l]$$



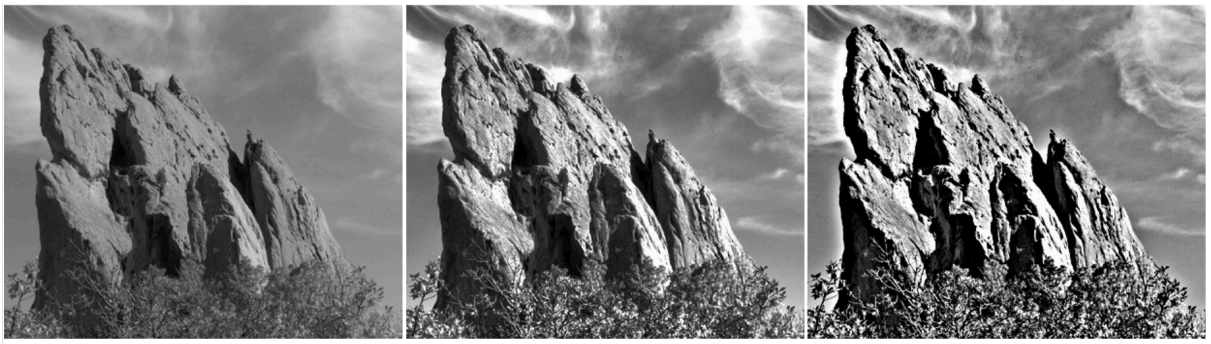
Application of Bilateral Filtering:

1. Denoising with Bilateral Filtering:

noisy input | bilateral filtering | median filtering

The input has a lot of noise in the white part, applying the bilateral filtering, or also the median filtering, we reduce the noise.

2. Contrast Enhancement with bilateral filtering:



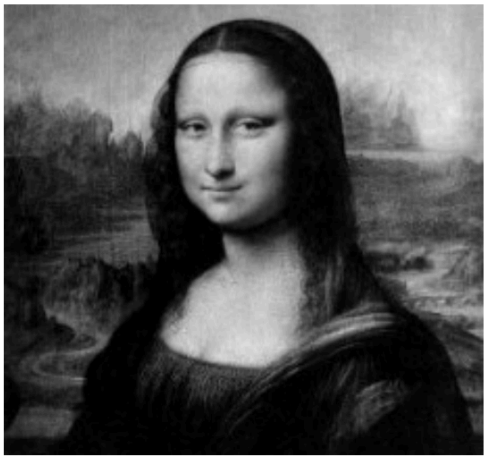
input | sharpening based on bilateral filtering | sharpening based on Gaussian filtering

3. Cartoonization with bilateral filtering:

input | cartoon rendition

The bilateral filter will reduce the color palette so it is useful to cartoonize.

Thresholding filter



$$g(m,n) = \begin{cases} 255, & f(m,n) > A \\ 0 & otherwise \end{cases}$$

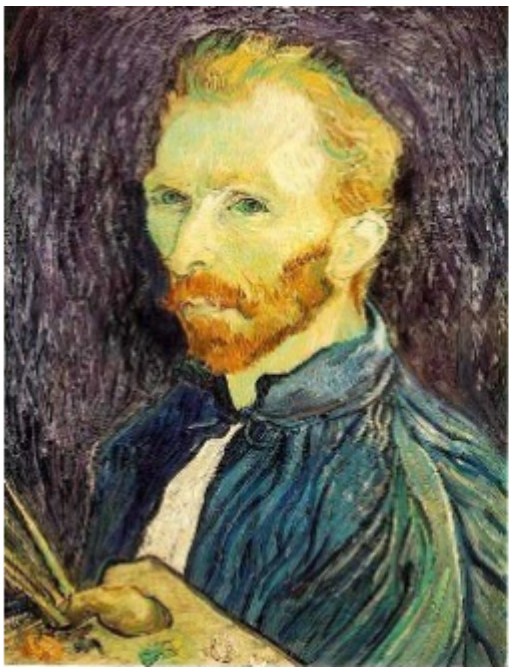
This can be considered the first segmentation algorithm, even if it's not useful for segmentation. It's just transforming a grayscale image to black and white. *Note that this is a non-linear filter, and so it cannot be implemented as a convolution*.

Image Resizing

Downsampling

Naive Subsampling: Aliasing

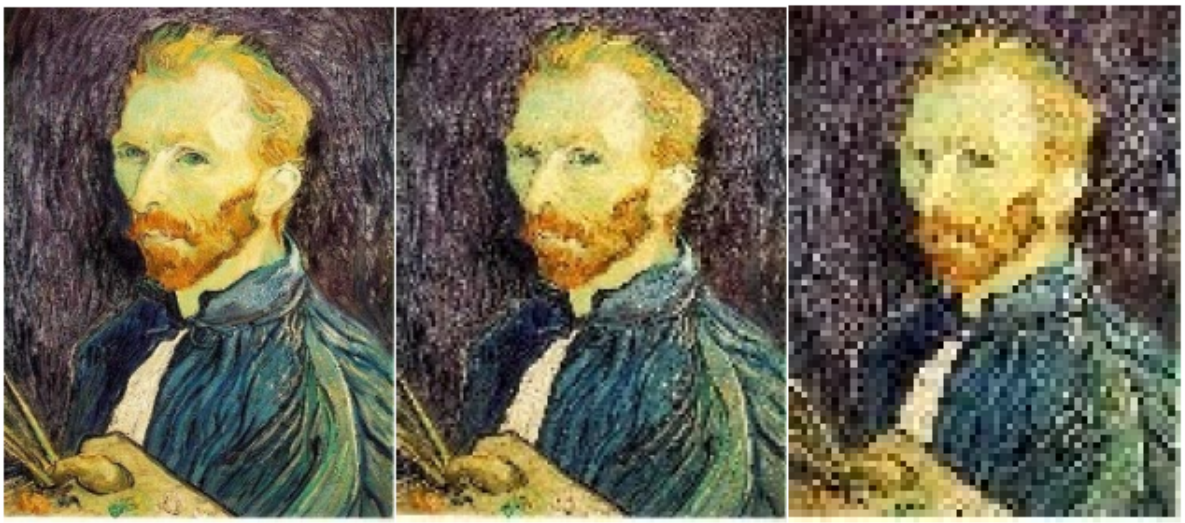
How to generate a reduced version of an image? With resizing, also called sub-sampling. A first naive approach could be, for instance, to throw away even rows and column



1/4

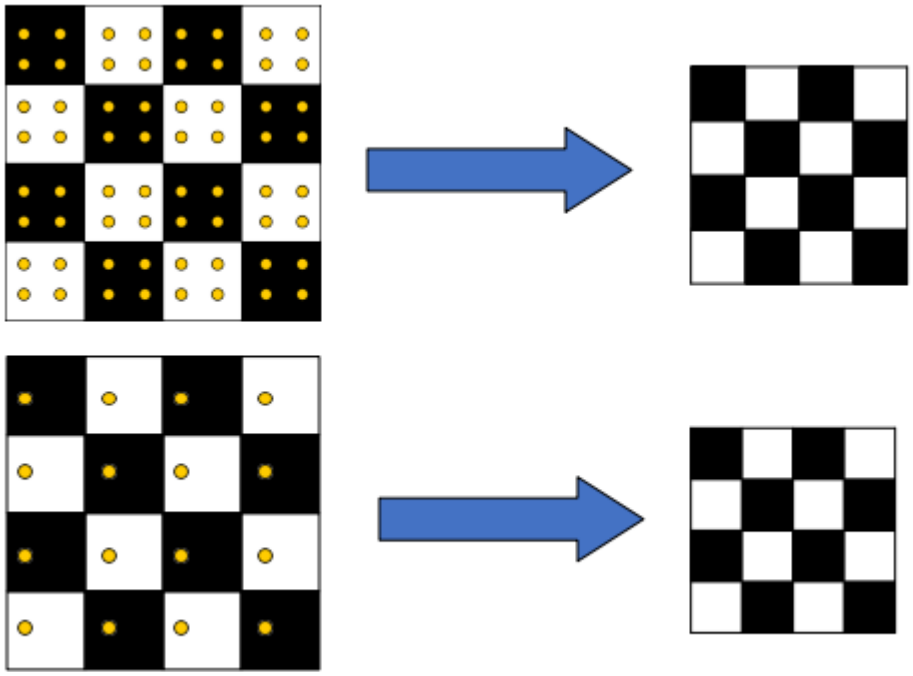
1/8

In this way I the image becomes ***aliased***, as you can see by zooming in the images:

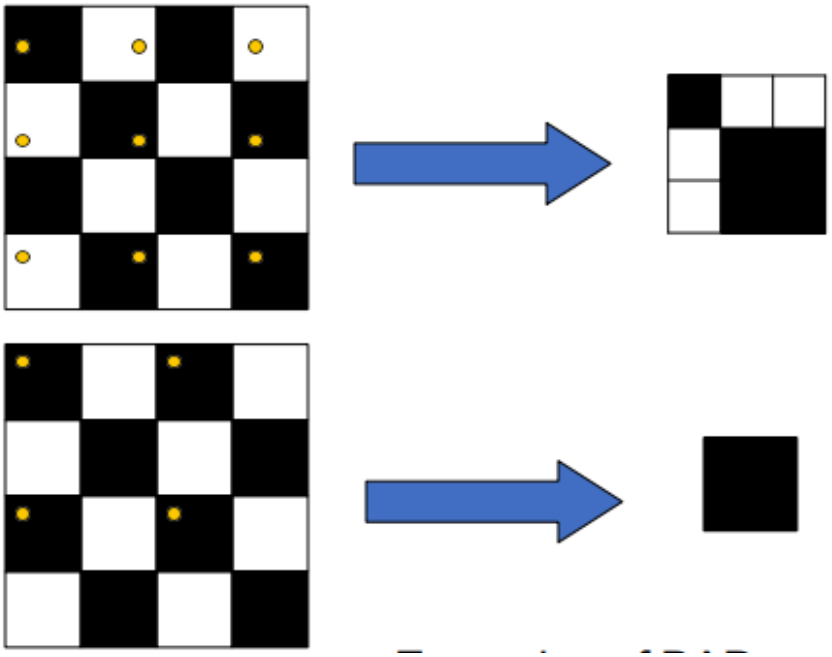


1/2 1/4 (2x zoom) 1/8 (4x zoom)

So the approach of removing even rows is not effective. We need to perform a good sampling, we want to keep some pixels instead of the other in a smart way, in order to maintain the content of the image.



This above is an example of good sampling, since we are maintaining the content of the image. This below is an example of bad sampling:



This is *undersampling*. As you can see, the image I obtained is completely different from the original one, since the rate of sampling, represented by the presence of yellow dots, is insufficient

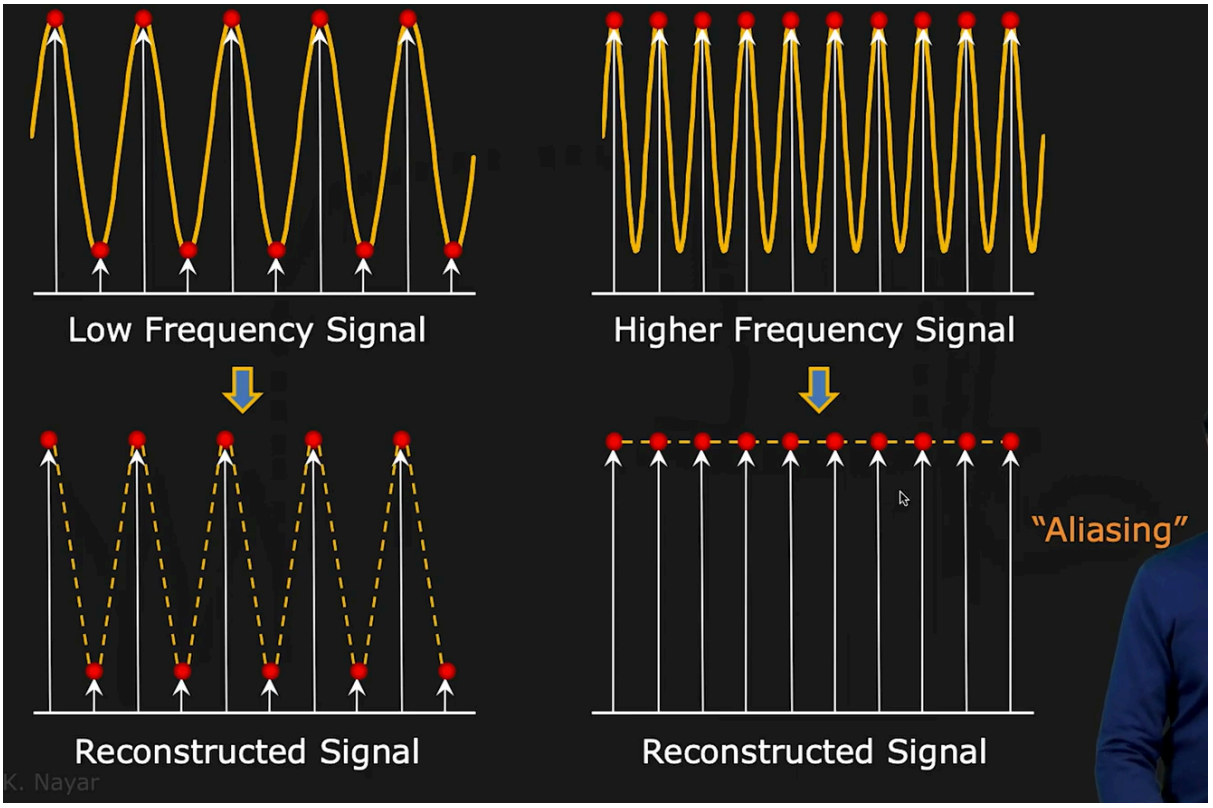
Nyquist Rate

The problem of aliasing occurs when the sampling rate is not high enough to capture the amount of detail in the image. In fact, the image is a 2D signal, and as all signals it has a frequency that measures how detailed the image is. In order to not have aliasing, the

sampling rate must be:

$$\text{sampling rate} \geq 2 * \text{max frequency in the image}$$

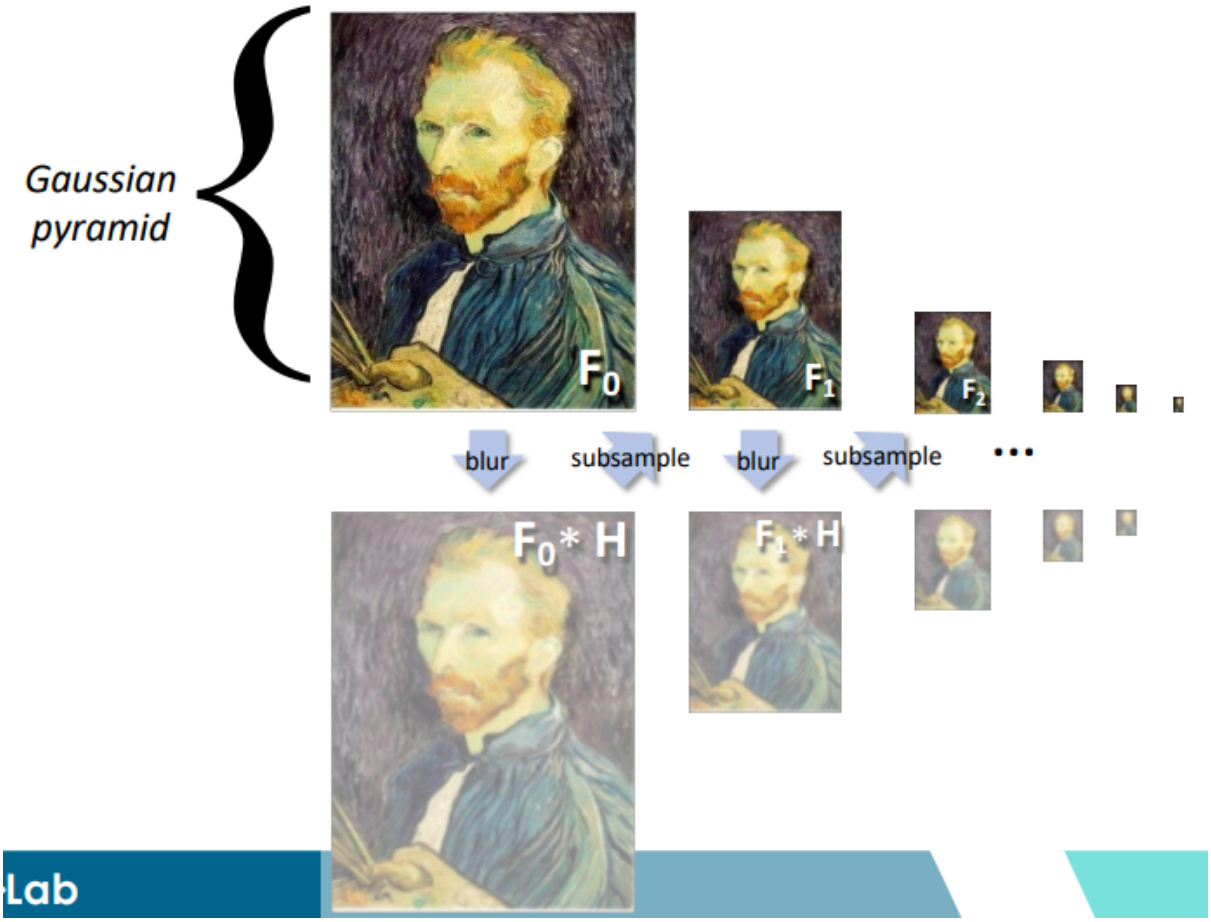
or in other words, at least 2 samples per cycle. This minimum sampling rate is called the Nyquist rate. So an image with high frequencies, meaning an image with high details, must have a higher sampling rate in order not to become aliased.



Sub-sampling creates problems in NN too: in the pooling operation, since it's a downsampling operation, the aliasing can happen. This creates problems in the context of classification stability. For solving this problem we can use an anti-aliasing CNN. So a module where the aliasing is taken in consideration and reduced.

Gaussian (Low-Pass) Pre-Filtering

This approach is better than the naive one. First we filter the image with a Gaussian, then we sub-sample. The gaussian filter size should double for each ½ size reduction.



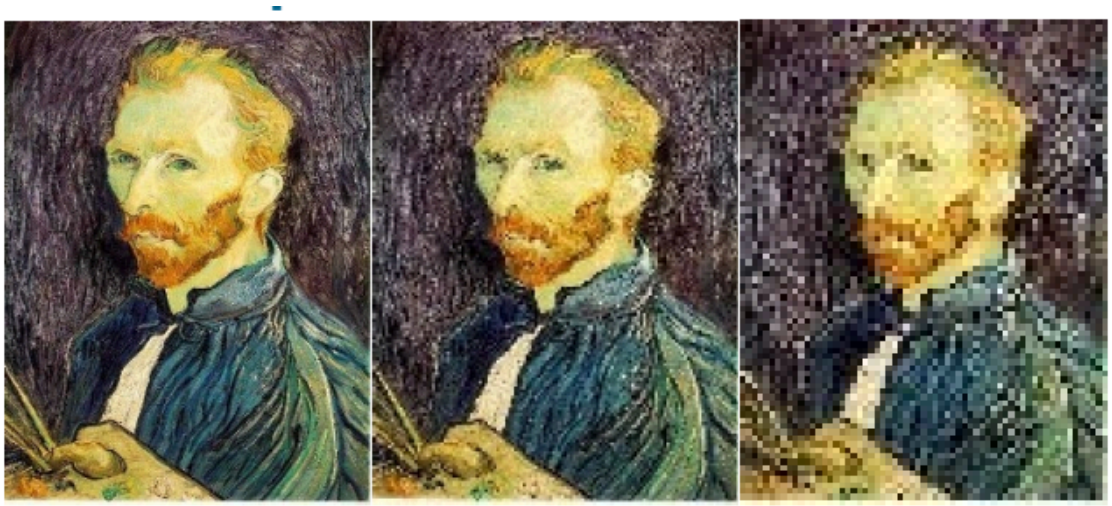
Lab

This is the result:

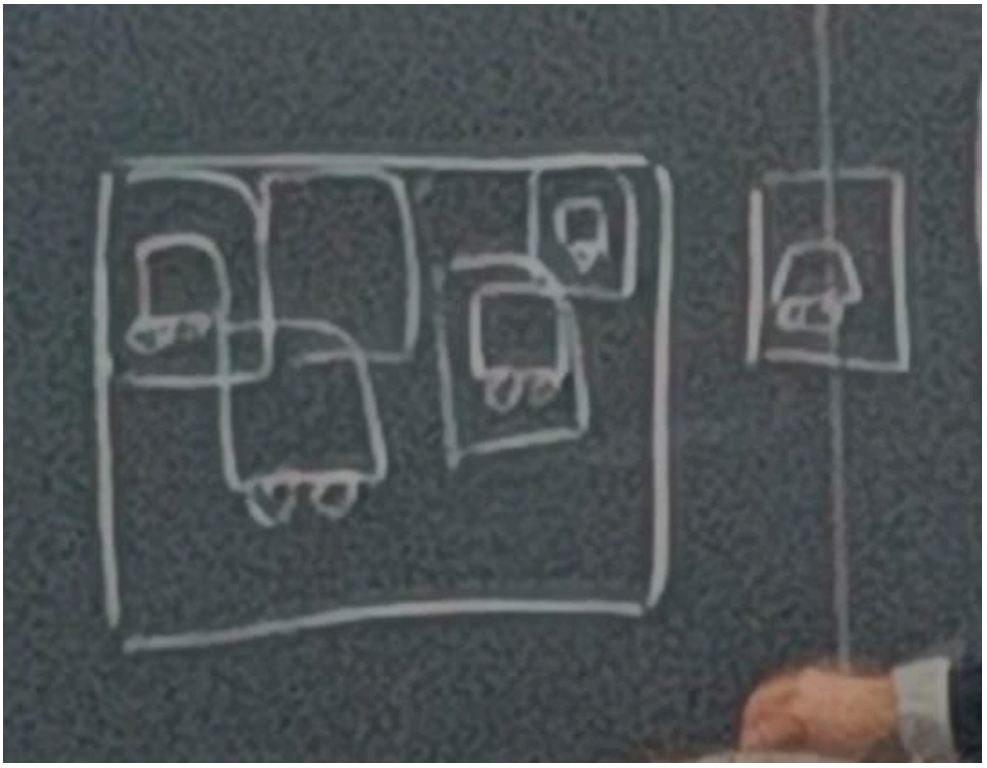
Gaussian 1/2 G 1/4 G 1/8

When I apply the gaussian filter the high-frequencies are removed. Namely, I remove the content from the image. This helps in the following subsampling.

Comparing it with the naive approach:



Subsampling is useful to improve search: e.g. template matching. For instance, there is an image with a car. I correlate the patch of the car with the image to see if I find the car. Initially I start from the original dimension, but what if the car is smaller? I reduce the image and I reapply the patch to find the car with this reduced size.



Upsampling

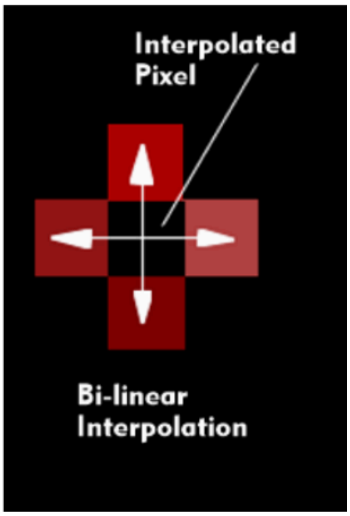
Increasing the size of the image.

Nearest Neighbor Interpolation

to make it 10 times bigger the naive approach is to repeat each row and column 10 times.

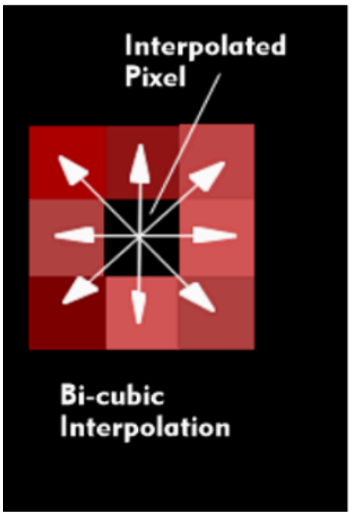
Bilinear Interpolation

The bilinear interpolation uses a weighted average of the four nearest pixel values to estimate the value of the new pixel. It produces a smoother image compared to NN interpolation.



Bicubic Interpolation

Uses cubic functions on the surrounding pixels in order to compute the pixel of interest. In each pixel interpolation it considers more neighboring pixels than the bilinear interpolation. It results in a smoother result than the one of bilinear interpolation, but it’s more computationally intensive.



Comparison between the three approaches:

Another approach is to use multiple images of the scene you want to upsample with small shifts, averaging then the information, getting a single photo. For instance our cell phone can capture a burst of photos to do super-resolution of the photo. In fact, when we shoot a photo we move naturally a little bit, and so we acquire photos that are very similar but not equal.

Image Derivatives

If we want to extract the content of the image, the edges are an important part to take into account. The concept behind edge detection is to compute the partial derivative of an image

Image is a function f(x,y)

Remember:

$$\frac{\partial f(x,y)}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon, y) - f(x,y)}{\epsilon}$$

Approximate:

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x+1,y) - f(x,y)}{1}$$

-1	1

Another one:

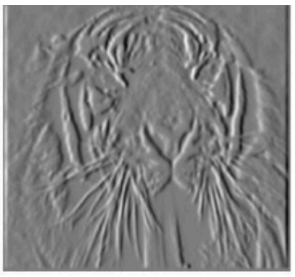
$$\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x+1,y) - f(x-1,y)}{2}$$

-1	0	1

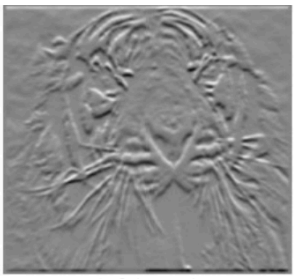
The smallest movement you can do in an image is 1, so I can substitute epsilon with 1.

Gradient Magnitude

Now, having this [-1, 1] for the derivative of x and $[-1, 1]^T$ for the derivative of y, we can get the partial derivative of f w.r.t. to x and y. Then, combining the two, we can get the gradient magnitude of f. As you can see the gradient is stronger on the edges:



$\frac{\partial f(x,y)}{\partial x}$



$\frac{\partial f(x,y)}{\partial y}$

-1	1

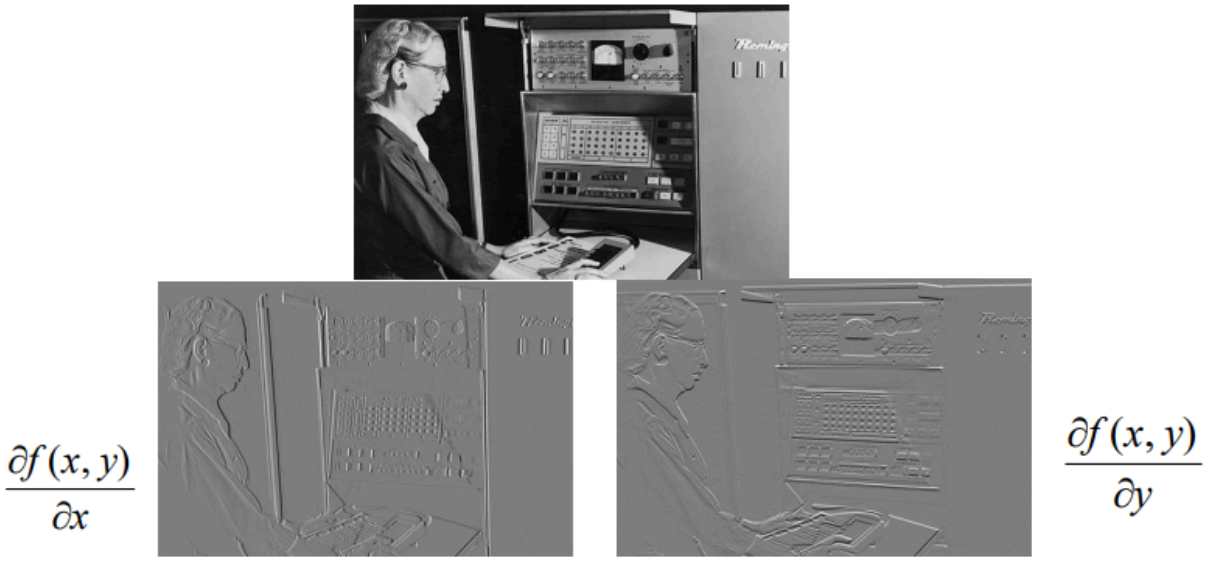
-1
1

or

1
-1

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

The partial derivative of x evidences the vertical edges, while the derivative of y the horizontal one. Another example



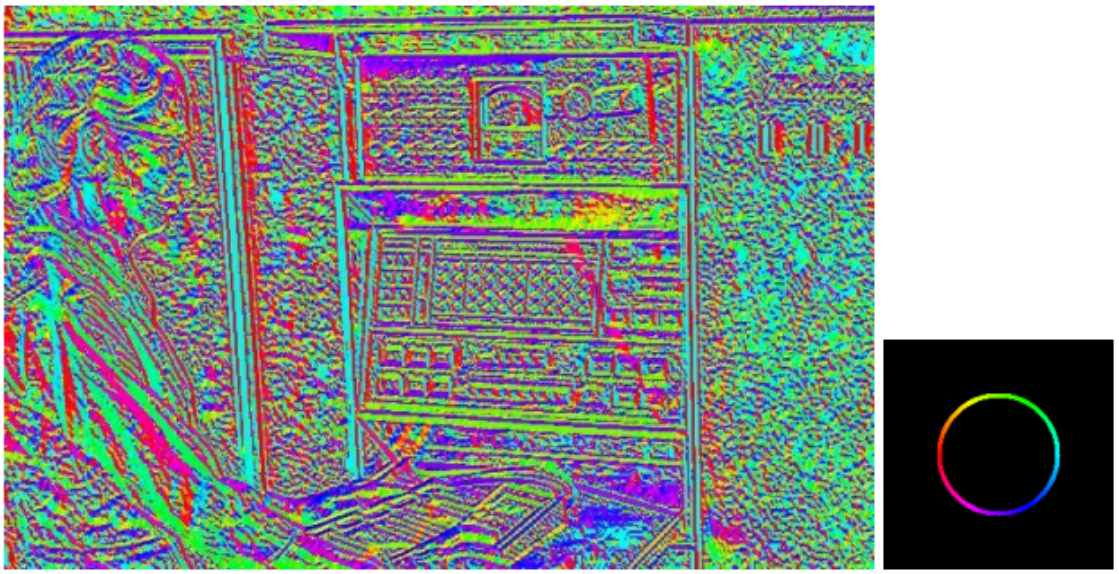
The gradient magnitude gives the *edge strength*:

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

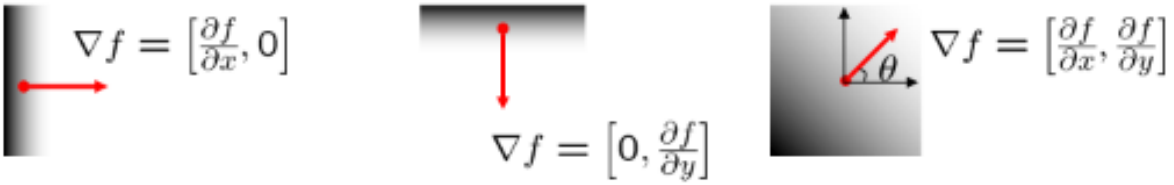
Gradient Orientation

We can also visualize the gradient orientation computing the following formula for each pixel. This will give the direction of the gradient for each pixel:

$$\theta = \tan^{-1}\left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x}\right)$$



We know that the gradient points in the direction of the most rapid increase in intensity. That means that in a uniform part of the image, where the intensity is almost the same in all the neighborhood pixels, the derivative is almost 0 in all the directions.

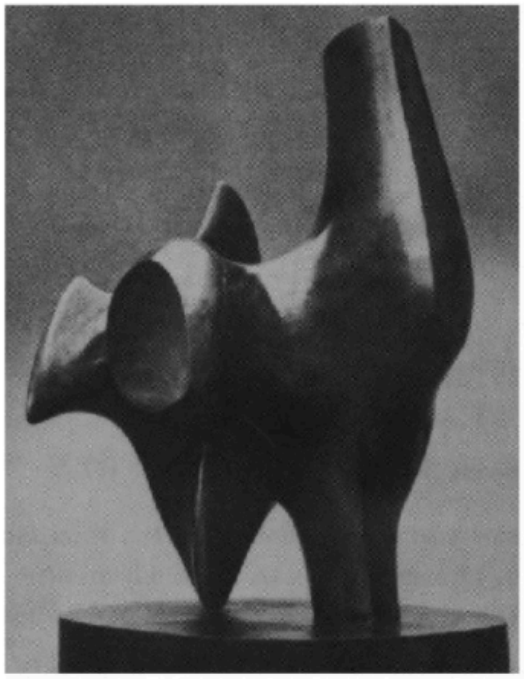


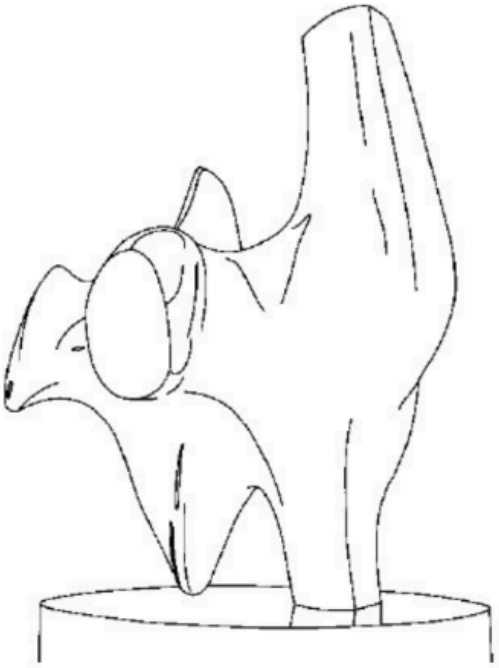
These above are some examples of gradients computed in some points: if the gradient is [df/dx, 0] we have an horizontal edge, while if the gradient is [0, df/dy] is a vertical edge.

Edge Detection

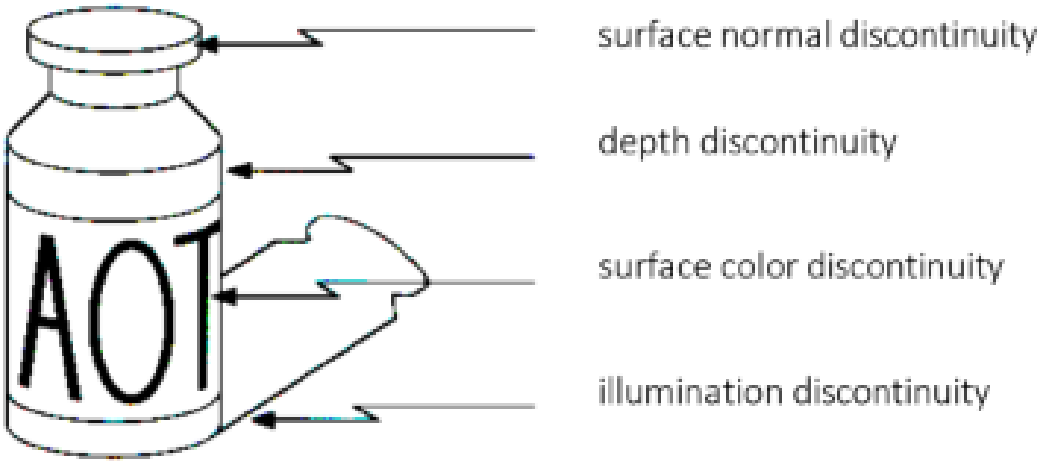
To convert an image into a set of curves. Advantages:

- extracts the salient feature of the scene
- more compact that the image with all the pixels

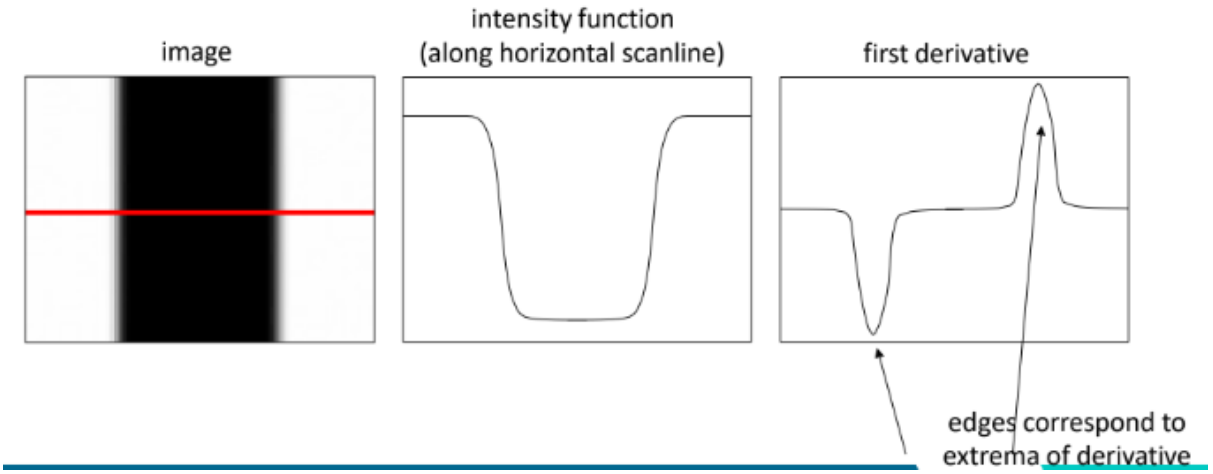




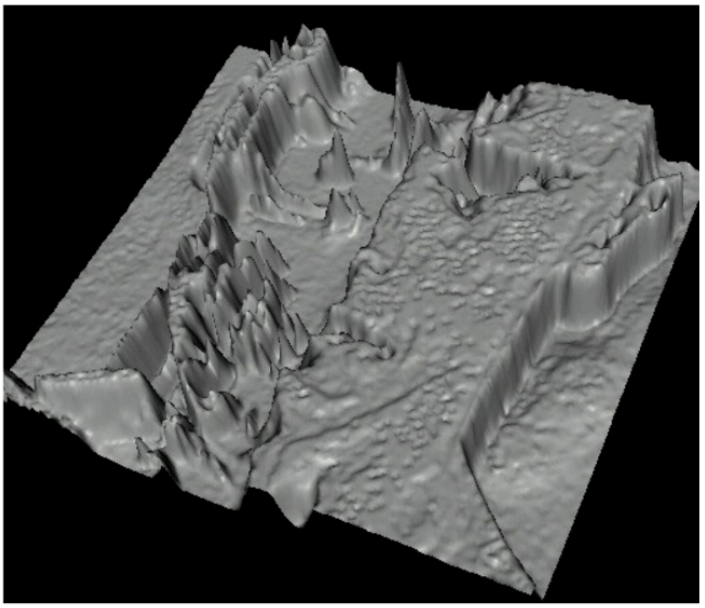
Edges are caused by a variety of factors:



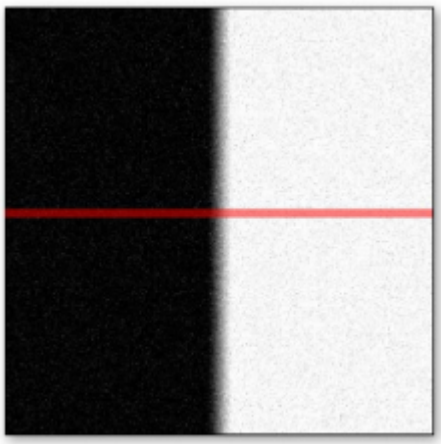
And they are basically place of rapid change in the image intensity function:



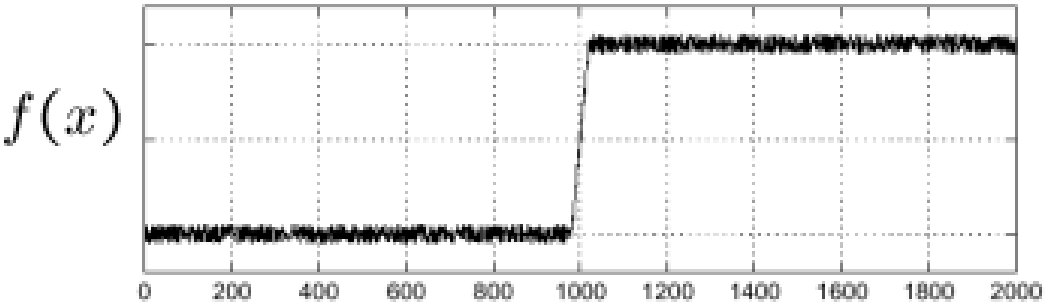
In fact, visualizing the image as a function, we can see that edges look like steep cliffs. For instance:



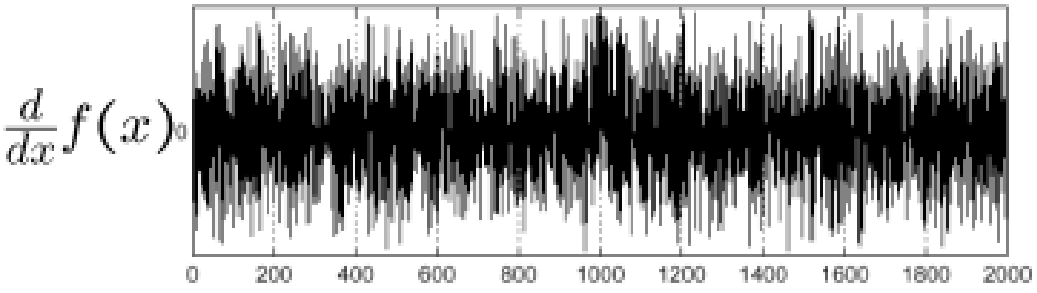
So, to detect edges, we just compute the derivatives, as in the previous section. The problem is that there is no image without noise. For example:



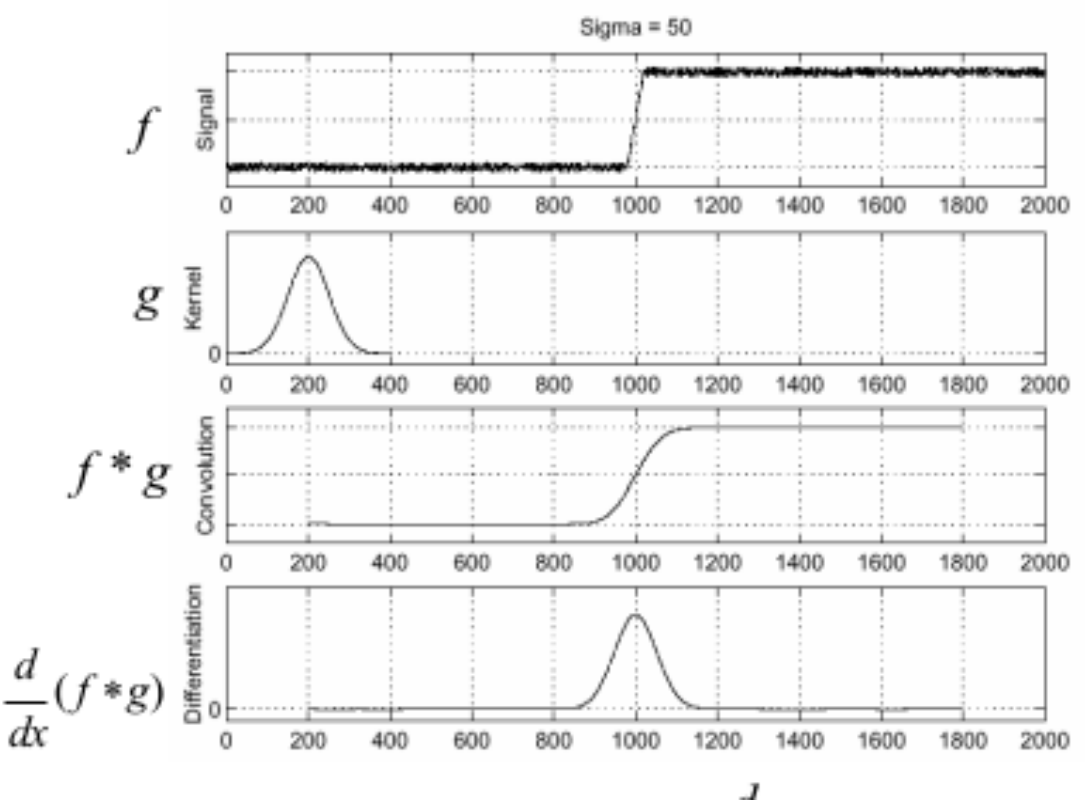
let's consider only the row of pixels indicated by the red line. We can plot the intensity of these pixels as a function f of position x. So f(x) is basically a signal.



The problem is that it is a noisy signal, so the derivative function is not detecting the edge properly.

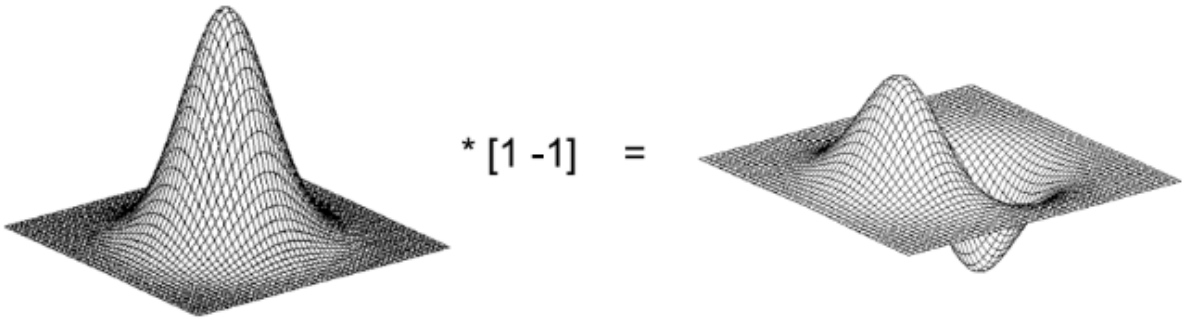


How to solve this problem and detect the edge clearly? We apply the gaussian filter:



applying the gaussian kernel you obtain a blurred, smoothed version of the image that has no noise variation. At this point if you compute the derivative the edge is easily detected.

For the derivative theorem of convolution, instead of doing the convolution and then the derivative, you can just convolute f with the derivative of the gaussian, that is a well-known function:



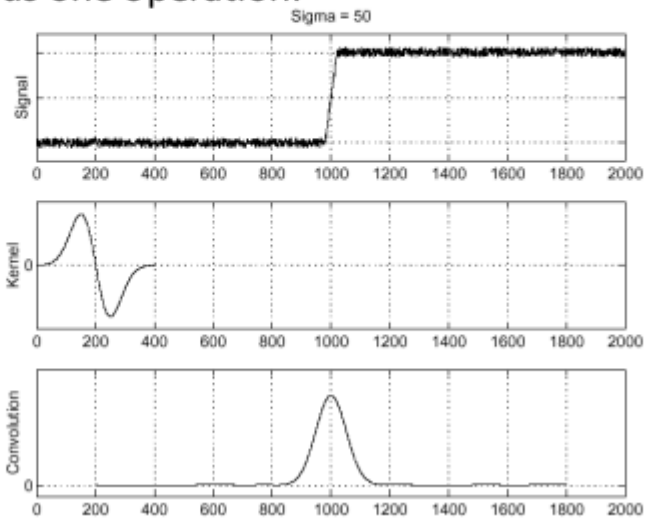
In this way we do one operation left:

$$\frac{\partial}{\partial x}(h \star f) = (\frac{\partial}{\partial x}h) \star f$$

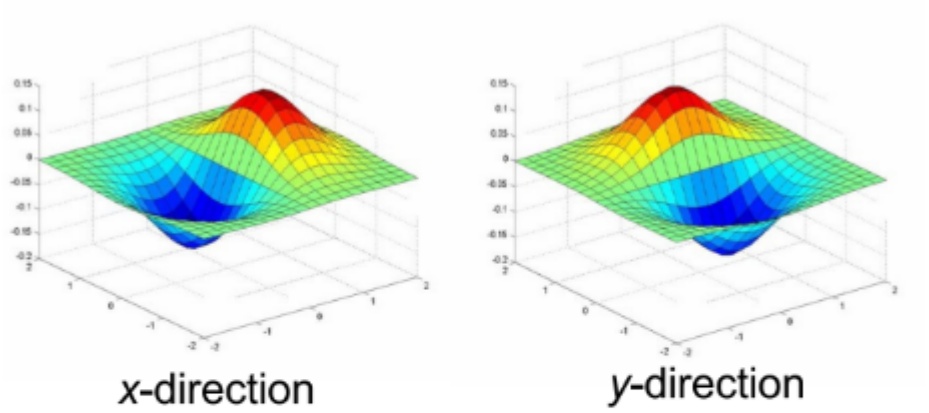
- This saves us one operation:

$\frac{\partial}{\partial x}h$

$(\frac{\partial}{\partial x}h) \star f$



That is for the x direction, but we can do that also for y of course. The derivative of the gaussian filter w.r.t to x and y are this one:



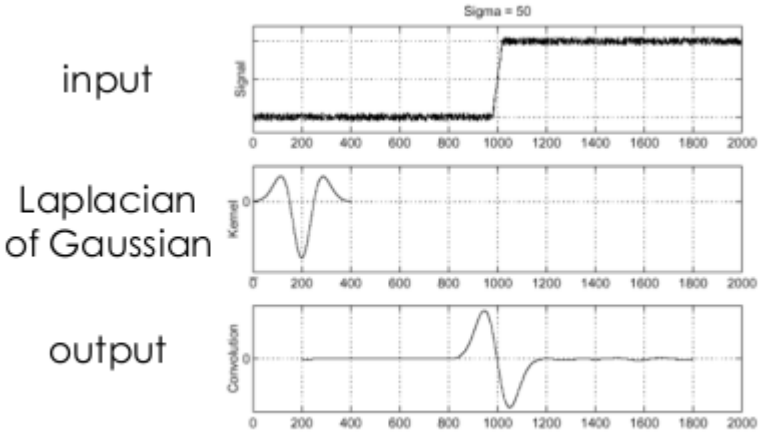
Laplace Filter as an Alternative

One of the most famous second derivative filters. Useful because it is easier to understand when a function is 0 than to find its maximum.

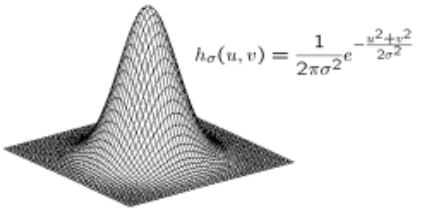
second-order finite difference $f''(x) = \lim_{h \to 0} \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$ → Laplace filter

1	-2	1

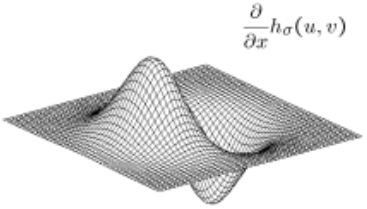
Laplacian of Gaussian (LoG):



where the laplacian of Gaussian is the second derivative of the gaussian. Since it's a second derivative, the zero crossing tells us where the edge is.



Gaussian



Derivative of Gaussian

Laplacian of Gaussian

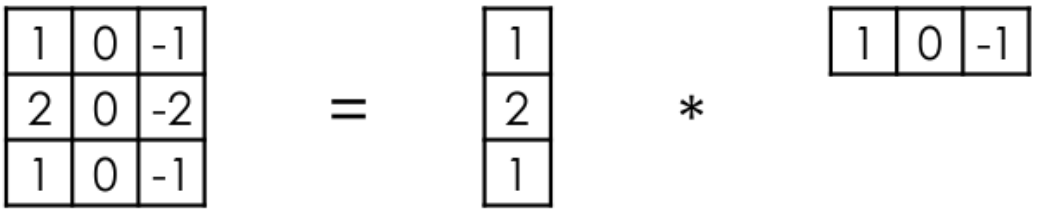
Sobel Operator as Approximation of Derivative of Gaussian

The Sobel operator is a common approximation of the Derivative of Gaussian

$$\frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

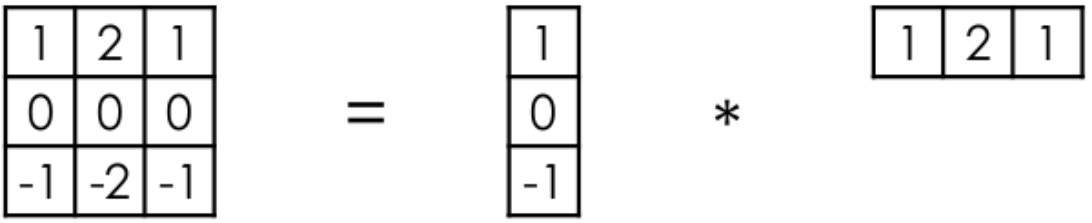
s_x s_y

The Sobel filter is separable. This is the horizontal:



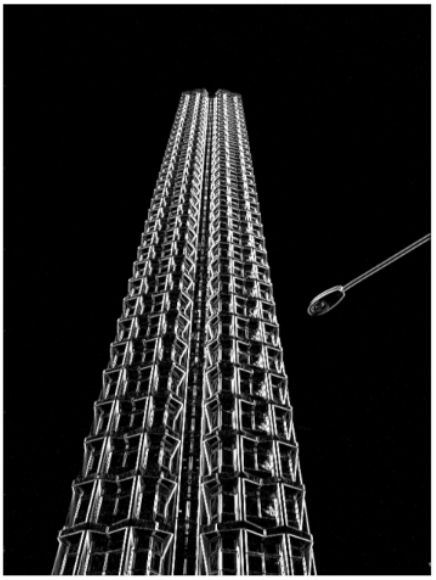
Sobel filter Blurring 1D derivative filter

While this is the vertical:

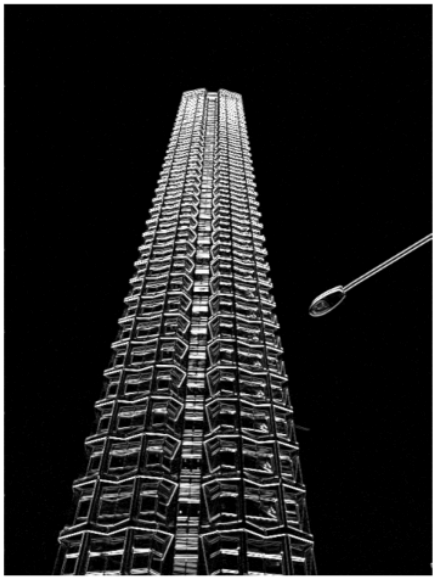


The horizontal Sobel filter extracts vertical edges, but it's called horizontal because the 1D derivative filter, once the filter is separated, is the horizontal component, namely the row. The opposite holds for the vertical.

original



horizontal Sobel filter



vertical Sobel filter

Other derivative filters, beside Sobel:

Scharr

3	0	-3
10	0	-10
3	0	-3

3	10	3
0	0	0
-3	-10	-3

Prewitt

1	0	-1
1	0	-1
1	0	-1

1	1	1
0	0	0
-1	-1	-1

Roberts

0	1
-1	0

1	0
0	-1

Canny Edge Detector

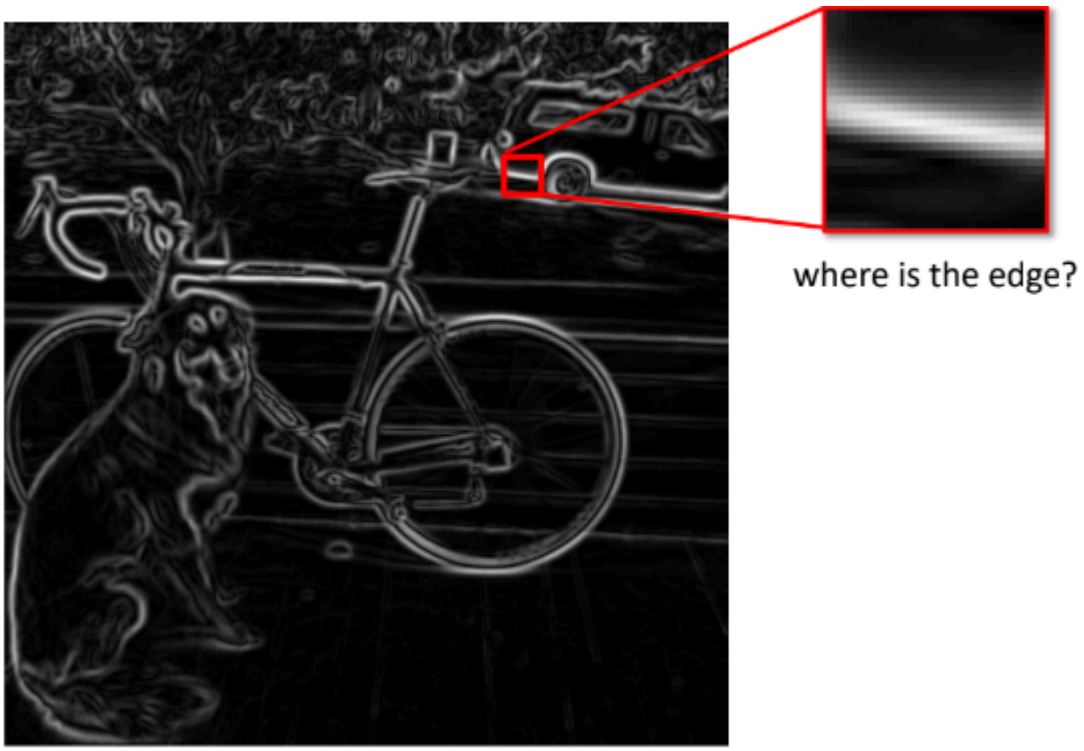
The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It provides:

1. Accuracy
2. Minimizing FP and FN
3. Precise localization: pinpointing edges where they actually are
4. single response: only one edge is found where there only is one edge

The algorithm has the following steps:

original image

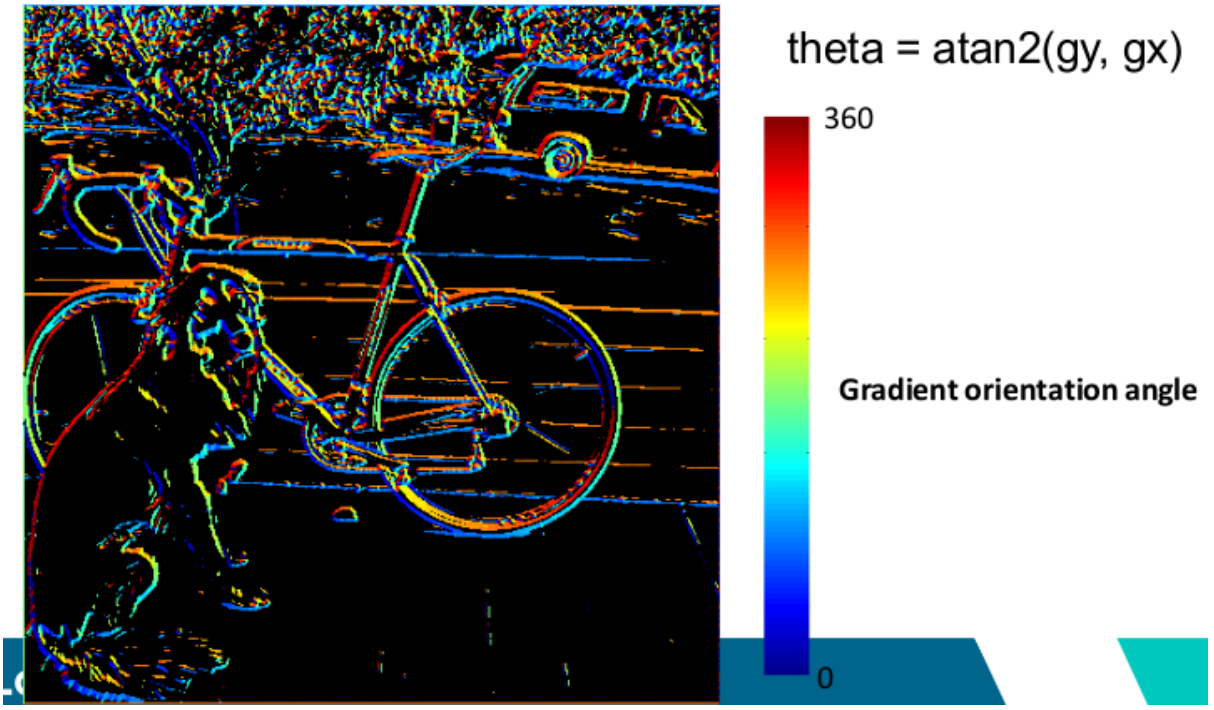
1. Compute the Gradient of the smoothed image, for instance using the Sobel filter. This below is the smoothed gradient magnitude



smoothed gradient magnitude

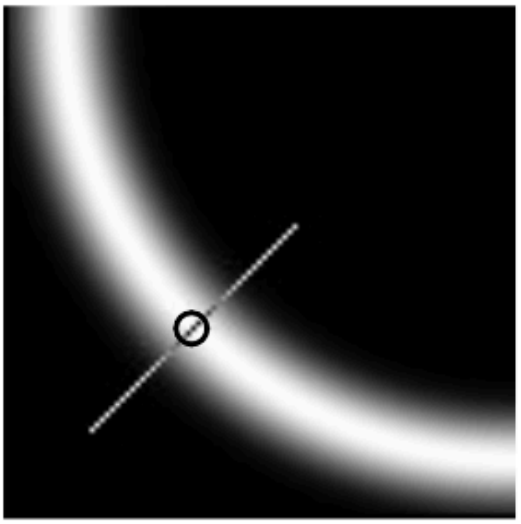
Inspecting the smoothed gradient magnitude, the exact edge location can be ambiguous. To increase the precision of the edge detection we do non-maximum suppression. Explained in the following steps.

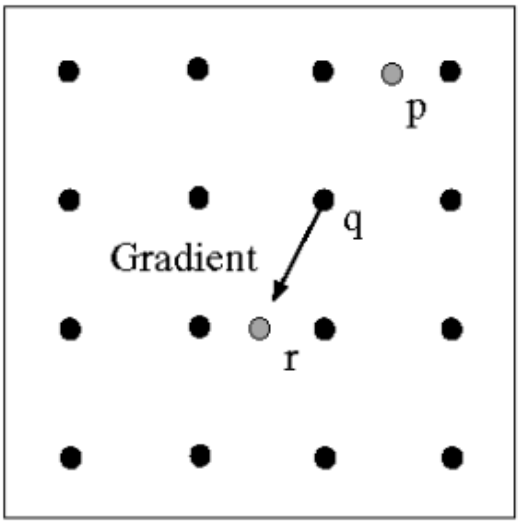
2. Get Orientation at Each pixel, from the smoothed gradient orientation

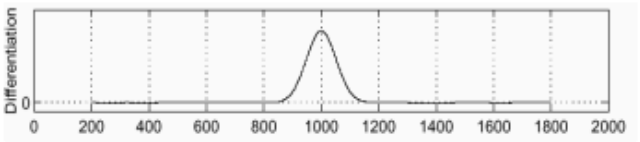


Each pixel will now have a gradient orientation.

3. Non-maximum suppression: suppress the pixels that are not local maximum in order to find clearly where the edge is. For each pixel in the edge, we check if the pixel is the local minimum along the gradient direction

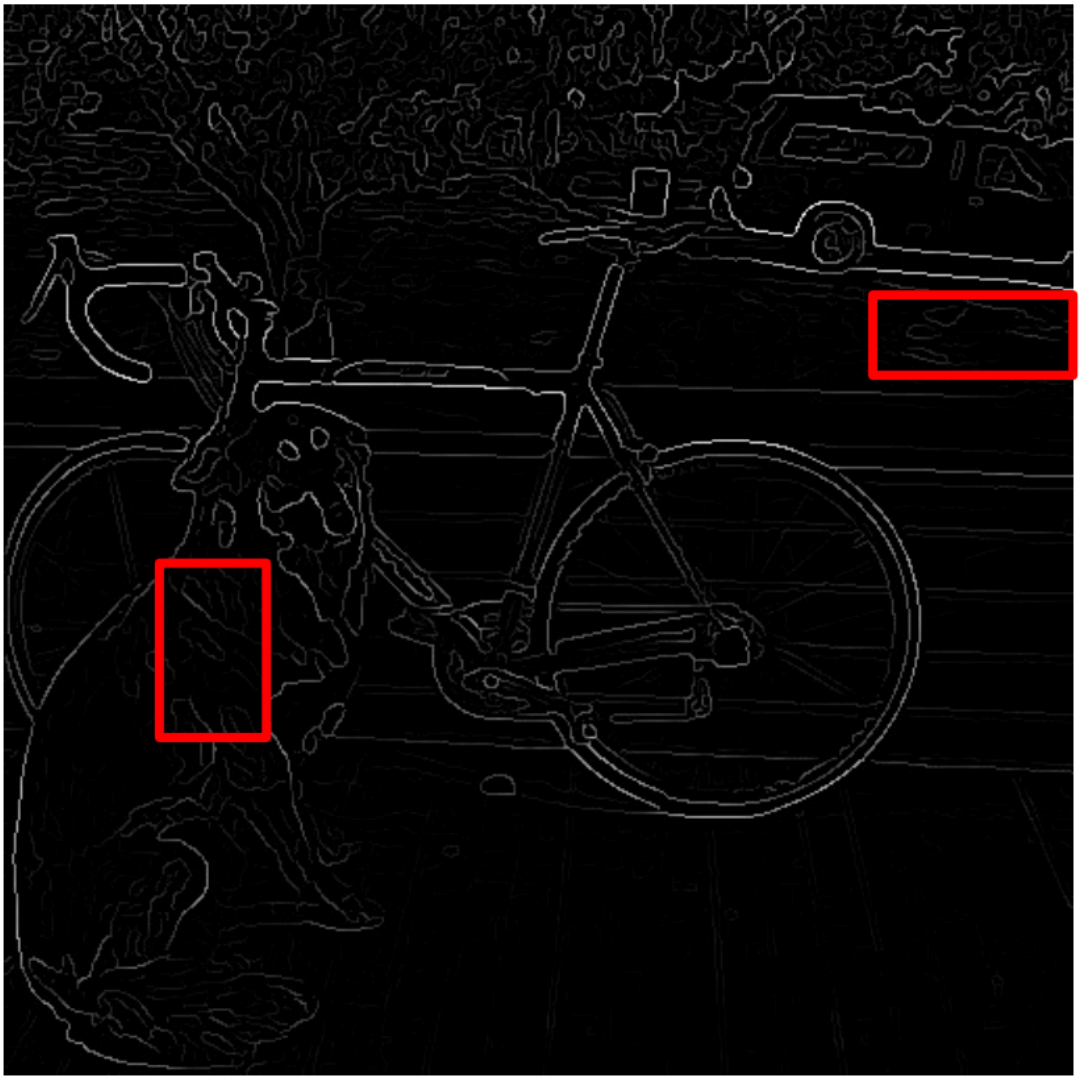






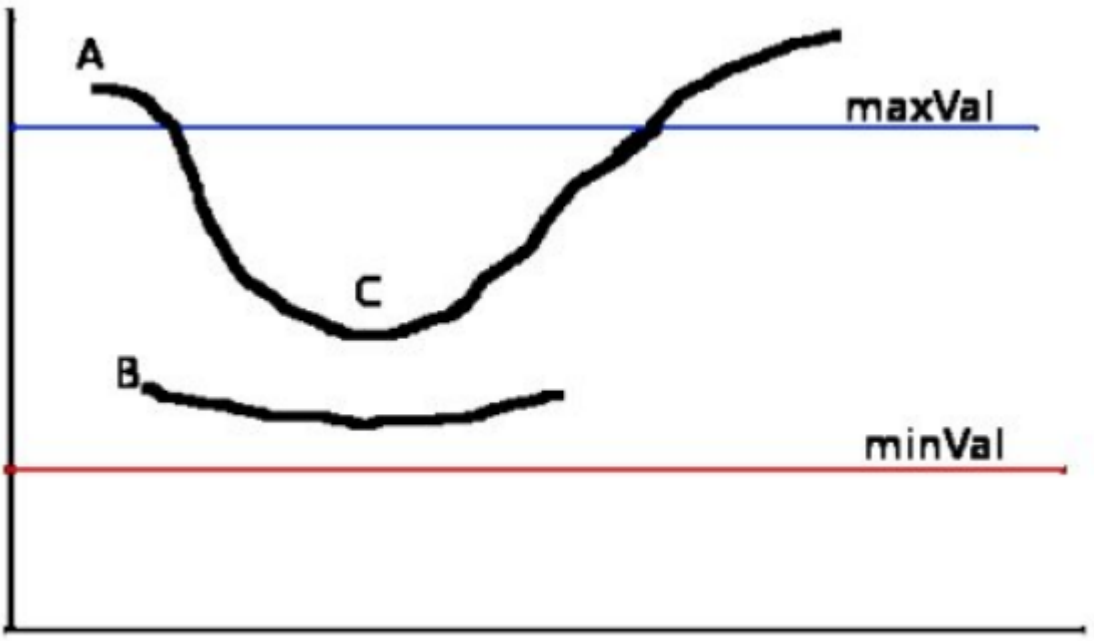
After non-maximum suppression:

4. Hysteresis Thresholding: even after non-max suppression, the noise still exists:



We want stronger edges in order to have a more precise contour of the object. To obtain it, given the intensity gradient R in the edge pixel considered, and T and t the two thresholds:

- If R > T: the edge is strong
- If R < T but R > t: the edge is weak
- R < t: no edge



Strong edges are always edges, while weak edges are edges iff they connect to a strong edge in the neighborhood. For instance, in the curves above, edge A is a strong edge, since R > T. Edge C is a weak edge, since T > R > t, but since it is connected to A that is strong, it is a valid edge. B on the other end, is weak and not connected to a strong edge, so it's discarded. We can say that weak edges are transformed into strong ones only iff at least one of the neighbors is a strong edge.

Parameters of the Canny Edge Detector

The parameters of this algorithm:

- The width of the Gaussian blur σ applied at point 1. Large σ detects large-scale edges, while small σ detects fine edges.

original | Canny with $\sigma = 1$ | Canny with $\sigma = 2$

- The thresholds T and t in the hysteresis thresholding at point 4.

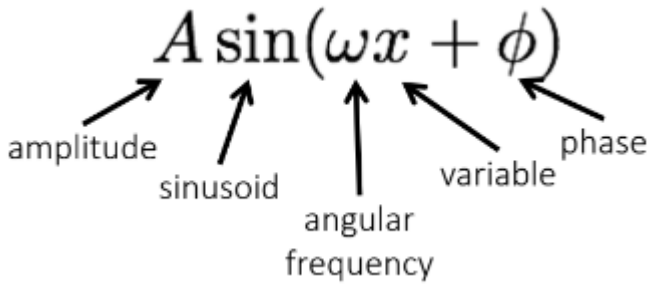
Fourier Analysis

6/03/2024

The Building Block of every Periodic Function

A periodic function can be written as the sum of sines and cosines with different amplitudes and phases. Low frequencies in an image correspond to smooth regions, since there the intensity is slowly varying, while high frequency corresponds to edges or noise: regions in which there is a rapid change in intensity. Working in the frequency domain allows for the manipulation of the frequency components of an image, enabling various types of filtering that are often more efficient or effective than direct spatial domain operations.

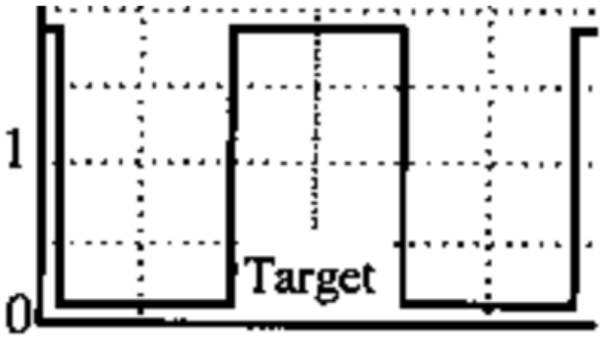
This is the building block of every periodic function:



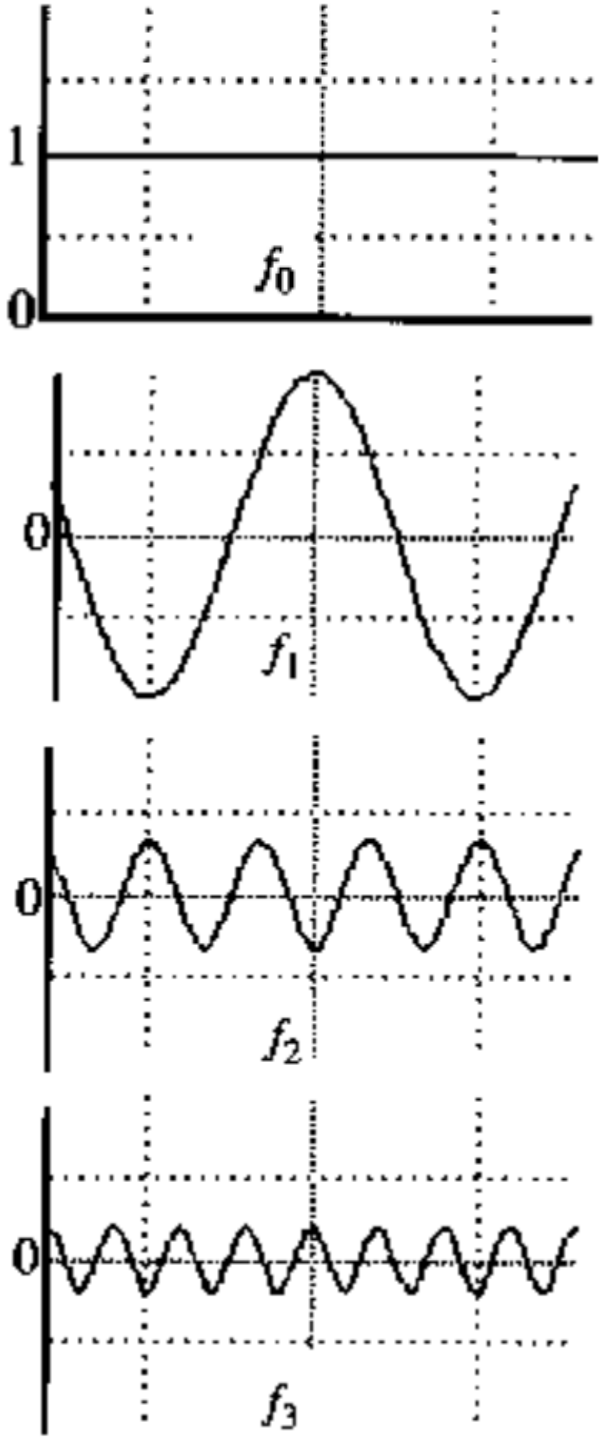
A is the intensity, w is the frequency, phi is the phase. The amplitude indicates the strength of the frequency components, while the phase information about the position of the frequency components.

Adding these basic building blocks you can get any periodic signal you want.

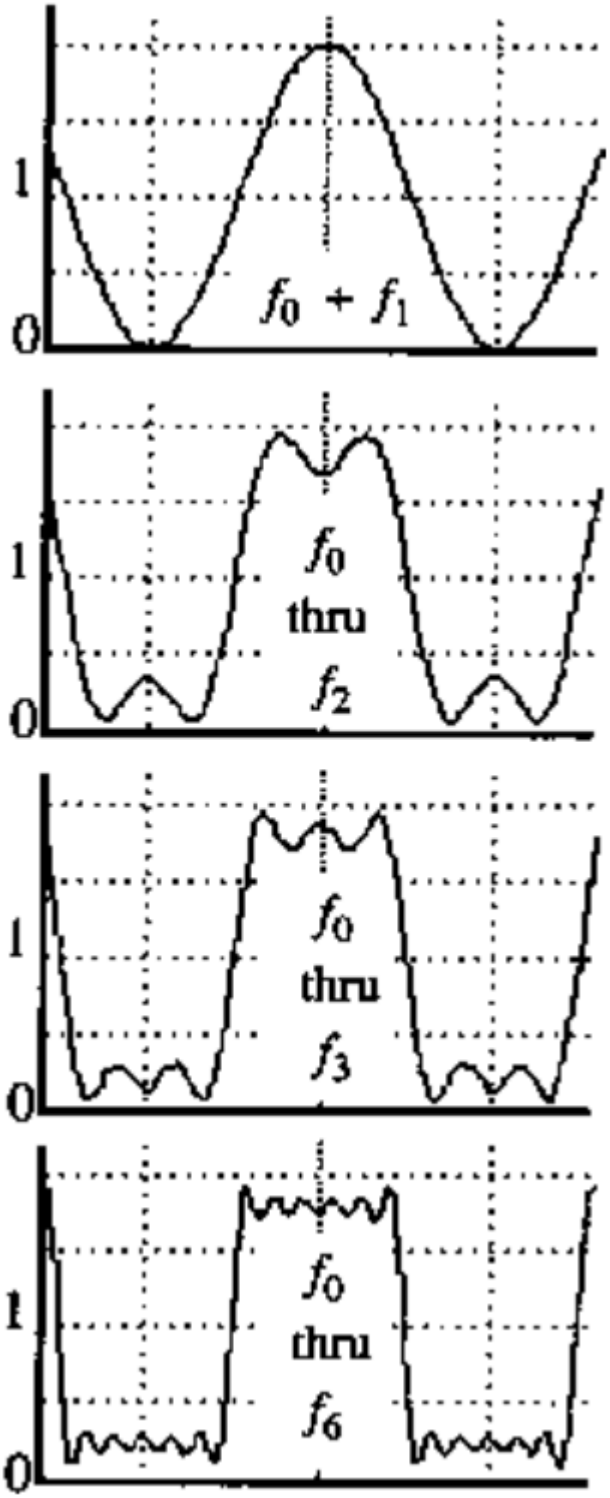
Let’s do an example. This is the target function we want to approximate:



We get it by summing $f_0 + f_1 + f_3 + \ldots$

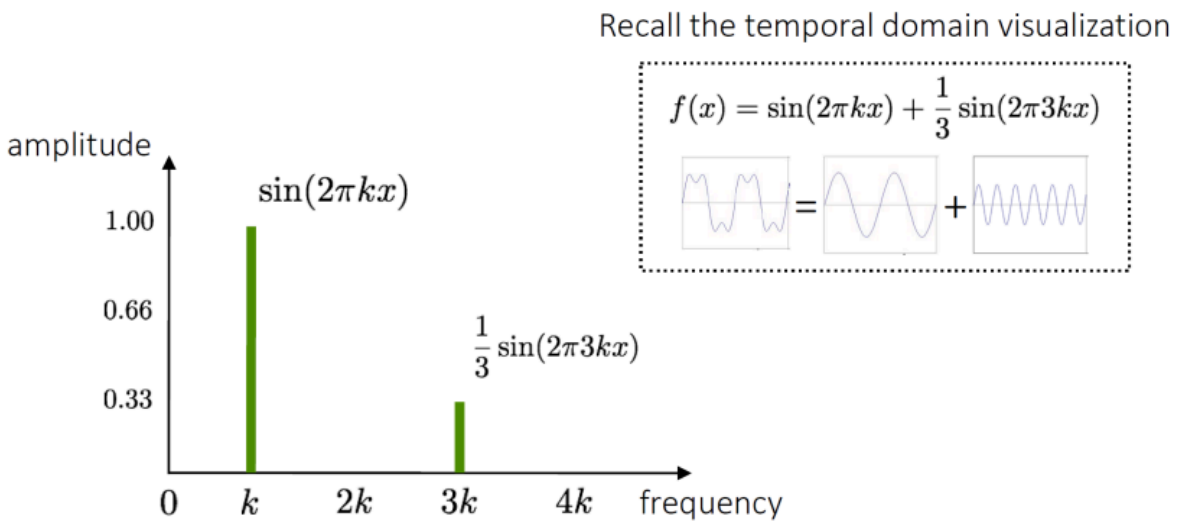


Summing more and more function we get closer and closer to the target:



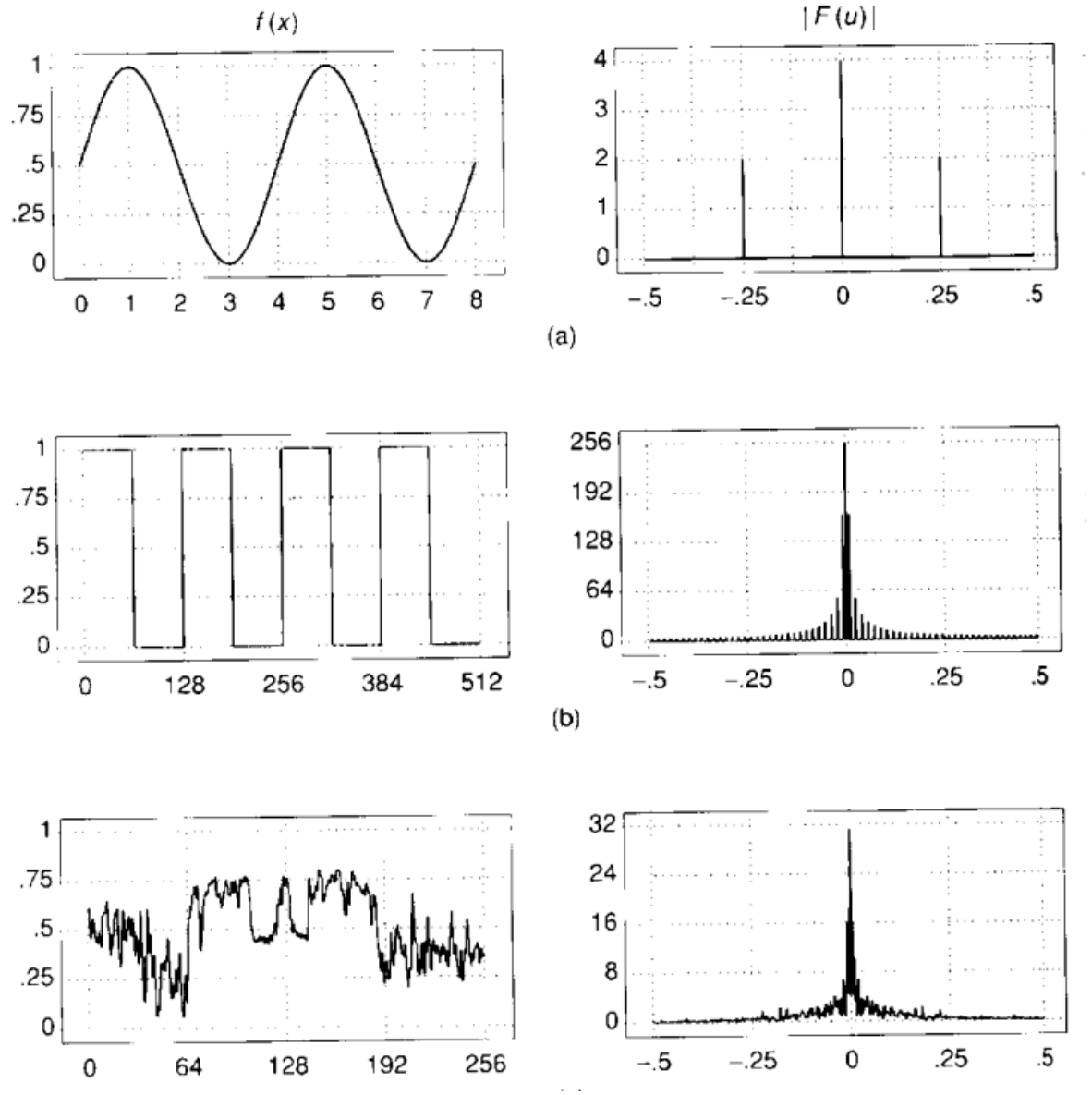
Frequency Spectrum

Given a signal in the time (spatial) domain, you can visualize its frequency spectrum. For instance:

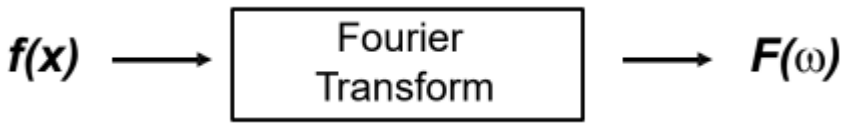


Note that only the positive part is shown, the negative part is symmetric. Moreover, note that the bar in 0 is the signal average. Since it's zero in this case, it means that this sine wave has no offset.

Other examples of frequency spectrums:

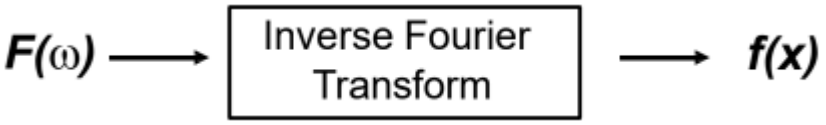


To see the frequency spectrum, we need to go from the x (spatial) domain to the w (frequency) domain. We do that with the Fourier Transform:



$$\text{Fourier Transform}: \quad F(\omega) = \int_{-\infty}^{+\infty} f(x) e^{-i\omega x} dx$$

That is also invertible:

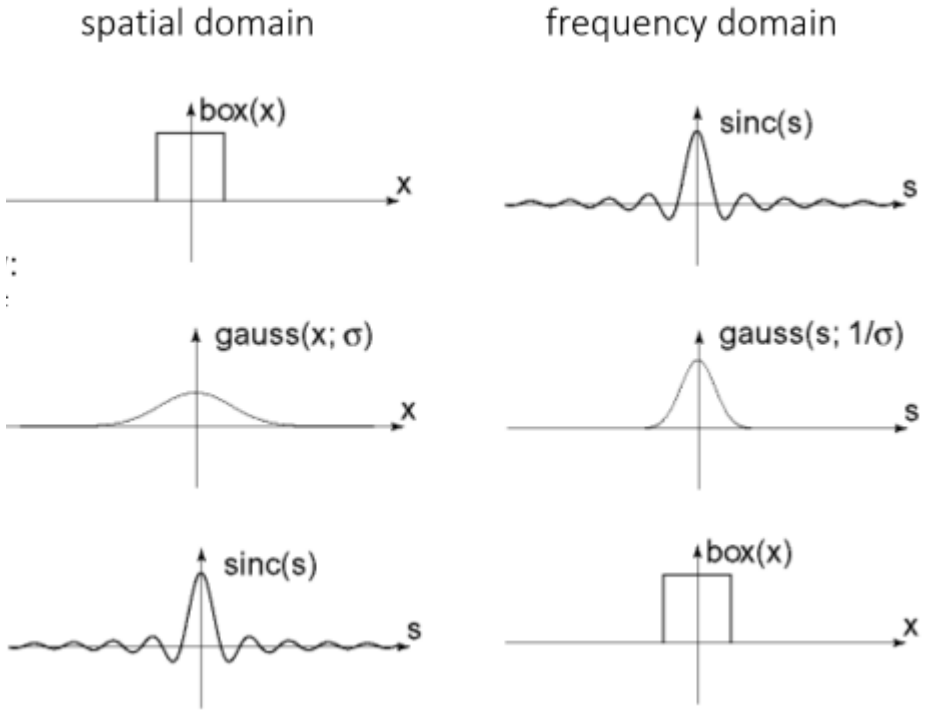


$$\text{Inverse Fourier Transform}: \quad f(x) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} F(\omega)e^{i\omega x}d\omega$$

having that

$$e^{i\omega x} = \cos(\omega x) + i\sin(\omega x)$$

Other examples of signals in time (space) and they corresponding frequency domain:



Note that the third row it's actually sinc(x) -> box(s)

Discrete Fourier Transform (DFT)

We use the discrete form since digital images are discrete. How to compute the discrete Fourier transform:

$$F(k) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{-j2\pi kx/N}$$

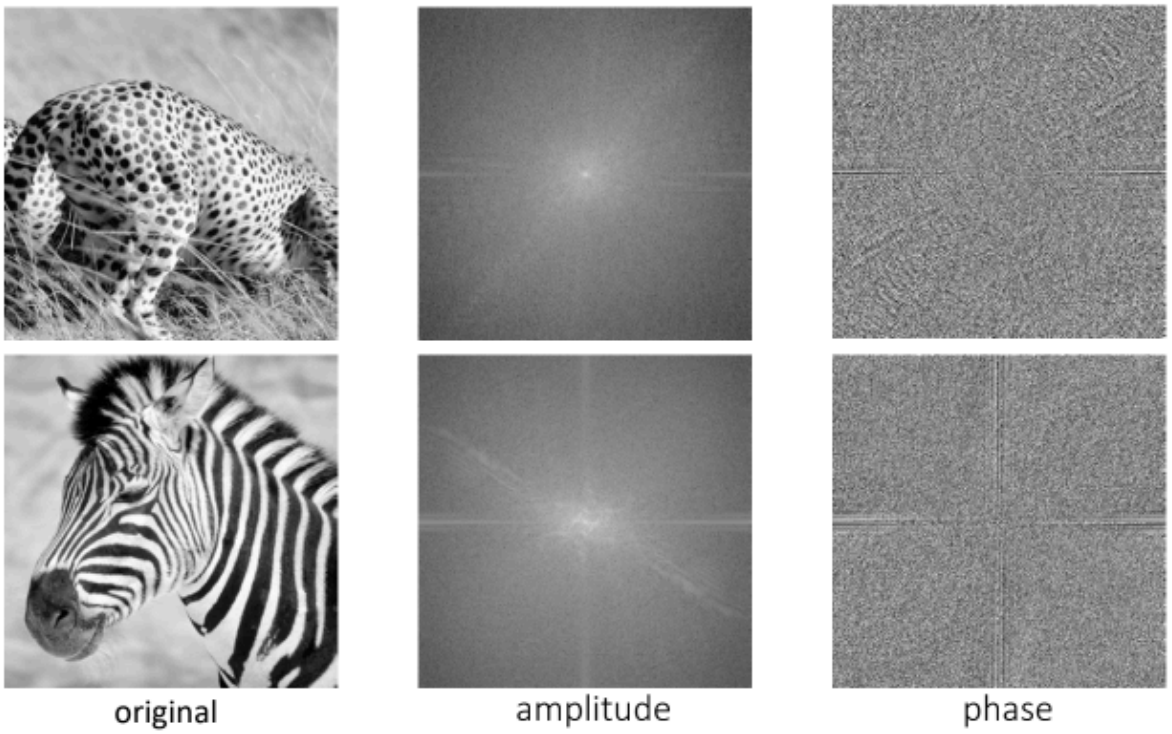
We can write it as a matrix multiplication:

$$\boldsymbol{F} = \boldsymbol{W}\boldsymbol{f}$$

In practice, this calculation is done using the Fast Fourier Transform (FFT), to compute it rapidly.

Fourier Analysis in 2D

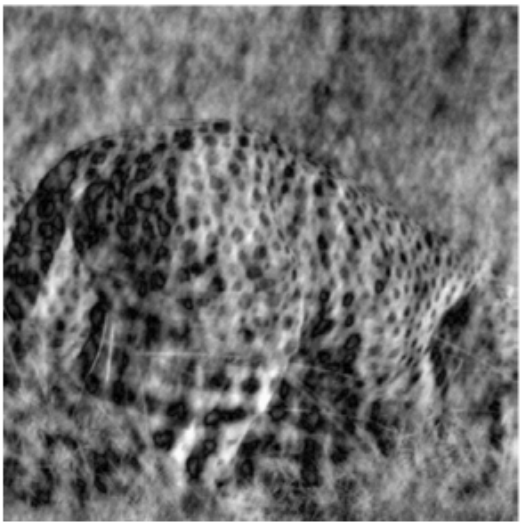
Fourier transform of natural images:



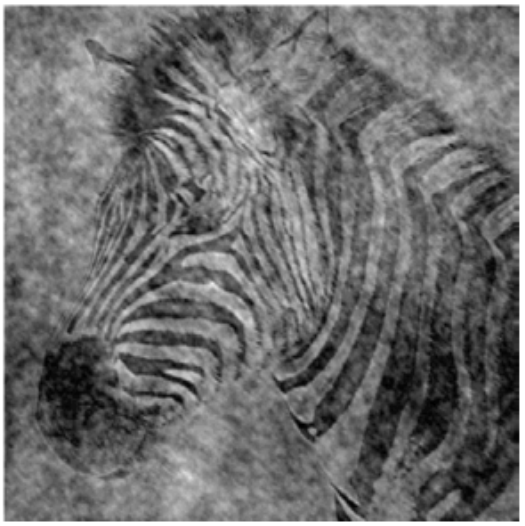
Amplitude provides information about the strength of different spatial frequencies in the image, while phase provides the structural information.

Importance of the Phase

The phase is the most informative one. Indeed, if we invert amplitude and phase:

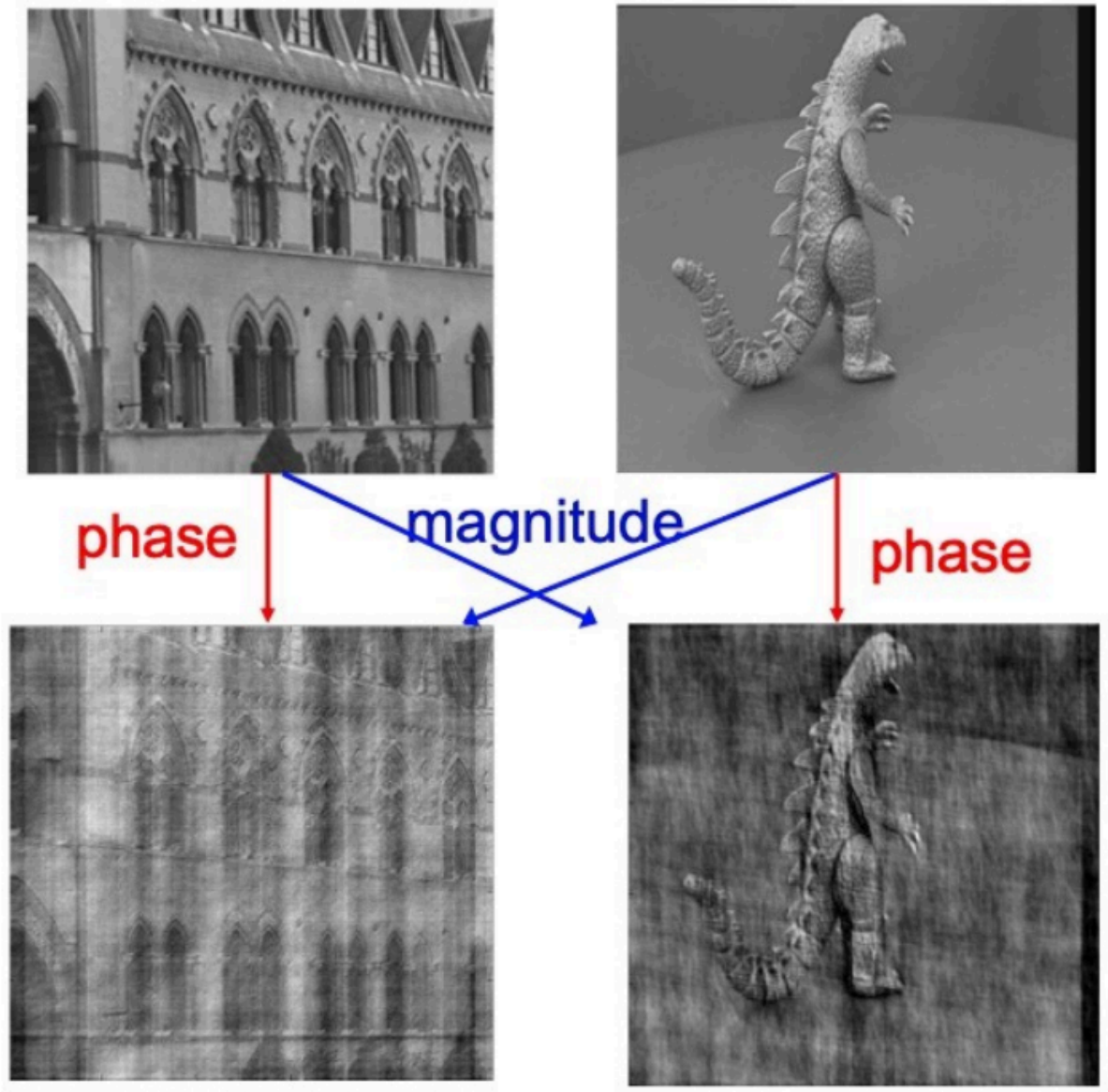


cheetah phase with zebra amplitude

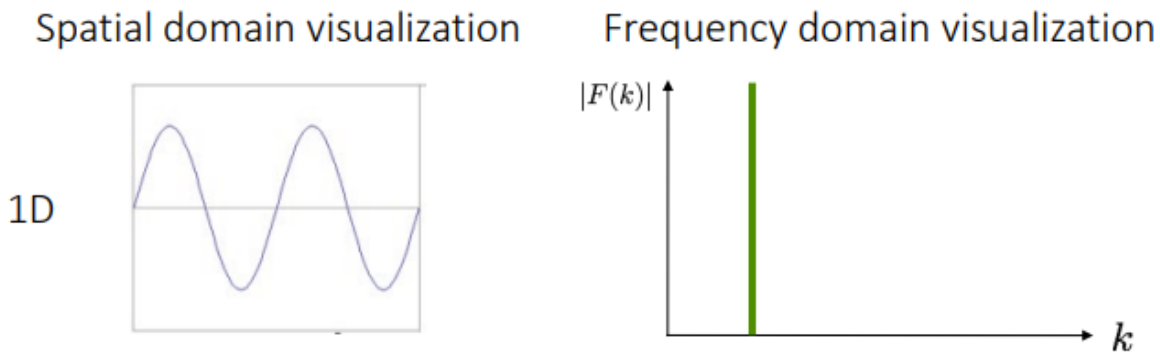


zebra phase with cheetah amplitude

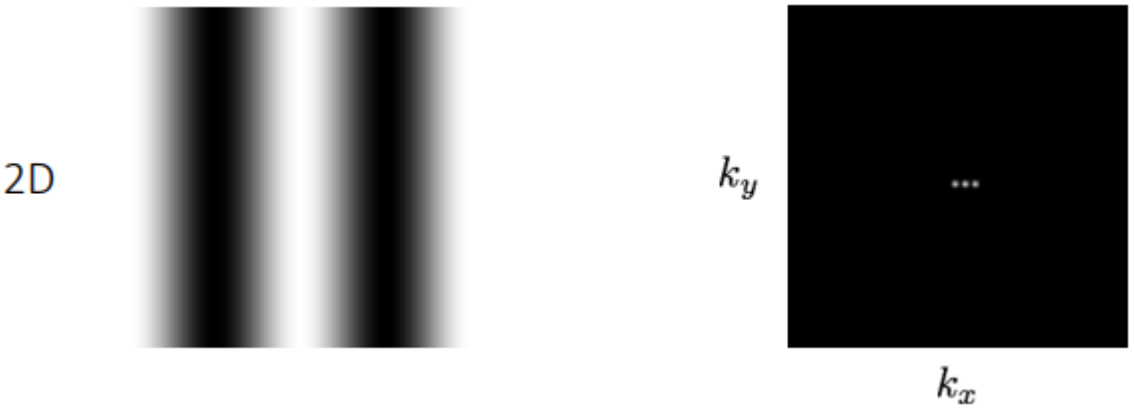
We see that the phase makes the image recognizable. Another example:



Simple Examples



This one below is a sinusoid in 2D, with the black part being the “high” part of the sinusoid (c.f. 1D sinusoid). On its right there is the amplitude part of its frequency domain, with k_x and k_y being the frequency direction in x and y respectively.

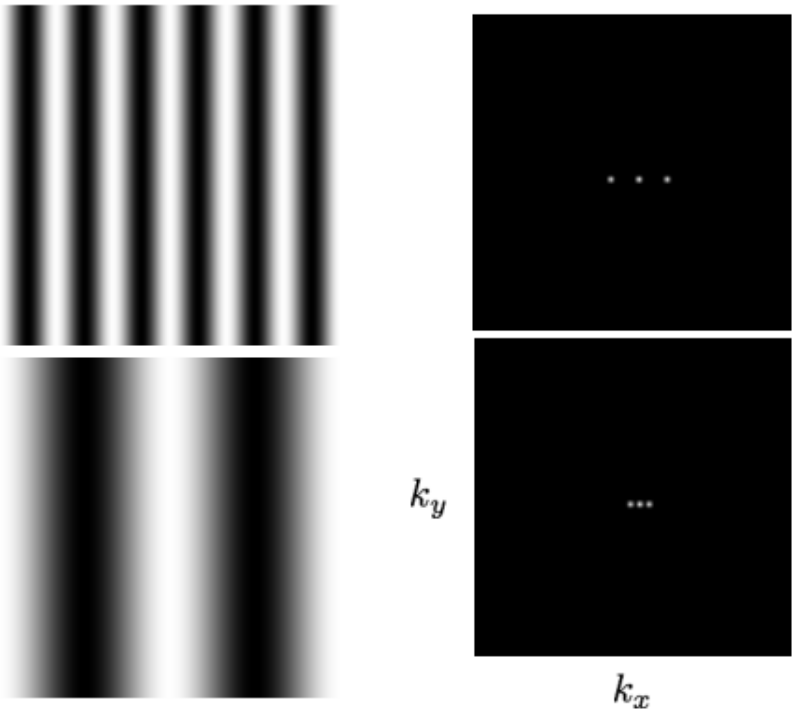


We have three dot that corresponds to the frequency components of the 2D signal:

- The central dot is the zero frequency component, which corresponds to the average brightness or DC component of the image
- The other two dots are the fundamental frequency components of the vertical stripes. They are symmetric w.r.t. the central dot along the k_x axis since the spatial pattern repeats in the horizontal direction. The distance of these two dots w.r.t. the center corresponds to the spatial frequency of the stripes.

Note that the k_x direction is the only relevant, there are no frequencies in k_y. If the black lines were horizontal, the points in the frequency domain would have been in vertical

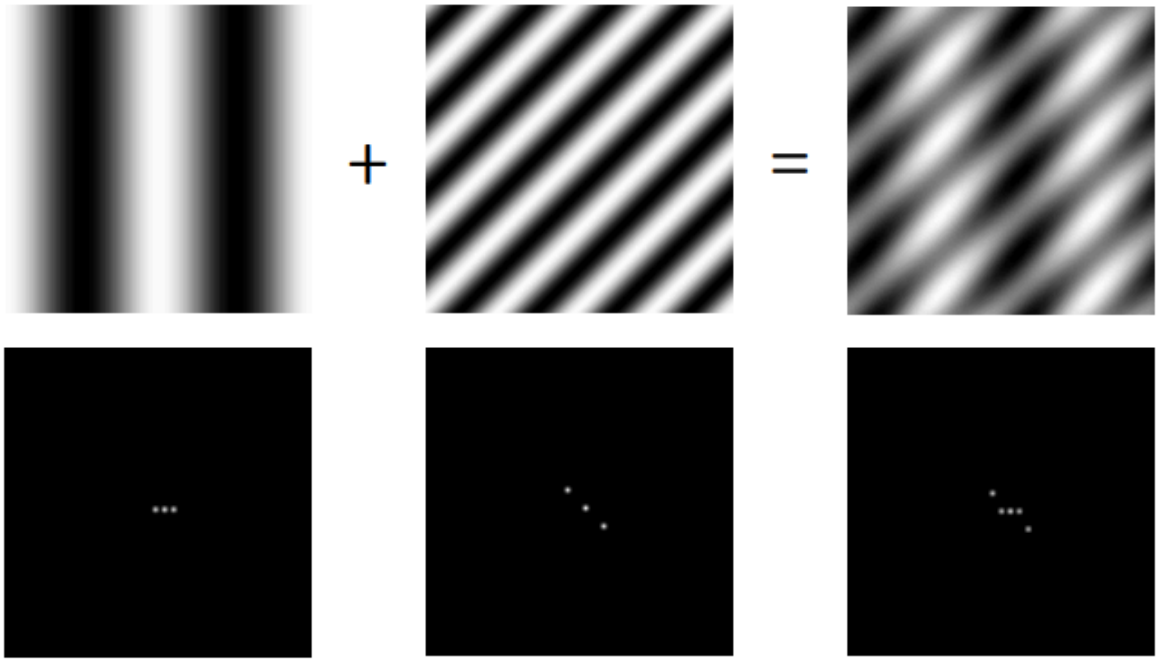
Scaling Property of the Fourier Transform



Stretching a function by a certain factor in the space domain is equal to squeeze the Fourier transform by the same factor in the frequency domain

Sum two 2D sinusoidal in the frequency domain

Note that, if we sum two 2D sinusoidals, we get in the frequency domain a combination of the frequency components from both sinusoidals.

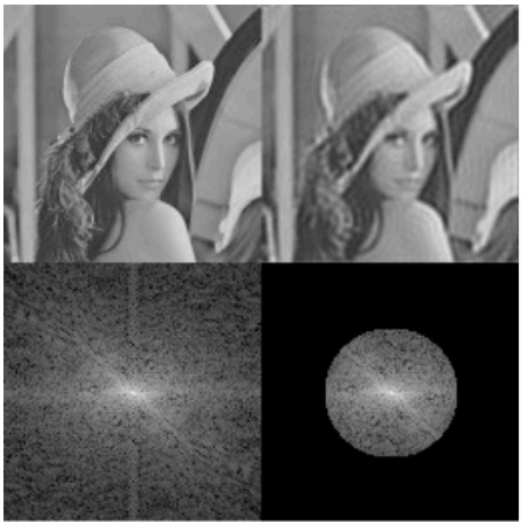


Filtering in the Frequency Domain

High-pass and Low-pass Filtering in the Frequency domain

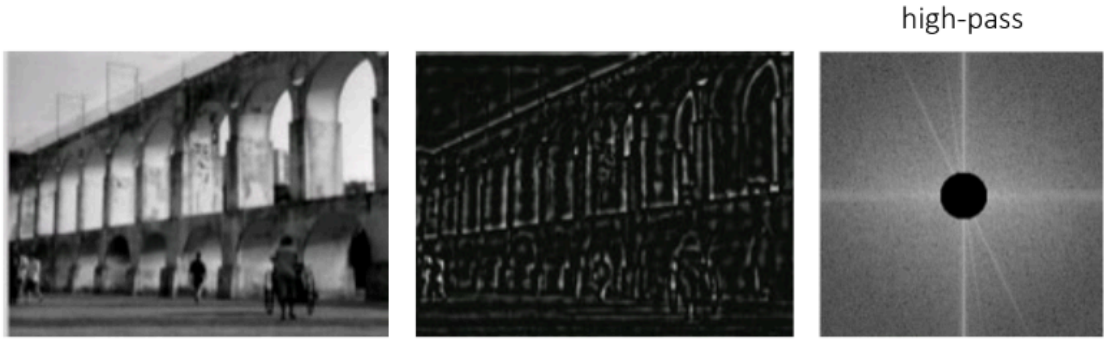
Instead of doing it in the spatial domain, we can perform LPF and HPF in the frequency domain. Performing Low-Pass Filtering we remove high-frequency components, preserving slow variations (smooth areas) and removing noise and fine details, while with High-Pass Filtering, we remove low-frequency components, emphasizing edges and fine details while suppressing smooth areas. For doing so, we apply the DFT to the image, getting its representation in the frequency domain

This is the equivalent of applying a Gaussian Filter, with a certain sigma, on the image in the spatial domain, since the high frequencies are the one that are farthest away from the origin. In fact, keeping the circle, we are keeping the low frequency, the ones that are closer to the origin. Now we can apply the inverse DFT, getting the image in the spatial domain, now blurry.



Low pass Filtering with Gaussian Filter

The contrary holds true for high pass filtering, where is the center that is removed.



Band-Pass Filtering

With band-pass filtering we keep only a “ring” in the frequency domain, removing high and low frequencies.

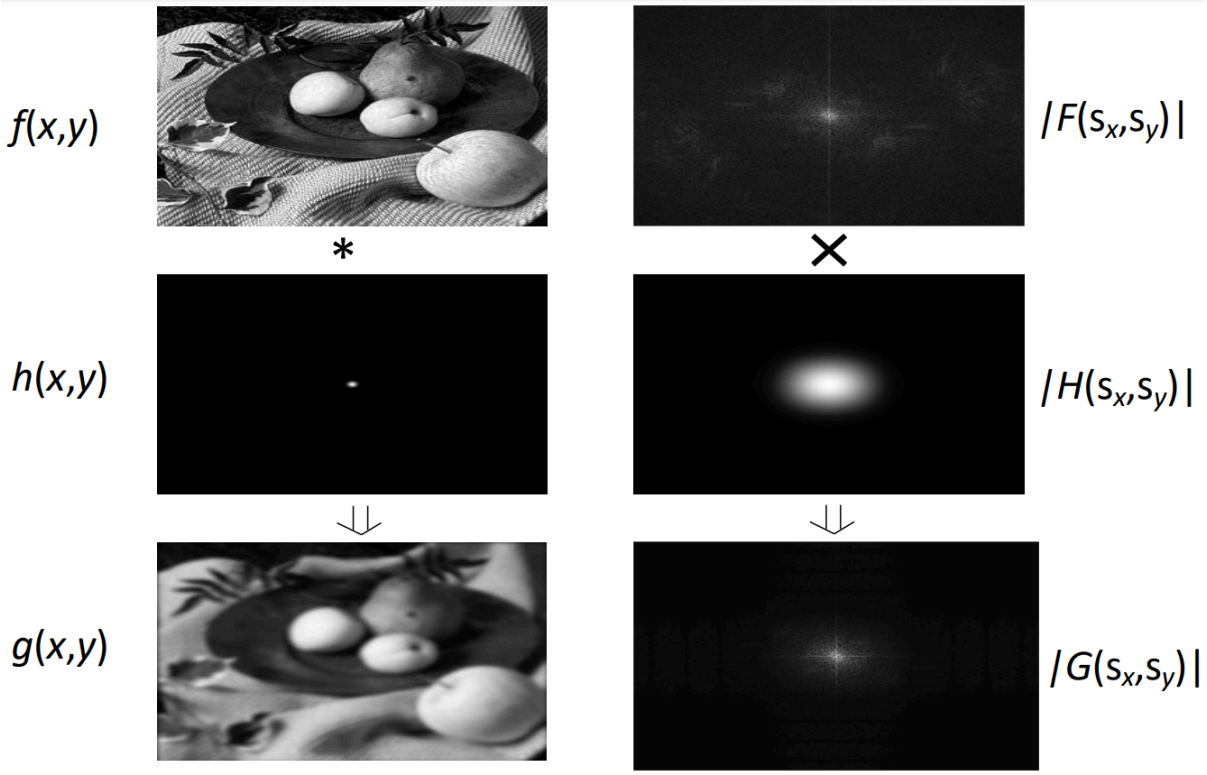
The Convolution Theorem

Convolution in the spatial domain is the multiplication in the frequency domain

$$\mathrm{F}[g * h] = \mathrm{F}[g]\mathrm{F}[h]$$

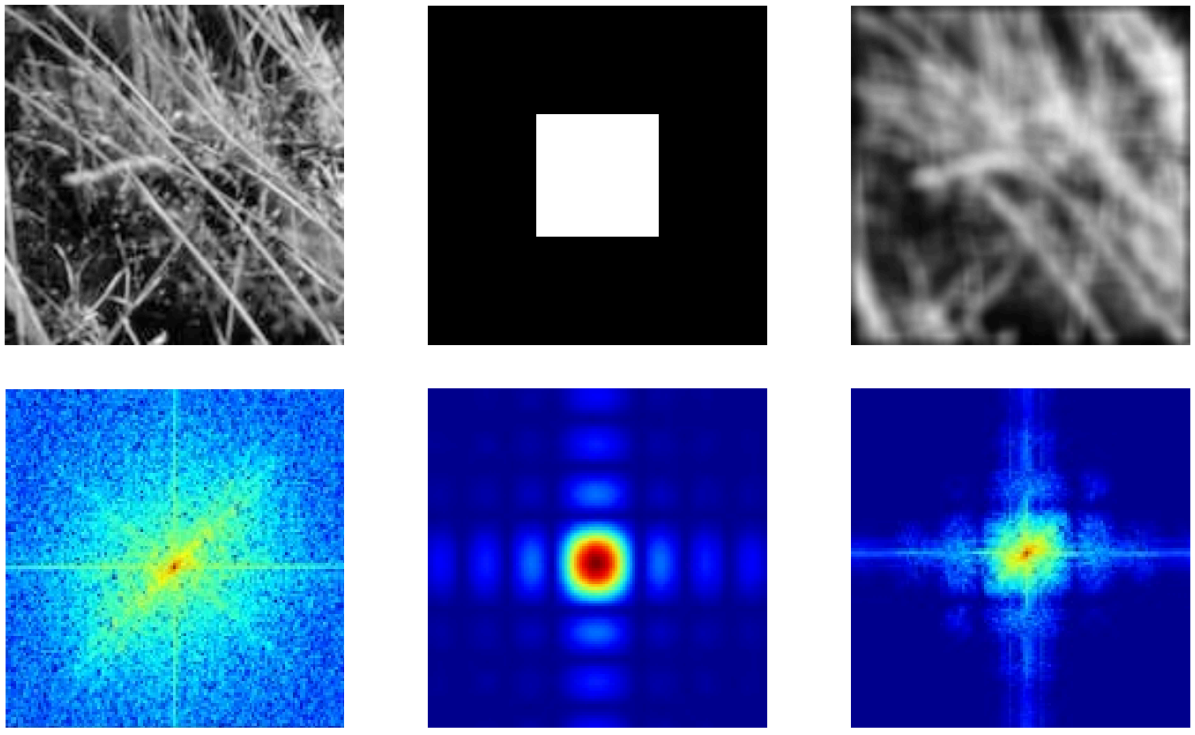
$$\mathrm{F}^{-1}[gh] = \mathrm{F}^{-1}[g] * \mathrm{F}^{-1}[h]$$

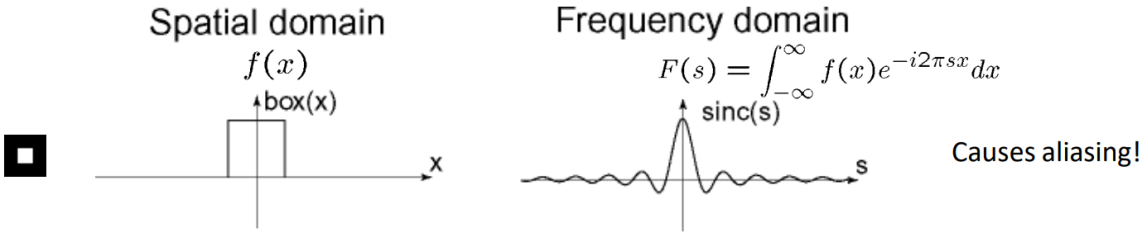
Let’s do an example:



So we have filtered the image in the frequency domain instead of in the spatial domain.

Finally, we can revisit blurring and understand why the gaussian filter is better than the box. This is the box filtering:





As you can see, the box filter, in the frequency domain, is a sinc function, that causes aliasing

This, instead, is the gaussian filtering:

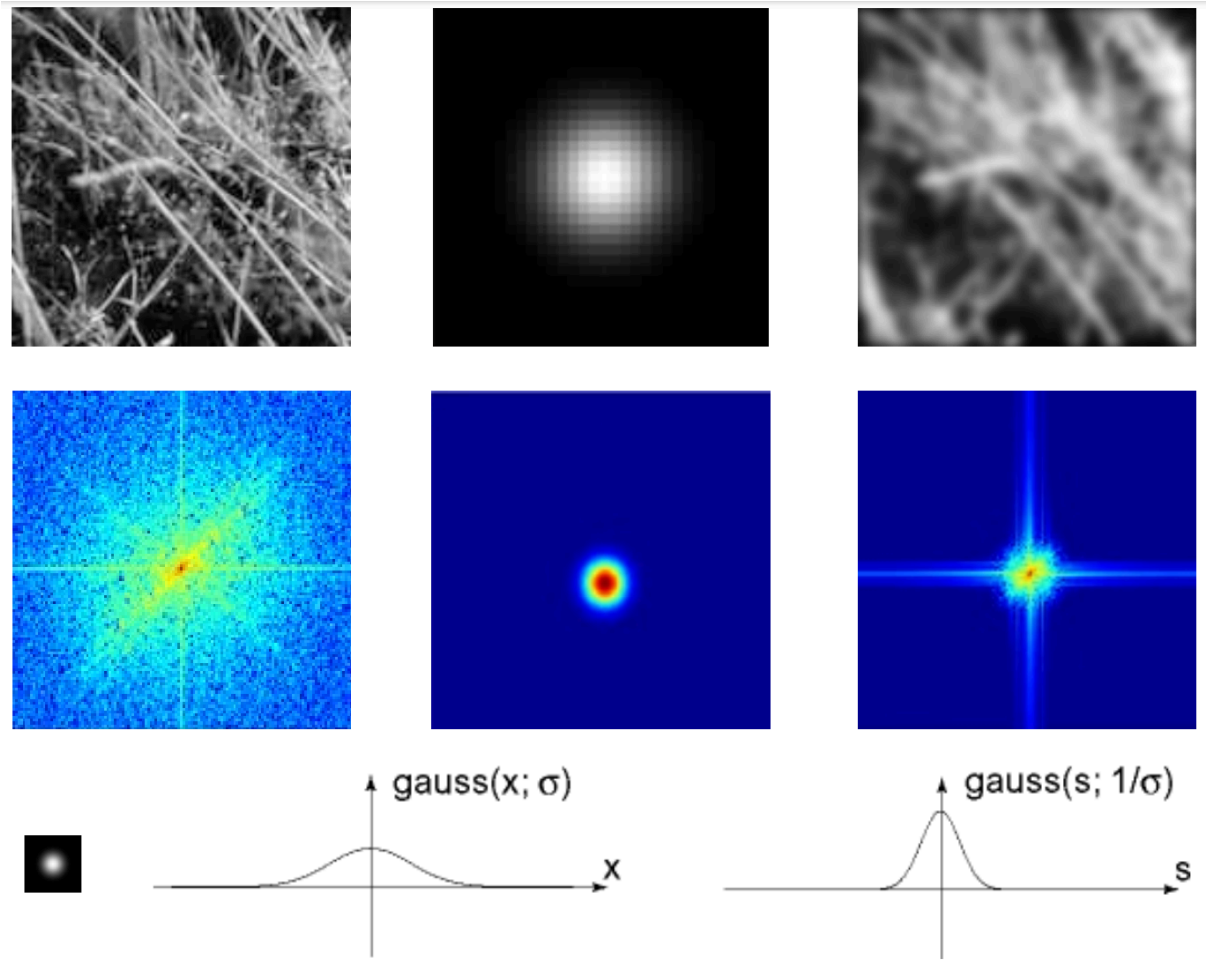


Image Pyramid

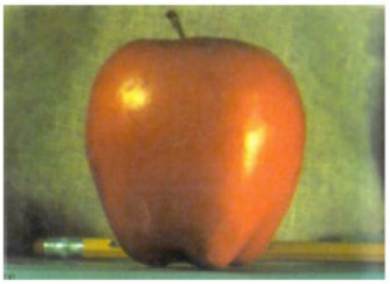
12/03/2024

Why use pyramids:

- Multi-scale image analysis:
    - task of finding an object in an image. We don't know the scale at which the object will appear, so we need to generate a whole pyramid of differently sized images and scan them.

The red box is the size of the template. Once it is fixed, the recognized birds are the only ones that have that size. By running this template across different levels of the pyramid, different bird instances are detected.

- blending image while maintaining details:



We build Laplacian pyramids for both images. Then we build a Gaussian pyramid for the mask and we combine all in a Laplacian pyramid

Gaussian Pyramid

The following is the algorithm to build a Gaussian pyramid:

Repeat:

Convolute the image with a Gaussian filter

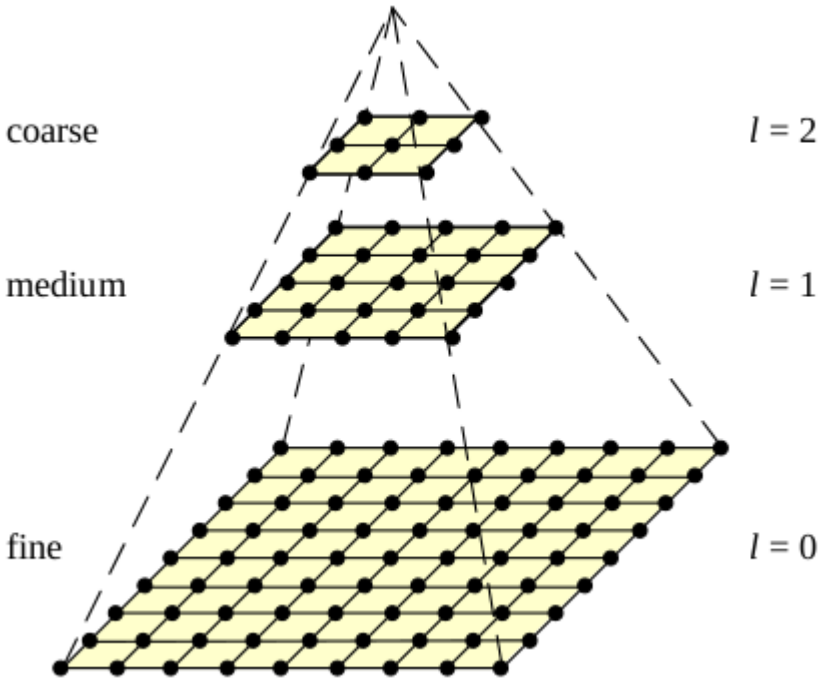
Subsample
Until minimum resolution is reached

Binomial Kernel

The binomial kernel approximates a Gaussian Kernel. Getting its value from the Pascal's triangle, the binomial coefficients of, for instance, a 1D kernel of width 5 are:

$$\frac{1}{16}\begin{array}{|c|c|c|c|c|}\hline 1 & 4 & 6 & 4 & 1 \\ \hline\end{array},$$

To get the 2D version, you just take the outer product of this kernel with itself.

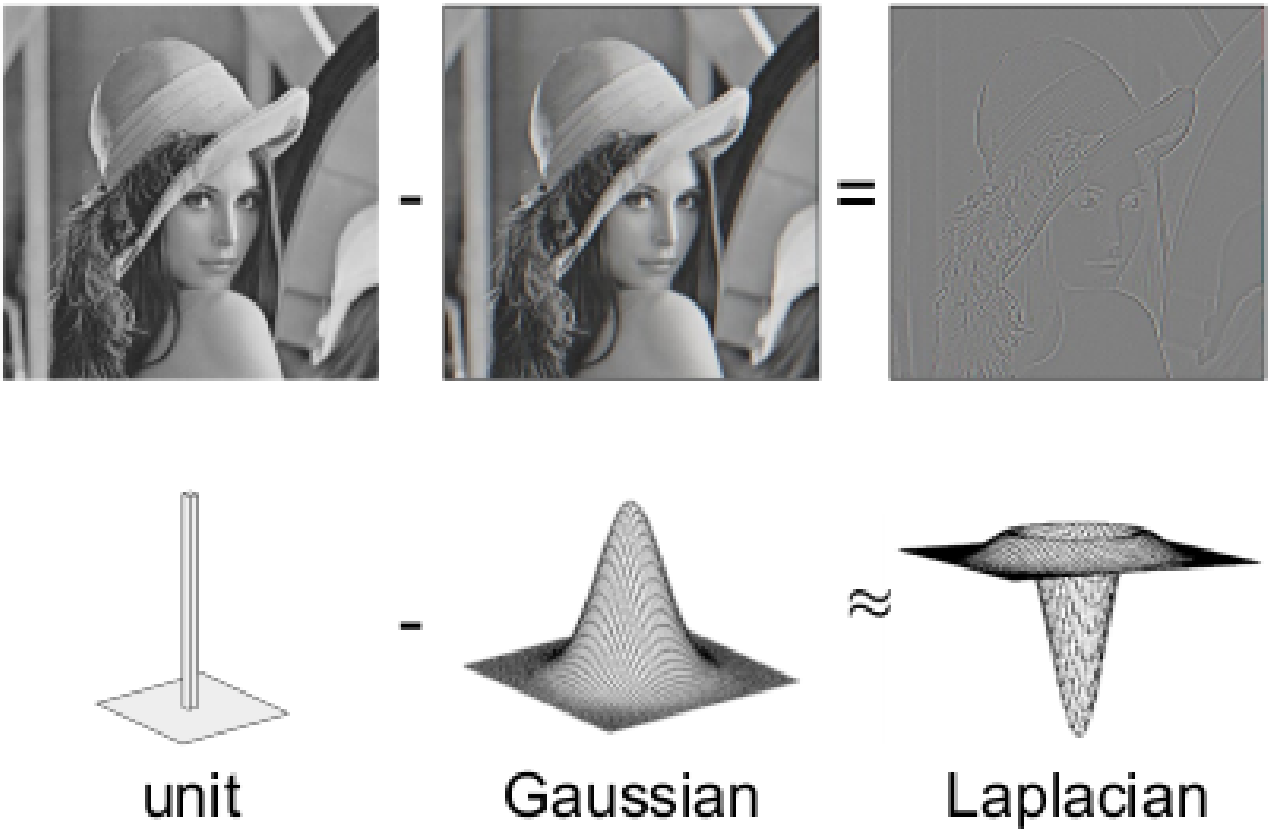


Adjacent levels of the pyramid have a sampling rate r = 2, so that means that each level has half the resolution of its parent.

Laplacian Pyramid

In the Gaussian pyramid, the details of the image get smoothed out as we move to higher level, and it's not possible to reconstruct the original image. This is because blurring is lossy. The Laplacian image pyramid is instead lossless. It's called Laplacian Pyramid since the

difference of two gaussians approximates a laplacian[4]



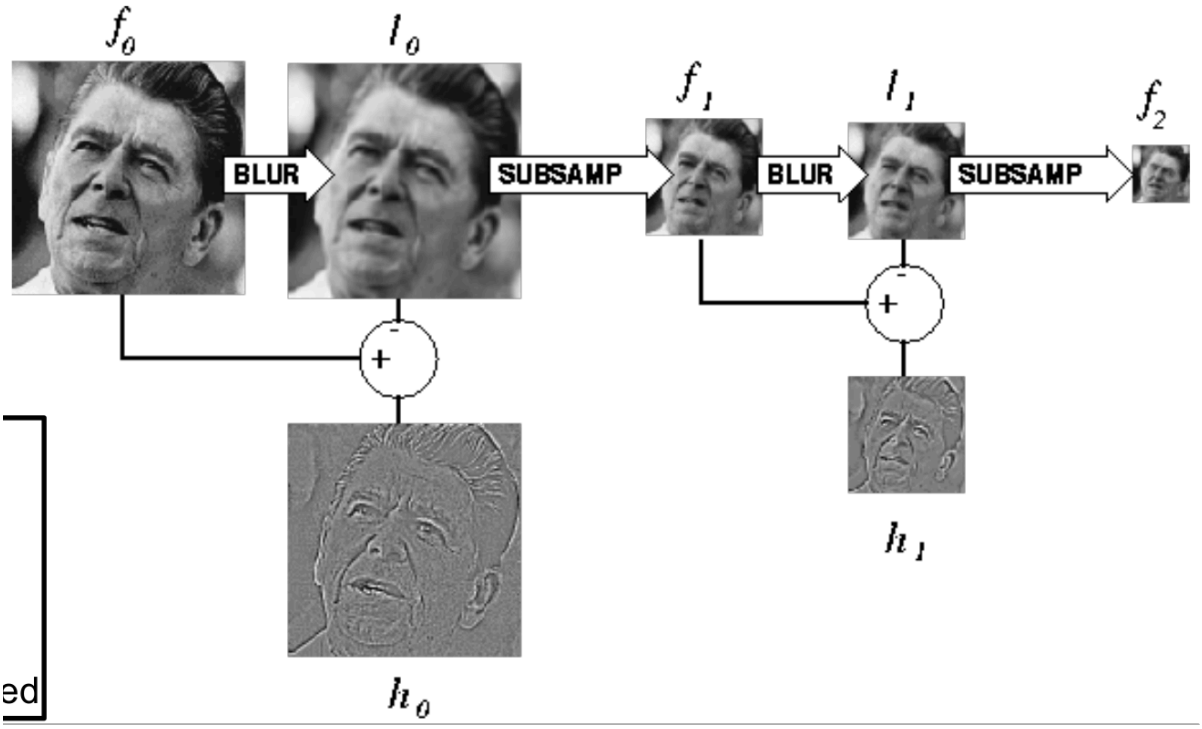
At each level of the Laplacian Pyramid we retain the residuals instead of the blurred images

Repeat:

- Convolute the image with the Gaussian Kernel
- Compute Residual
- Subsample

Until minimum resolution is reached

[4] indeed: Image*Unit - Image*Gaussian = Image*Laplacian



What we essentially need to reconstruct the image is:

- The smallest image, so the peak of the pyramid. The image with min resolution.
- All the residuals

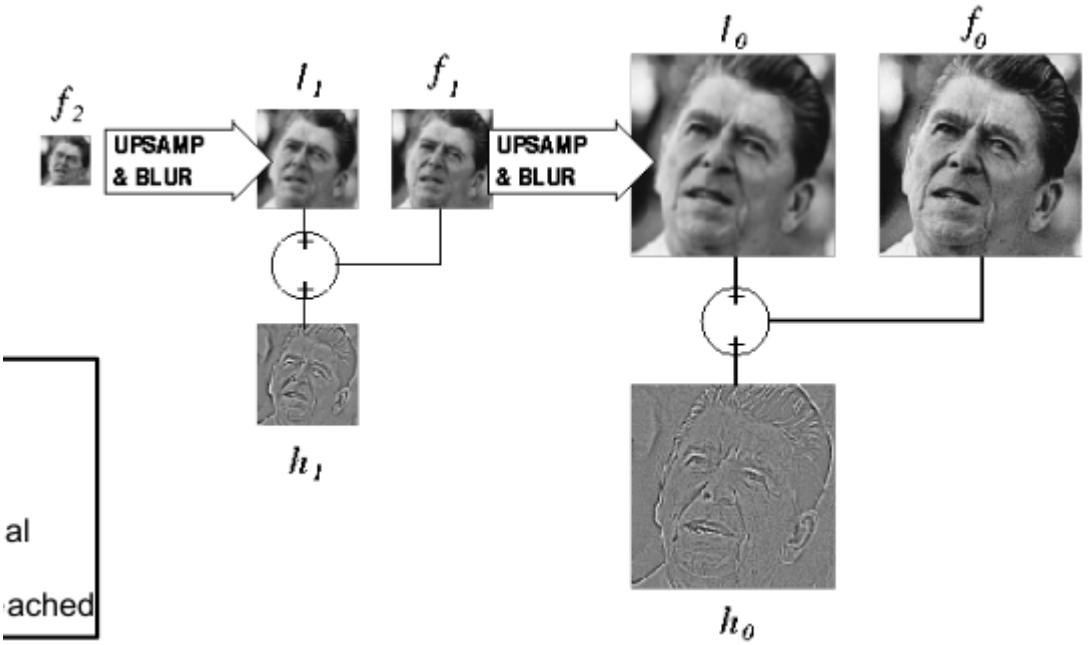
having these, you can do the following:

Repeat:

Upsample

Sum with residuals

Until original resolution is reached



So you don't need to store all the intermediate blurred images

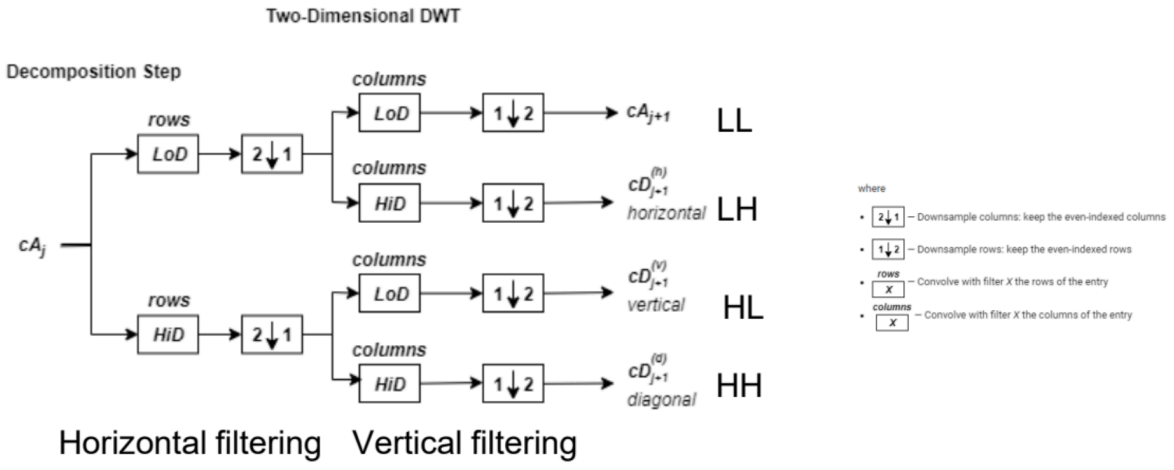
Wavelets

Wavelet represents brief oscillations. They are used to divide an image, that is a signal, in two sub-signal:

- approximations
- details

Applying a 2D Discrete Wavelet Transform (DWT) to an image, we first convolve the image with the high pass and low pass filters along the rows, resulting in 2 intermediate sub-signals. Then both of these intermediate sub-signal are downsampled and convolved with the two filters along the columns, getting, then a final downsampling, in order to obtain:

- LL: this is the approximation component, representing the overall trend
- LH: this represents the horizontal details
- HL: this represents the vertical details
- HH: this represents the diagonal details



To have a practical example of wavelets, let’s consider this 3-level 2D DWT pyramid:

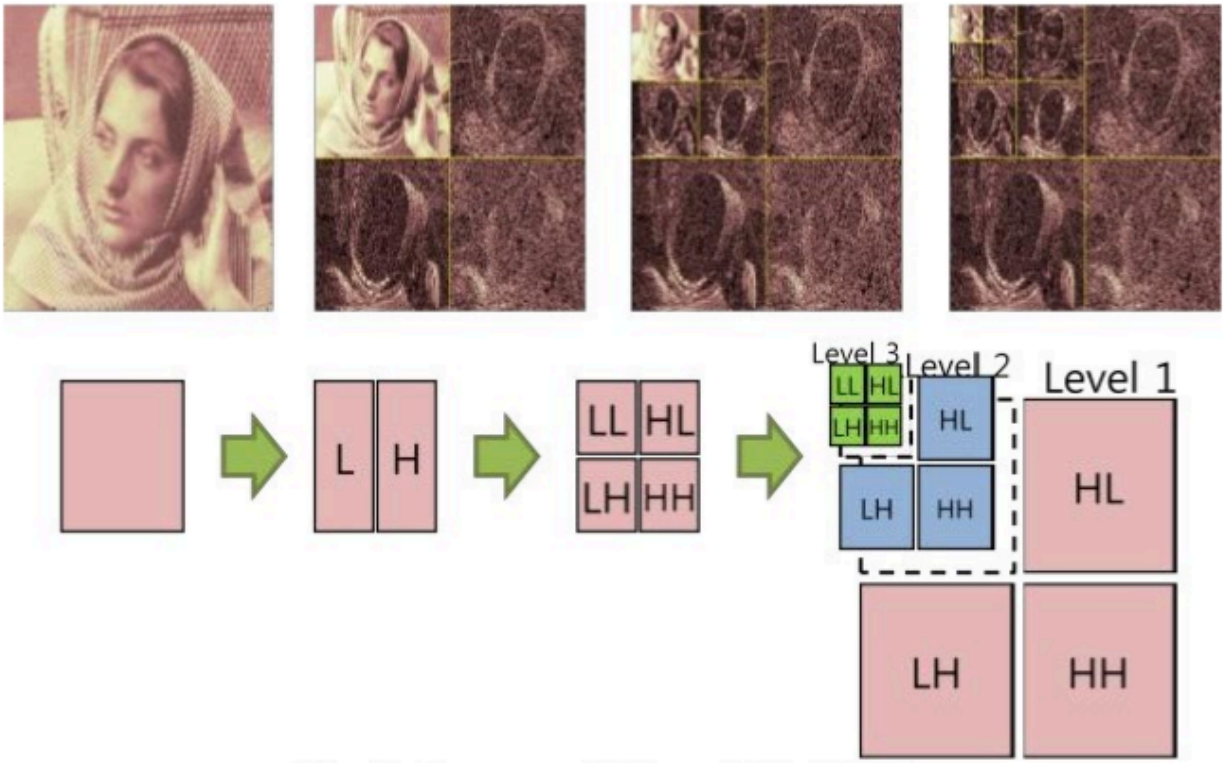
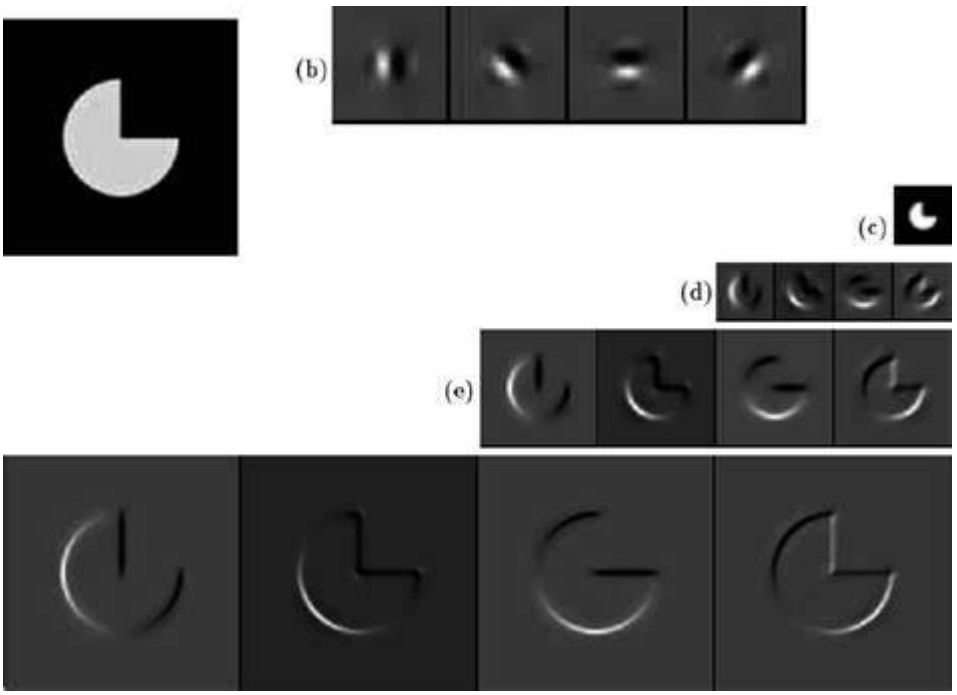


Fig. 3. Process of 3-Level 2D-DWT.

The process begins with the original image, passed through the DWT in order to obtain the four components. Then, on the second level, the approximation component (LL) from the first level is further decomposed. This second-level decomposition again produces four sub-bands: LL, LH, HL, and HH. The same holds for the third level.

Steerable Pyramid

At each level keep multiple versions, one for each direction



Feature Detection

12/03/2024, 13/03/2024

First of all, what is a feature? Is a set of salient keypoints (so basically pixels) of an image.

Global Features vs Local Features

Global features describe the image as a whole to generalize the entire object, while local features describe the image patches (key points) of the object.

Advantages of Local Features

1. Locality: so they are robust to occlusion and clutter
2. Quantity: they are hundreds or thousands in an image
3. Distinctiveness: can differentiate a large database of objects
4. Efficiency: real-time performance achievable

Invariance of Local Features

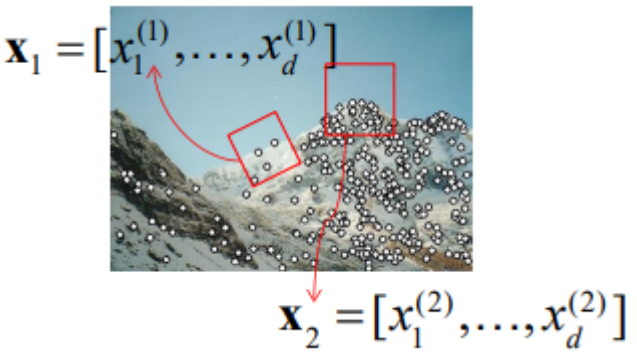
1. Geometric invariance: translation, rotation, scale
2. Photometric invariance: brightness, exposure

Main components of the Algorithm

The passages are:

1. Detection: identify the interest points, i.e. looking for good features to match

2. Description: extract vector feature surrounding each interest point, this vector is a descriptor that can be matched against other descriptors.

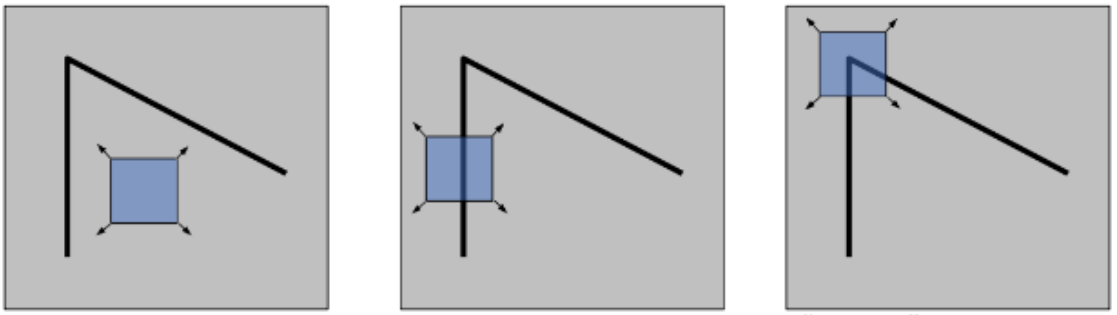


3. Matching: determine correspondence between descriptors in two views

An application of this can be the following**:** after matching, you can align the images computing transformation to align them and blend. After that you will have combined the images.

Harris Corner Detection

It is used to detect corners and edges. The idea is that it is possible to localize a corner by shifting a window:



"flat" region: no change in all directions

"edge": no change along the edge direction

"corner": significant change in all directions

Consider shifting the window W by (u, v). We can compare each pixel before and after the shifting with the Sum of Squared Difference (SSD)

$$E(u,v) = \sum_{(x,y)\in W} [I(x+u, y+v) - I(x,y)]^2$$

Image moved by the displacement — original

if this error is high is good, since the corner has high E(u,v) for all values of (u,v). **Drawback**: it's slow to compute exactly for each pixel and each offset (u, v).

Small Motion Assumption

(u,v) is a small shift, so we can approximate the Taylor Series expansion:

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

with the first order:

$$\begin{aligned} I(x+u, y+v) &\approx I(x,y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v \\ &\approx I(x,y) + \begin{bmatrix} I_x & I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \end{aligned}$$

So the error becomes:

$$
\begin{aligned}
E(u,v) &= \sum_{(x,y)\in W} [I(x+u,y+v) - I(x,y)]^2 \\
&\approx \sum_{(x,y)\in W} [I(x,y) + I_x u + I_y v - I(x,y)]^2 \\
&\approx \sum_{(x,y)\in W} [I_x u + I_y v]^2
\end{aligned}
$$

Thus, **E(u,v) is locally approximated as a quadratic error function**

Second Moment Matrix

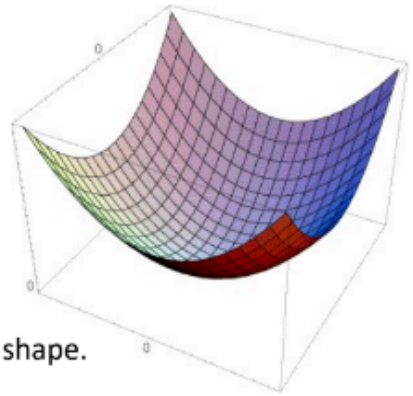
We can rewrite the quadratic function with a matrix form

$$
\begin{aligned}
E(u,v) &\approx Au^2 + 2Buv + Cv^2 \\
&\approx \begin{bmatrix} u & v \end{bmatrix} \underbrace{\begin{bmatrix} A & B \\ B & C \end{bmatrix}}_{H} \begin{bmatrix} u \\ v \end{bmatrix}
\end{aligned}
$$

$$
A = \sum_{(x,y)\in W} I_x^2
$$

$$
B = \sum_{(x,y)\in W} I_x I_y
$$

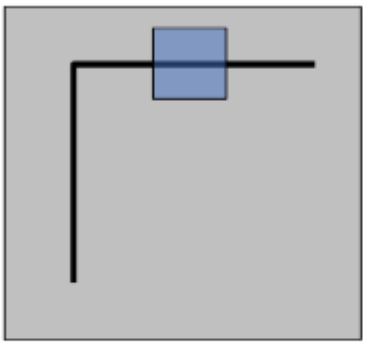
$$
C = \sum_{(x,y)\in W} I_y^2
$$



Let's try to understand its shape.

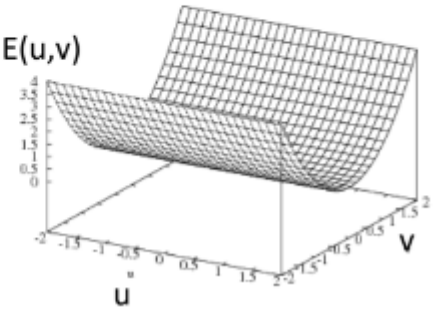
Let's do two examples:

1. In the case of an horizontal edge, we have



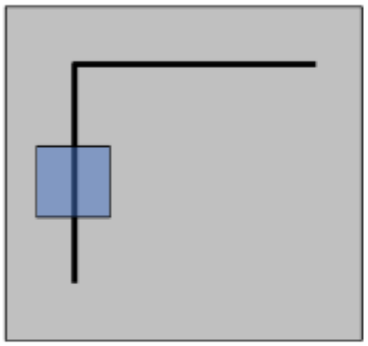
Horizontal edge: $I_x = 0$

$$H = \begin{bmatrix} 0 & 0 \\ 0 & C \end{bmatrix}$$



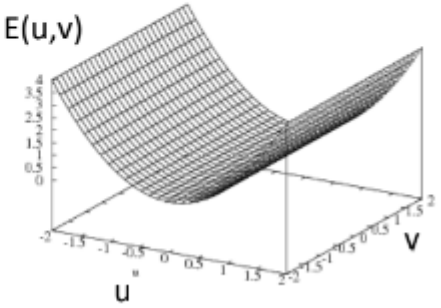
In fact, E(u,v) is [0, C*v^2], so it's a quadratic function w.r.t. to v, and is 0 w.r.t u, as shown in the 3D plot above.

2. In the case of a vertical edge:

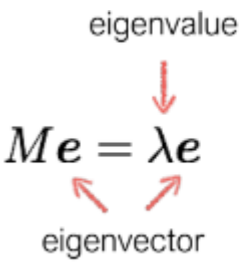


Vertical edge: $I_y = 0$

$$H = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix}$$



Eigenvalues and Eigenvectors and Ellipse Visualization



$$(M - \lambda I)e = 0$$

To find eigenvalues and eigenvectors, these are the steps:

1. Compute the determinant of $M - \lambda I$ (returns a polynomial)

2. Find the roots of polynomial $\det(M - \lambda I) = 0$ (returns eigenvalues)

3. For each eigenvalue, solve $(M - \lambda I)e = 0$ (returns eigenvectors)

Once you have them, you can visualize the M matrix as an ellipse, since the ellipse equation in quadratic form is the following:

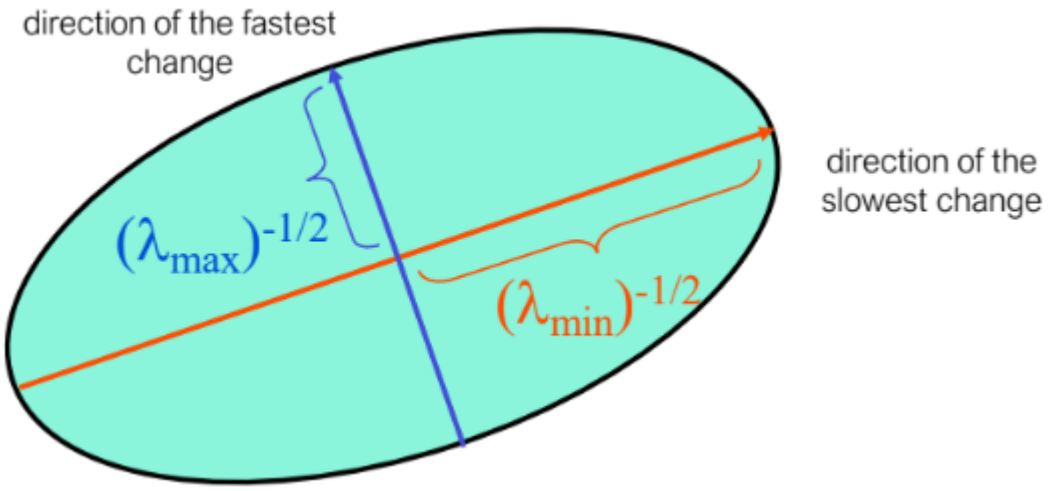
$$\begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} = \text{const}$$

Where M, since it's symmetric, can be written as:

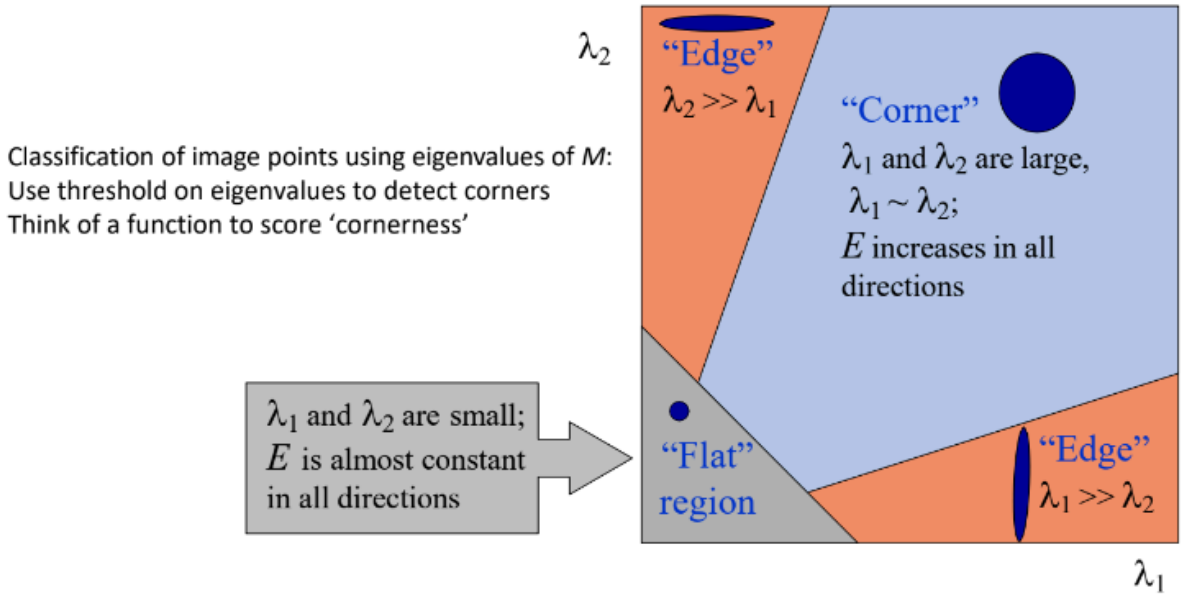
$$M = R^{-1} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} R$$

The smaller eigenvalue corresponds to the major axis direction, that is the direction of the slowest change, while the bigger eigenvalue corresponds to the ino axis direction, that is the direction of the fastest change.

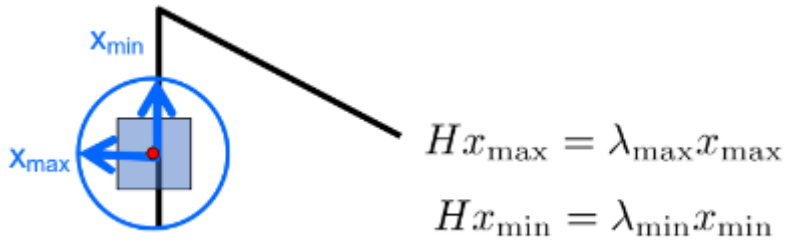
The directions of these axis are determined by the eigenvectors **R** = (**e1**, **e2**)



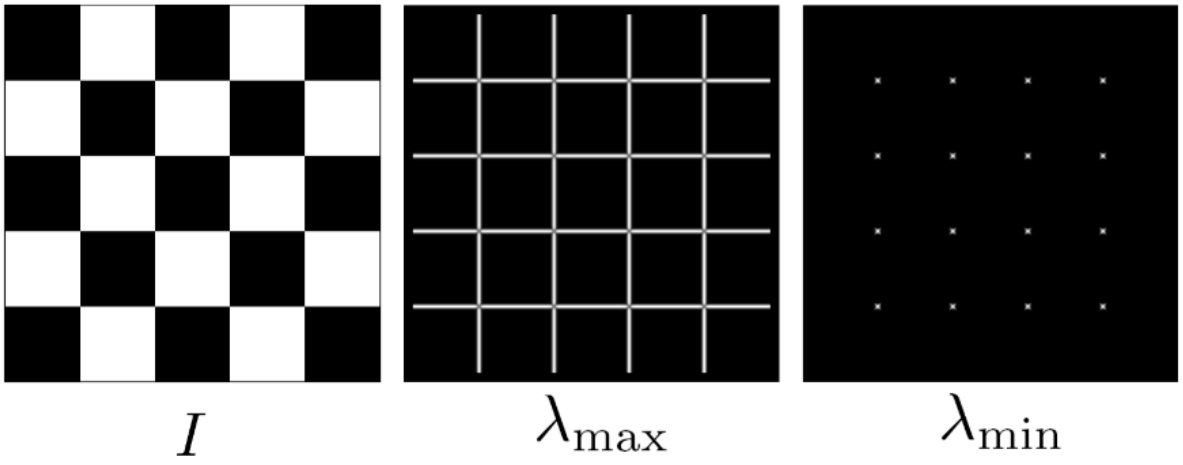
Now that we know eigenvectors and eigenvalues, we can understand if we are dealing with an horizontal edge, a vertical edge, a corner or a flat area:



This is an example:



Let's visualize this concept: in the figure above, I is the original input. lambda max indicates where an edge (both vertical and horizontal) is detected, while lambda min is big[5] in corners because both eigenvalues are big in corners.



Corner Response Functions

Since the presence of the corner or not depends on the smaller eigenvalue, it makes sense to find maxima in the smaller eigenvalue to locate good features to track. However, this is not the only quantity used to find keypoints, but we can use:

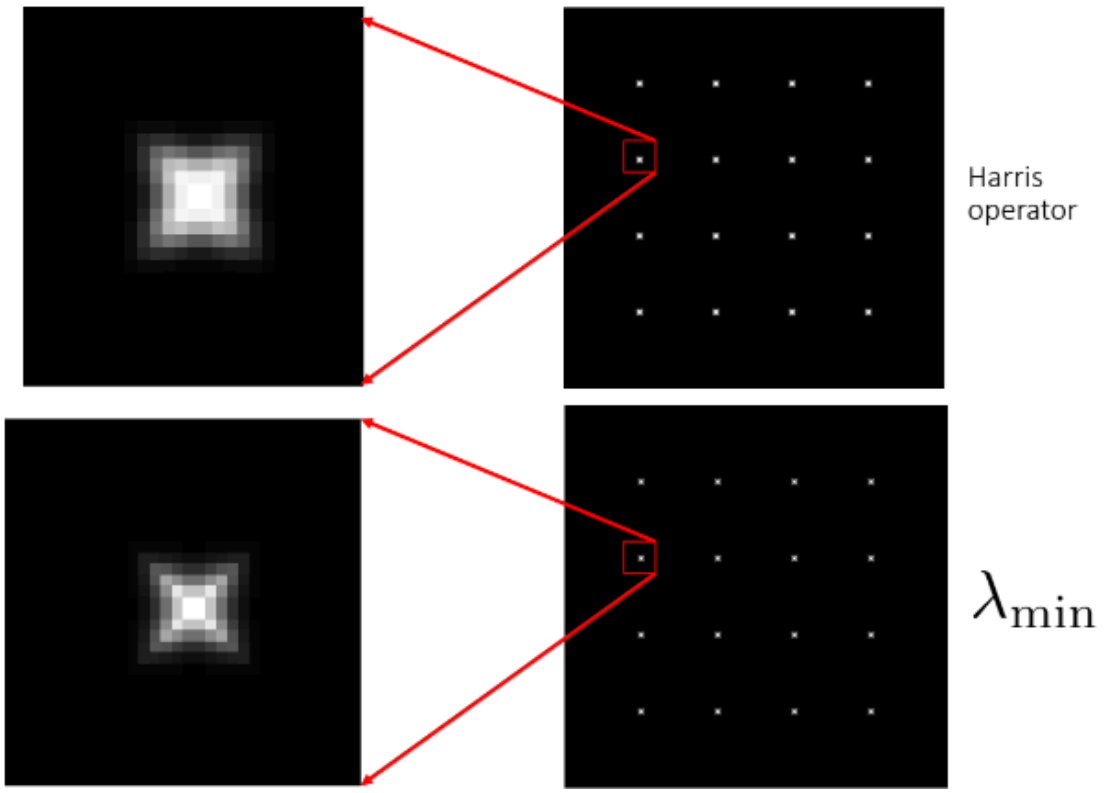
$$f = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2}$$

$$= \frac{determinant(H)}{trace(H)}$$

$$R = \lambda_1 \lambda_2 - k \cdot (\lambda_1 + \lambda_2)^2 = \det(M) - k \cdot \mathrm{tr}(M)^2$$

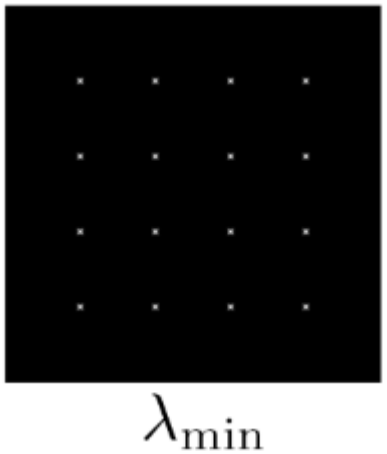
Difference between using the Harris Operator or the simplified version lambda_min

[5] Big = bright. So we are increasing the intensity when lambda is increasing



Algorithm Steps

1. Compute the gradient at each point in the image.
2. For each point:
    a. Compute H and its eigenvalues/eigenvector in a window around the pixel
    b. Compute corner response function
    c. Threshold the corner response function, namely keep only the point where the response function is above the threshold
3. Non-Maximum Suppression: find local maxima of response function. In the figure below, the bright points are the local maximum.

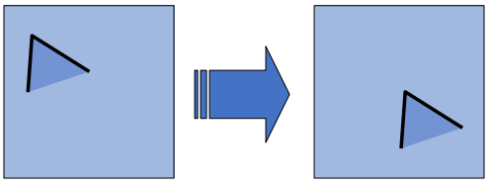


These points will be the features.

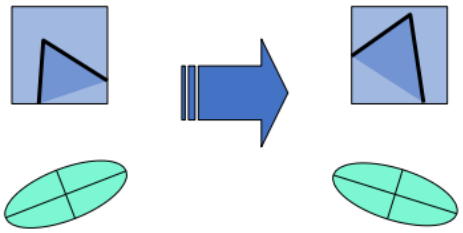
Invariance and Equivariance of Harris Corner Detector

We want corner locations to be invariant[6] to photometric transformation (e.g. intensity change) and equivariant[7] to geometric transformations (e.g. rotation and scaling). We have:

1. Corner location is equivariant w.r.t. to Translation



2. Corner location is equivariant w.r.t. Image Rotation



   The Second Moment Ellipse is rotated but its shape (so the eigenvalues) is the same

3. Corner location is partially invariant to affine intensity change (I -> I*a + b)

   a. Invariance to intensity shift (I -> I + b): in fact, since only derivatives are used, if you shift every intensity in the image by a certain constant b, you have the same values for the differences of the intensity.

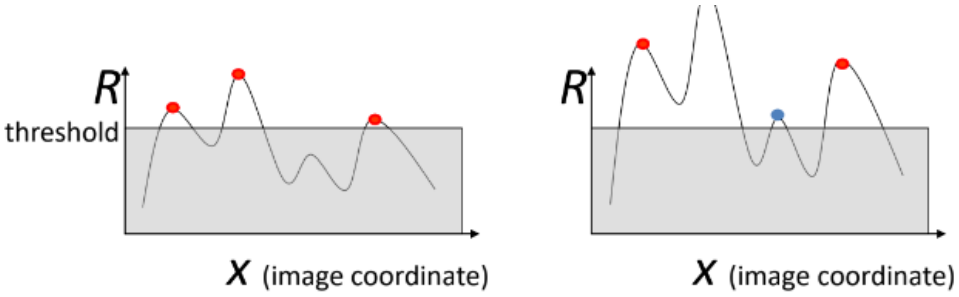
   b. Partial Invariance to intensity scale (I -> a*I): because there is an intensity threshold that is fixed (so you are not scaling it by a), if you scale the function some local minimum may get above or below the threshold, namely the corner response function will be scaled by a. So the corner location is partial

---

[6] the corner locations do not change. Sometimes invariant is used to refer to “equivariant”. If your pattern is 0,3,2,0,0 and the result is 0,1,0,0 then pattern 0,0,3,2,0 would lead to the same result 0,1,0,0

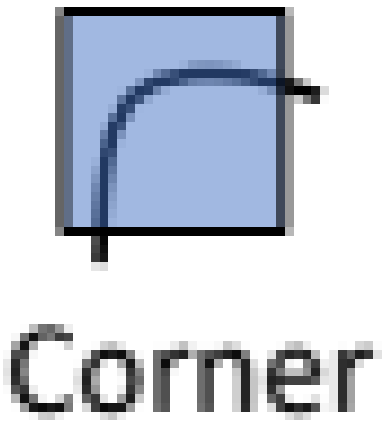
[7] if we have two transformed versions of the same image, features should be detected in corresponding locations. Sometimes called covariant. If your pattern is 0,3,2,0,0 and the result is 0,1,0,0 then pattern 0,0,3,2,0 would lead to the same result 0,0,1,0

invariant w.r.t to intensity scale

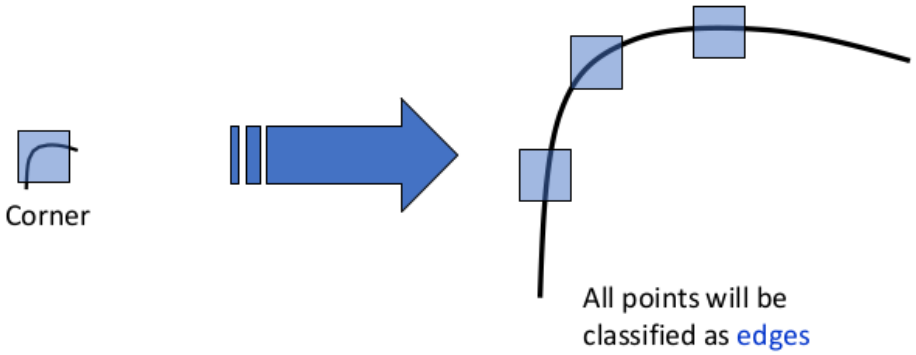


In summary, the point is that the derivative of (I + b) is equal to I, so we have invariance, while the derivative of I*a is a * derivative(I), so this constant will scale the corner response function by a factor a, and since the threshold remains fixed, it will change the accepted features.

4. Corner location is neither invariant nor equivariant to scaling. For instance take this image:



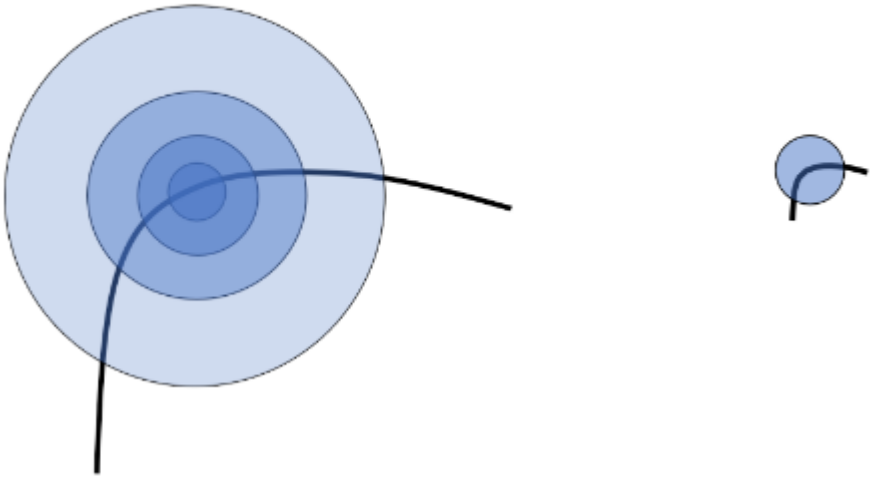
This is detected as a corner, but if we scale this image and the patch keeps being of the same size, the point of the corner will be classified as edges.



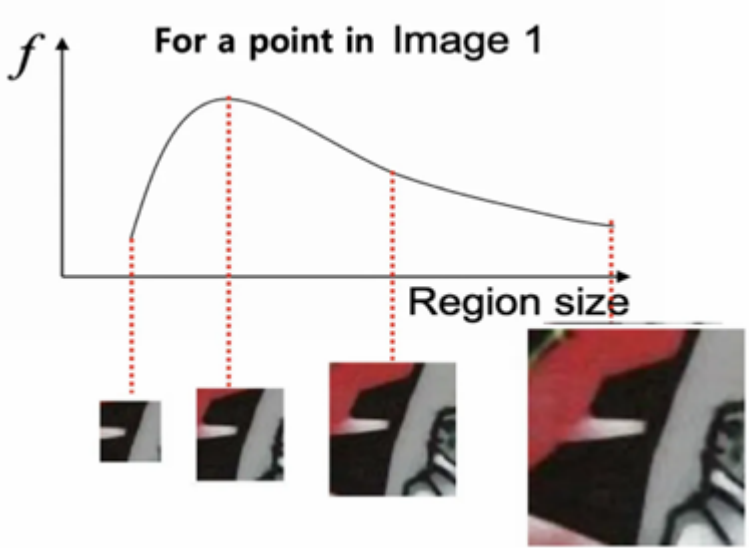
That means that we need a scale-invariant detector.

Scale Invariant Detection: Blob Detector

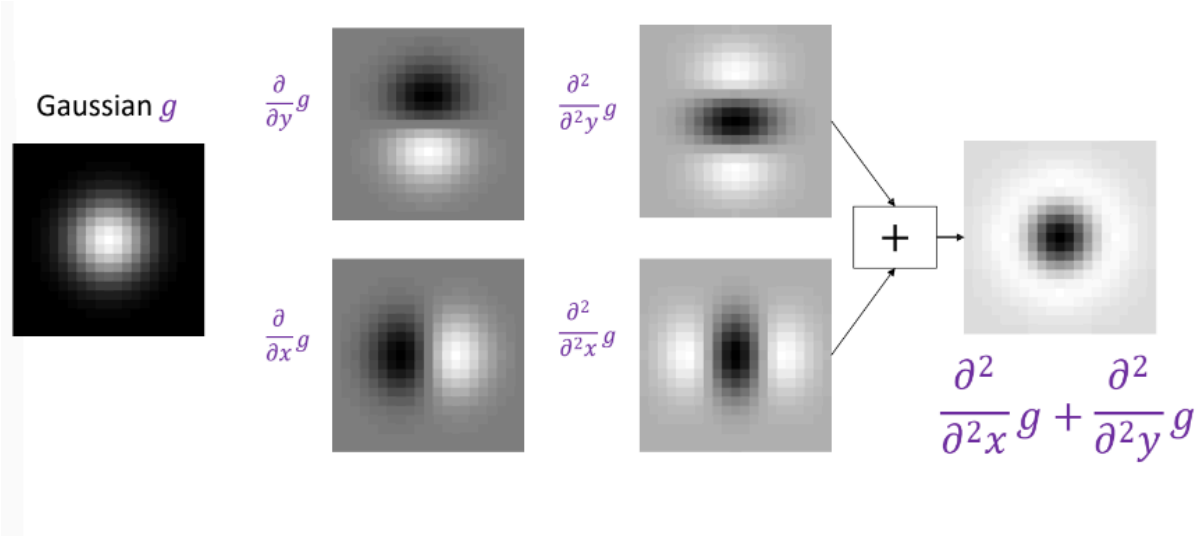
To solve the problem of non-invariance or equivariance to scaling. We can do the following: we consider regions (e.g. circles) of different sizes around the point. Regions of corresponding sizes will look the same in both images. So now we are looking for *blobs*.



But how do we choose corresponding circles independently in each image? We choose the scale of the “best” corner. The idea is, given a function f, to find the scale that gives the local maximum of f. Precisely, we are using a function f that is a function of the region size



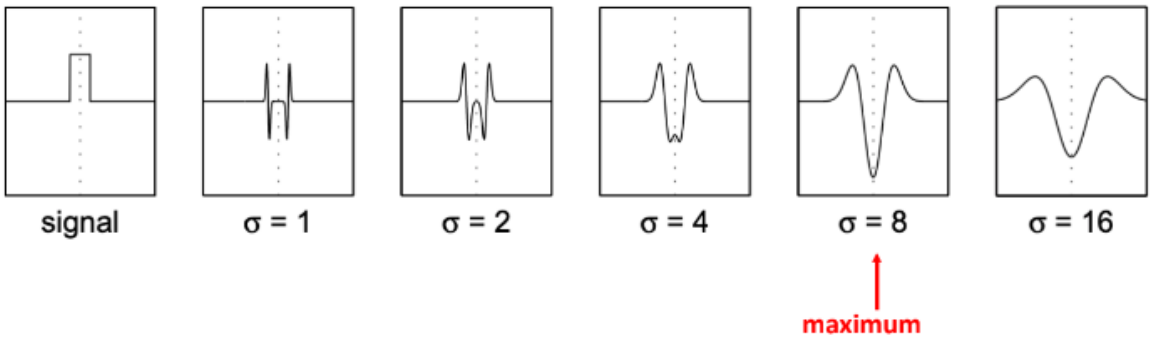
The local maxima of the function is a strong clue indicating that the corresponding region size is invariant to image scale. What scale-invariant response function f to use? We can use the scale-normalized LoG filter (NLoG):



This is the Laplacian of Gaussian, but we have to multiply it by sigma^2 to make responses comparable across scales

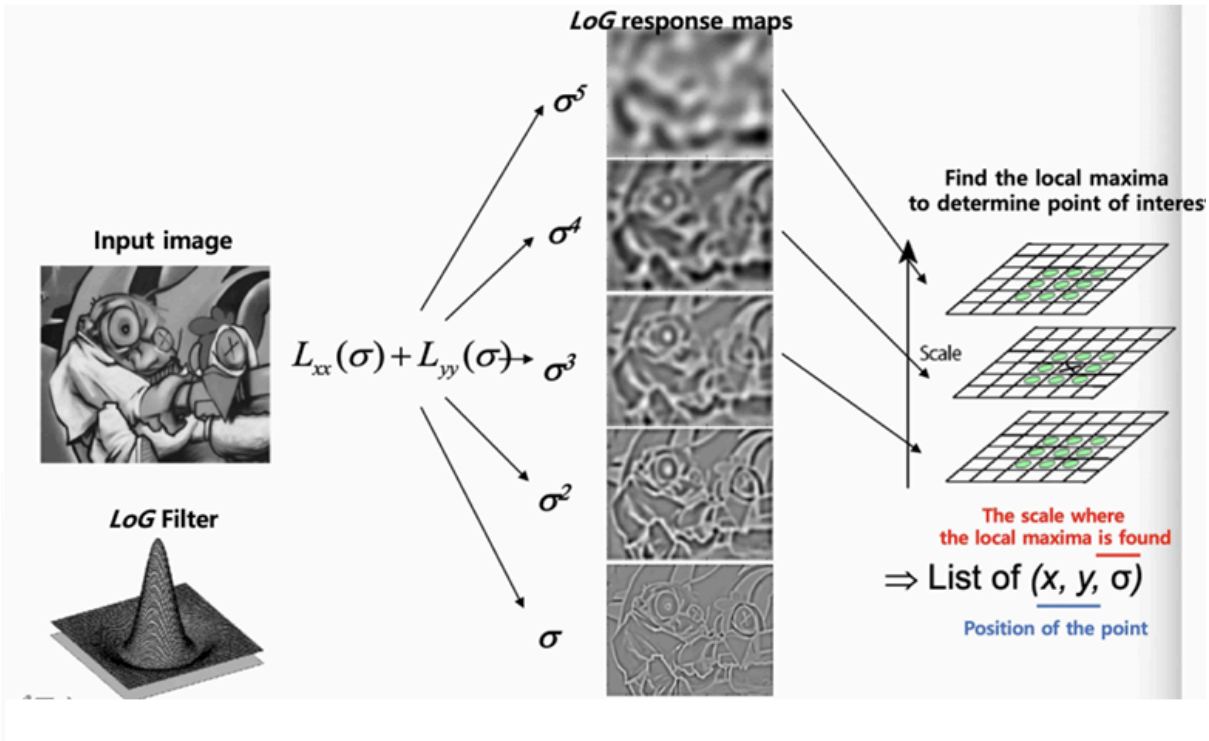
$$\nabla^2_{\text{norm}} = \sigma^2\left(\frac{\partial^2}{\partial x^2}g + \frac{\partial^2}{\partial^2 y}g\right)$$

With this function we compute the response over neighborhoods centered at each location (x,y), and also at different ranges of scales (sigma). So we need to find (x,y, sigma) where the function reaches the local maximum. For doing so, we convolve the blob with the scale-normalized LoG at several scale sigma, and take the sigma with maximum response



This is the approach used in the SIFT detector, that is indeed scale invariant.

Let's see this example:



Here we are applying the LoG filter to the input images, several times, varying the scale sigma. Now, instead of computing LoG for larger and larger windows, we can use a fixed window size with a Gaussian Pyramid.

Feature Description

13/03/2024

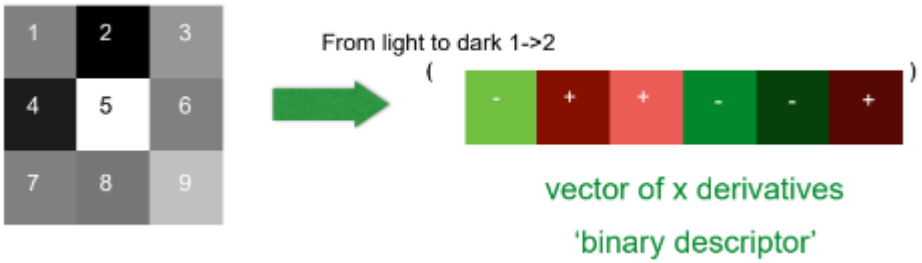
If we know where the good features are, how do we match them?

Attempts to find a Good Descriptor

Let's say we have found our feature (so our corner), this is in the form of a patch, so a matrix of values that are the intensities of the pixels:

The simplest approach is to just use the pixel values of the patch as our descriptor:

This is of course not invariant to absolute intensity value changes: so if you change the intensity value the descriptor will change. This is a problem since maybe the same corner has a different intensity in another image, so the two patches should be recognized by the algorithm with the same descriptor. To solve this we can think of using pixel differences:



This is invariant to absolute intensity values, but if you deform the image the intensity variation will change, so it's not invariant to deformation. We can think of using color histogram, counting the colors in the image:

This is invariant to changes in scale and rotation, since even if you rotate or scale the images the colors will remain the same. The problem is that this is not sensitive to spatial layout: two completely different images can have the same histogram color, and so you will recognize them as equals:

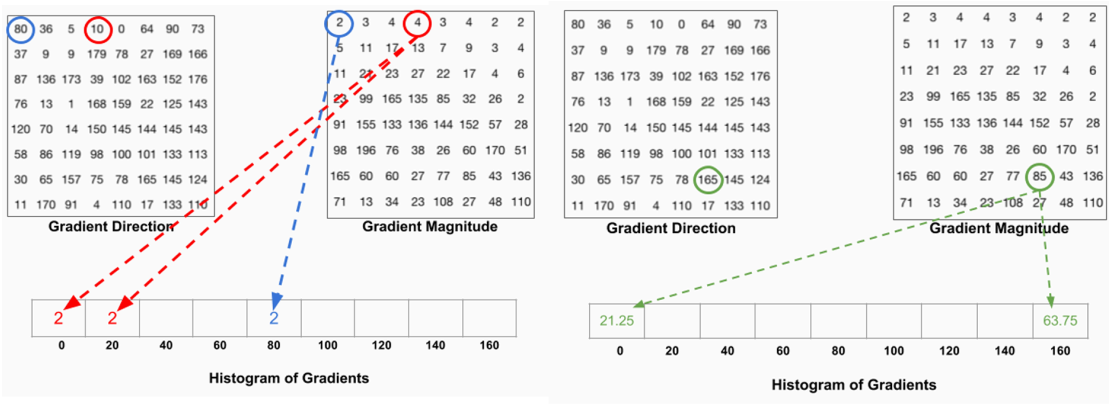
To solve this we can compute histogram over spatial cells in the image. In this way we can retain rough spatial layout. And there is some invariance to deformation:

However, there is no complete invariance to rotation. To solve this we can use the dominant image gradient direction to normalize the orientation of the patch:

So we compute the direction and rotate the patch in that orientation. Basically we are saving the orientation angle theta along with (x, y, sigma). Problems with this approach: not all objects can be identified by their color distribution + the colors change with the illumination (color constancy problem).

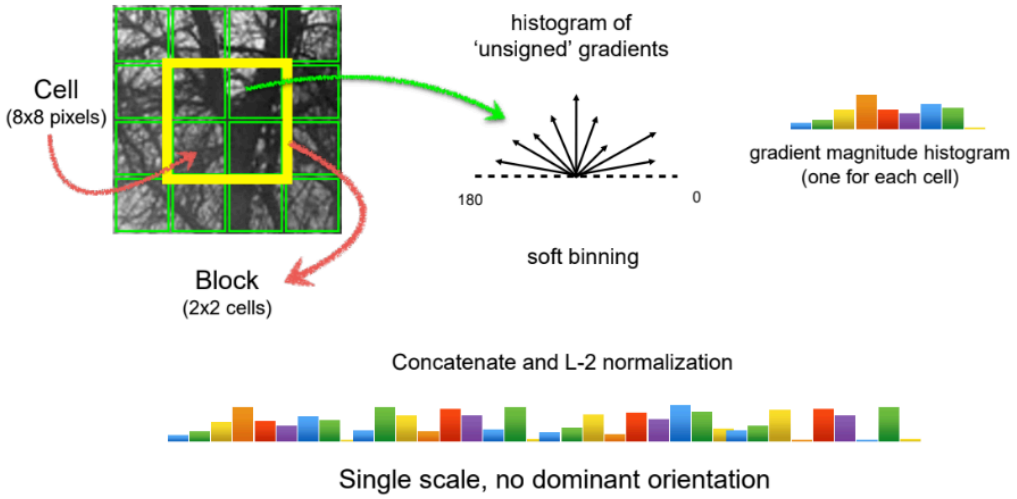
Histogram of Oriented Gradients (HOG) descriptor

1. Calculate horizontal and vertical gradients
2. Calculate magnitude and direction of gradient
3. For each 8x8 cell, calculate the HOG in this way:



We have 9 bins[8]: for each pixel, the bins are selected based on the contribution of the direction and the value that goes in those bins is based on the magnitude.

4. 16x16 Block normalization is used to normalize the histogram with L2 norm[9], in order to not be affected by lighting variation[10].



So now, for the 2x2 block of 8x8 cells chosen, we have 4 histograms that we can concatenate to get a 36 x 1 vector.

5. The 2x2 block window is then moved by 8 pixels and the process is repeated. Note that the step of 1 cell means that there is block overlap.

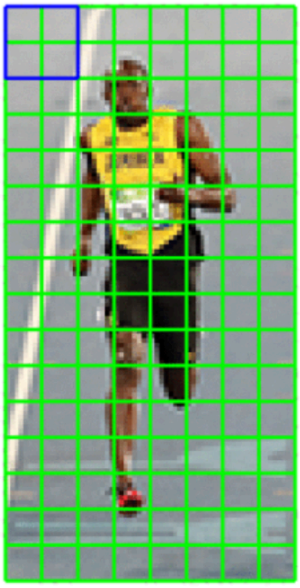
Finally, the HOG feature vector can be computed. Assuming the input image size is 64 x 128 pixels, commonly used, such as pedestrian detection, we have 8 x 16 cells for the image. Since the blocks are 2x2, we have 7 x 15 = 105 blocks. Each block has 9 * 4 = 36 bins, so the HoG feature vector of the image has size 105 * 36 = 3780.

---

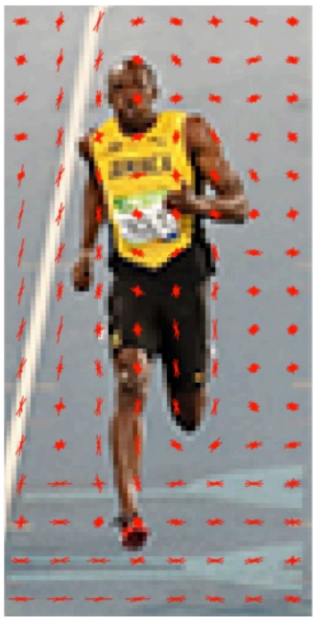
[8] we use angles between 0 and 180, called “unsigned” gradients since a gradient and its negative are represented by the same numbers

[9] Normalizing the vector with the L2 norm means to divide it by sqrt($x^2 + y^2 + \ldots$)

[10] we need this, otherwise, for instance, making the image darker by dividing all pixels values by 2 will change the histogram values dividing them by half.



The HoG descriptor is usually visualized by plotting the dominant direction of the 9x1 normalized histogram in each 8x8 cell.



Pedestrian Detection: train a model (e.g. SVM) over thousands of HoG feature sets. Then, given a new image, obtain the HoG feature vector and let the model predict if it's a pedestrian or not.

Histogram Comparison

How to compare two histograms?

1. Intersection

$$\cap(Q, V) = \sum \min(q_i, v_i)$$

   - Measures the common part of both histograms
   - Range: [0, 1]
   - There is also the formula for unnormalized histogram

2. Euclidean Distance

$$d(Q, V) = \sum (q_i - v_i)^2$$

   - Focuses on the differences between the histograms
   - Range: [0, inf]
   - All cells are weighted equally so it's not very discriminant

3. Chi-square

$$\chi^2(Q, V) = \sum_i \frac{(q_i - v_i)^2}{q_i + v_i}$$

   - Statistical background: test if two distributions are different
   - Range [0, inf]
   - Cells are not weighted equally: more discriminant

Both intersection and chi-square are good.

Feature Matching

For each keypoint that we have detected and described, search similar descriptor vectors in a reference image. Note that a descriptor vector may match more than one reference descriptor vector.

Given a certain feature in I_1 how to find its best match in I_2?

1. First, we need to define a distance function to compare the two. Any distance metric would work, for instance, L2 (Mean Squared Error), L1 (Mean Absolute Error) are commonly used in this context.
2. Now test all the features in I_2 and find the one with minimum distance from the feature in I_1. We can also find the k top matches instead of just 1.

Ratio Test

Let's see an example:

I_1 I_2

A simple approach could be to use L_2 distance: $||f_1 - f_2||$. The problem is that it can give small distances for incorrect matches that are ambiguous, like in the example below:

Since the pattern is repetitive, even if f_2 is the real match, f'_2 is also detected as a good match. To solve this we use the distance ratio = $||f_1 - f_2|| / ||f_1 - f_2'||$, with:

1. f_2 is the best SSD match to f_1 in I_2
2. f_2' is the second best SSD match to f_1 in I_2

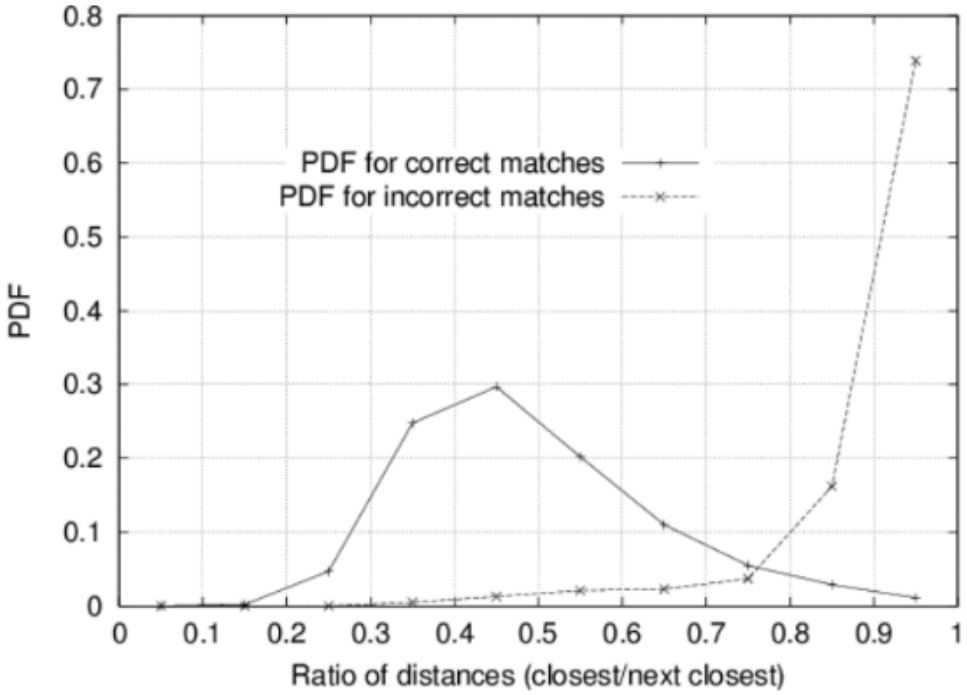
Sorting matches by this ratio puts matches in order of confidence, so the match with high ratio has high confidence[11]. We then discard matches that are ambiguous and retain only the one that respects the threshold. This is called Ratio Test:

- Keep top 2 matches f_2 and f'_2
- If $d(f_1, f_2) / d(f_1, f'_2) < 0.75$ then match f_1 with f_2 and keep the point. Else reject the match f_2 since it's ambiguous.
- Check if the remaining matches $d(f_1, f_2) <$ Threshold, if so keep that as a strong match, else reject the match since, although they are not ambiguous, they are bad

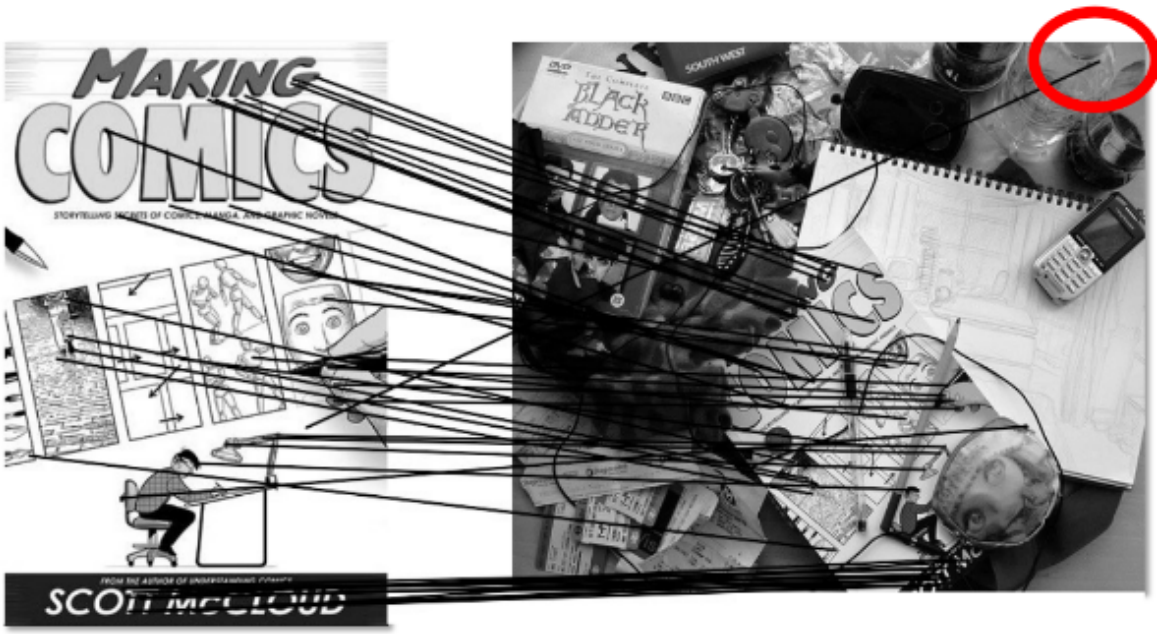
$d(f_1, f_2) / d(f_1, f'_2)$ thresholds in the range [0.75, 0.80] have given good results in object recognition[12]

[11] If, for instance, we set a distance ratio threshold to 0.5, we require that our best match is at least twice as close as our second best match to our initial features descriptor, in fact: $||f_1 - f_2|| / ||f_1 - f_2'|| < 0.5 \rightarrow ||f_1 - f_2|| < 0.5*||f_1 - f_2'|$

[12] As you can see from the graph in fact, using 0.75 as a threshold, about 90% of false matches are above this value, while only 5% of correct matches are above this value. So we discard a lot of bad matches, and a small number of correct matches, as we desire.



RANSAC is used to deal with outliers during the matching:



The one in red is an outlier. We will deal with RANSAC afterwards.

Evaluating Results

True Positive and False Positive

Ok so now we have matched the features. How can we measure the performance of a feature matcher? After choosing the threshold that removes the features, we can count the numbers of True and False Positives:

- TP: number of matches that survive the threshold (so they are below it) and are correct, so they are allegedly and really good features
- FP: number of matches that survive the threshold (so they are below it) but are incorrect, so they were allegedly good features but they are actually wrong.

If a feature matcher has a lot of FP and few TP it's a bad feature matcher.

Let's do a numeric example. Our matcher computes 1000 matches between 2 images, assuming we have a ratio distance of 0.6:

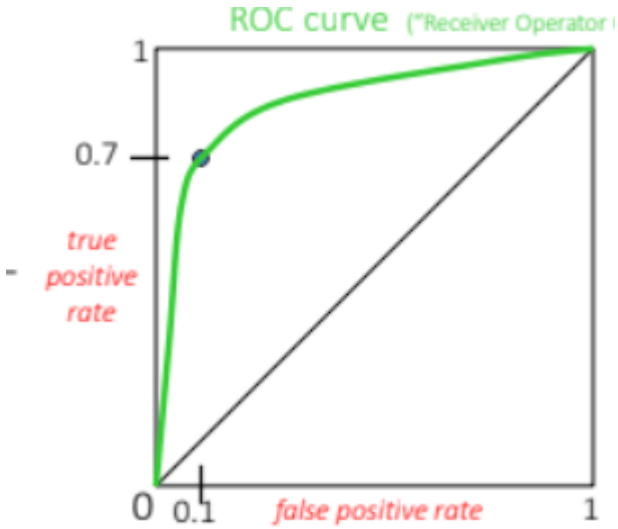
1. Assume the threshold gives us 700 matches that survive the threshold. Assuming we have the ground truth matches:
    a. 100 out of this 700 are incorrect -> FP
    b. 600 out of this 700 are correct -> TP
2. 300 will be discarded, it will contain both FN and TN

If we know the ground truth matches, and, let's say, we know that 800 of those 1000 matches are correct matches, we have that:

- TPR = number of survivors that are correct / number of correct matches = 600 / 800 = ¾
- FPR = number of survivors that are incorrect / number of incorrect matches = 100 / 200 = ½

These two metrics can tell us the performance of the feature matcher. In order to maximize TP, we have to modify the value of the ratio distance threshold

ROC and AUC



This ROC curve is generated by choosing a different ratio distance threshold, which leads to different sets of matches with different TRP/FPR. Then we can compute the Area under the

Curve (AUC) to understand the performance of the feature pipeline: the higher the AUC the better.

Accuracy

The simplest measure of performance is the fraction of items that are correctly classified, so

$$\frac{tp + tn}{tp + tn + fp + fn}$$

But this measure is not optimal for an unbalanced set of data. For instance if 5% of the matches are truly positive, then a feature pipeline that always says “negative” is 95% accurate.

Confusion Matrix

The confusion matrix is this one

Estiamted class \ Actual class	1 (p)	0 (n)
1 (Y)	True positive (TP)	False positive (FP)
0 (N)	False negative (FN)	True negative (TN)

Precision, Recall and F1 Score

Precision tells us how much we are precise in the detection[13]

$$\frac{\text{True Positive}}{\text{\# Estimated Positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

[13] Of all patients that we have classified with disease, what fraction actually has the disease?

While Recall how much we are good at detecting[14]

$$\frac{\text{True Positive}}{\text{\# Actual Positive}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

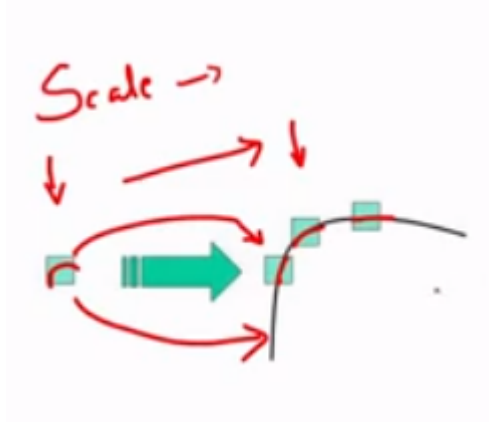
F1 Score is the harmonic mean of Precision and Recall

$$2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

Scale Invariant Feature Transform (SIFT)

19/03/2024

Problem:

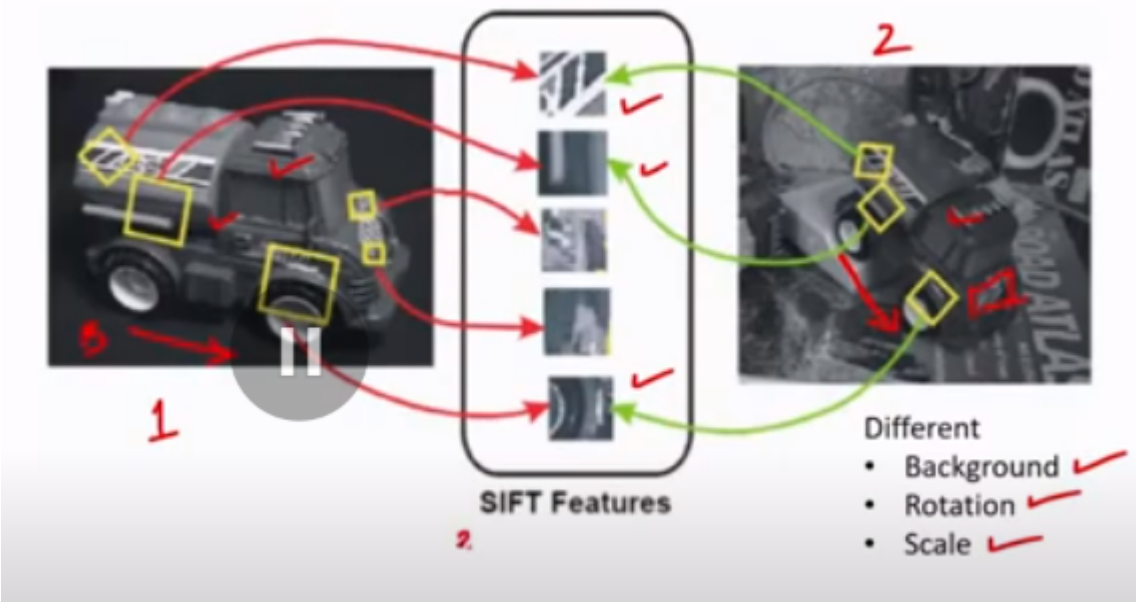


the corner, if scaled, is recognized as edge from the patch.

We need a Scale Invariant algorithm.

Let's look at this example. You have two images in which there is the same object but rotated, scaled and with a different background. The SIFT algorithm was able to identify, in the second figure, 3 out of the 5 features present in the second figure (the other two were not found in the second image probably because they were too small), so the algorithm says that the objects in the two images are the same with a probability of 60% - 70%

[14] Of all patients that have the disease, what fraction did we correctly detect as having the disease?



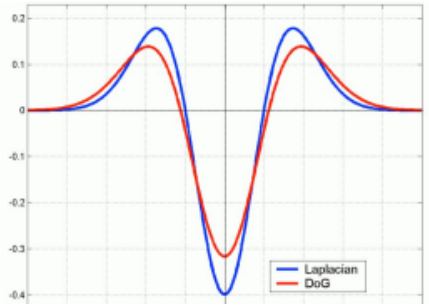
The problem is that we don't know in advance what is the scale of the object in a new image, so we need to explore the features of the new image in different scales. We can use SIFT, that uses DoG, that is an approximation of NLoG[15] used in the generic blob detector.

$$L = \sigma^2 \left(G_{xx}(x,y,\sigma) + G_{yy}(x,y,\sigma) \right)$$

(Laplacian)

$$DoG = G(x,y,k\sigma) - G(x,y,\sigma)$$

(Difference of Gaussians)



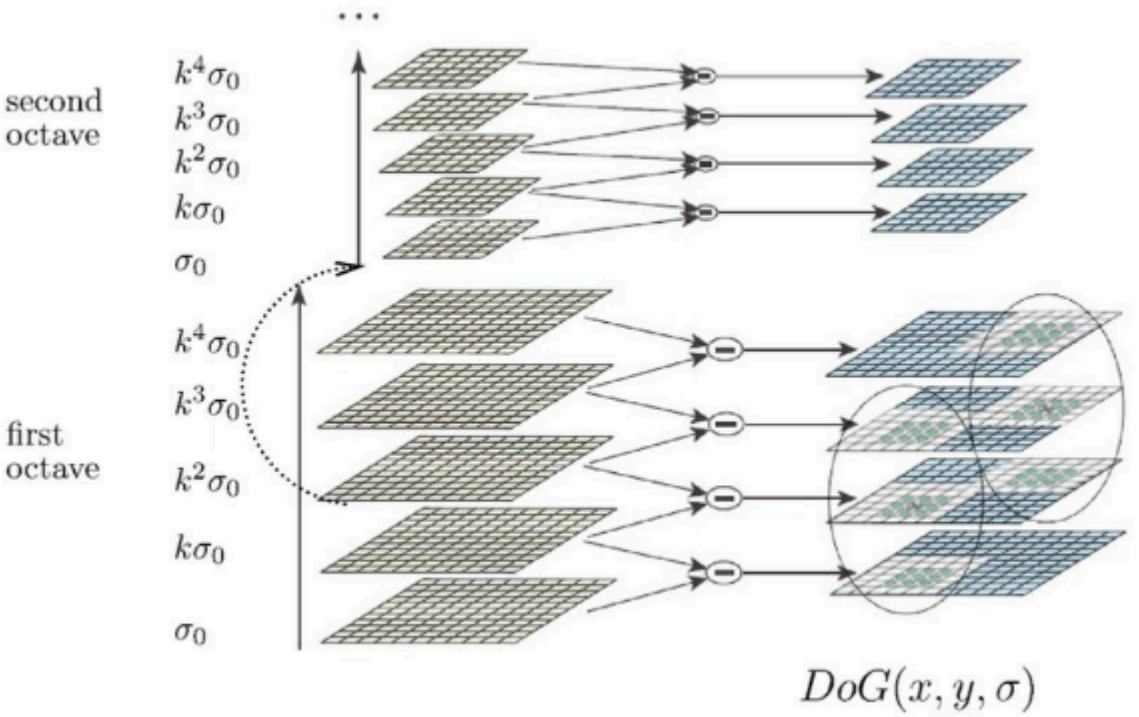
Let's go into the details of SIFT. SIFT describes both a detector and a descriptor. It maximizes the DoG in scale and in space in order to find the same key points independently in each image. SIFT is robust to changes in scale, rotation and illumination-

Sift Detector

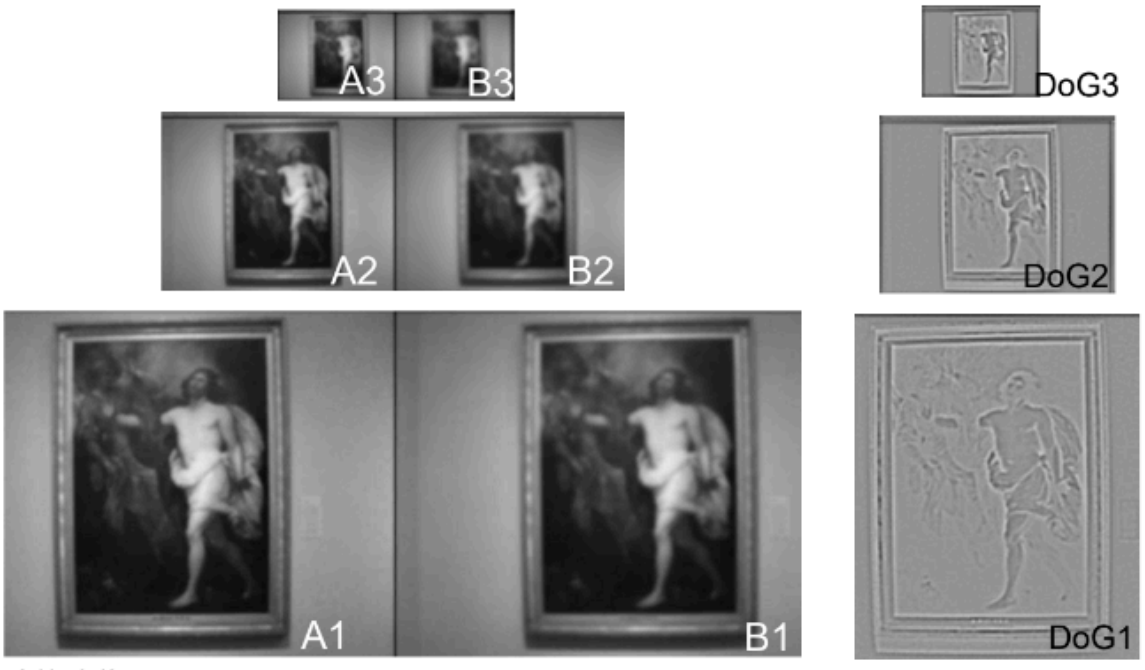
Scale-Space Extrema Detection

We compute the DoG via an image pyramid. The pyramid is composed of different octaves. Each octave is composed of layers. Ascending to the upper layers of the octave we progressively blur the image. To go to the next octave we downsample the image by a factor of 2. From the gaussian pyramid, we subtract each pair of layers to get the DoG pyramid.

[15] Normalized Laplacian of Gaussian



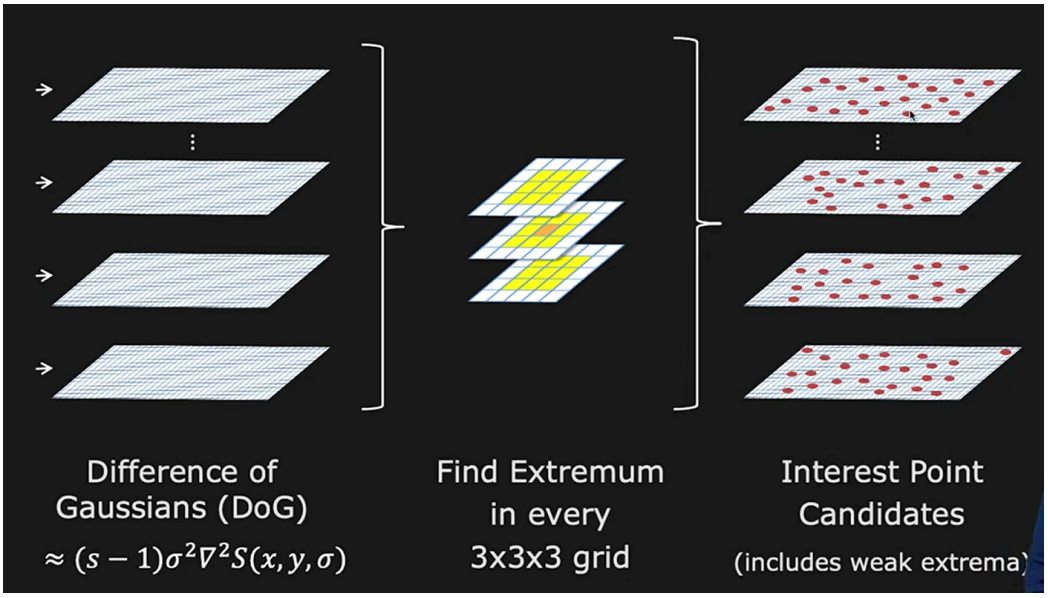
An practical example is the following:



A1 and B1 belong to the same octave, then we have A2 and B2, A3 and B3.

Keypoint Localization and Illumination Thresholding

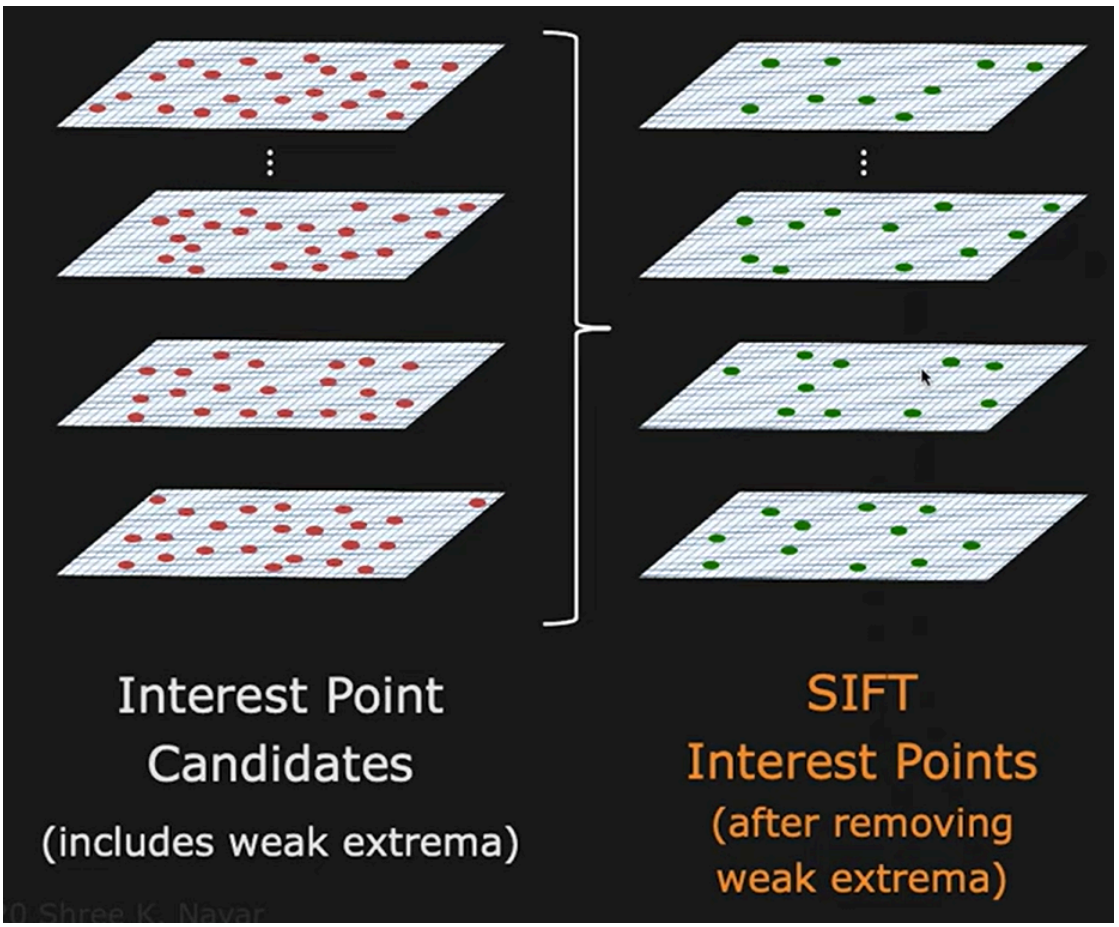
Now that we have the DoG pyramid, we need to find the extrema in space (x,y) and scale (sigma). A way of doing it is with a 3D window (a cube), moving it around trying to find the extrema. So, if the cube is 3x3x3, we look for points that are larger than all the neighboring points in the 3x3x3 window. There are of course other ways of finding peaks (both in 2D and in 3D, as in this case).



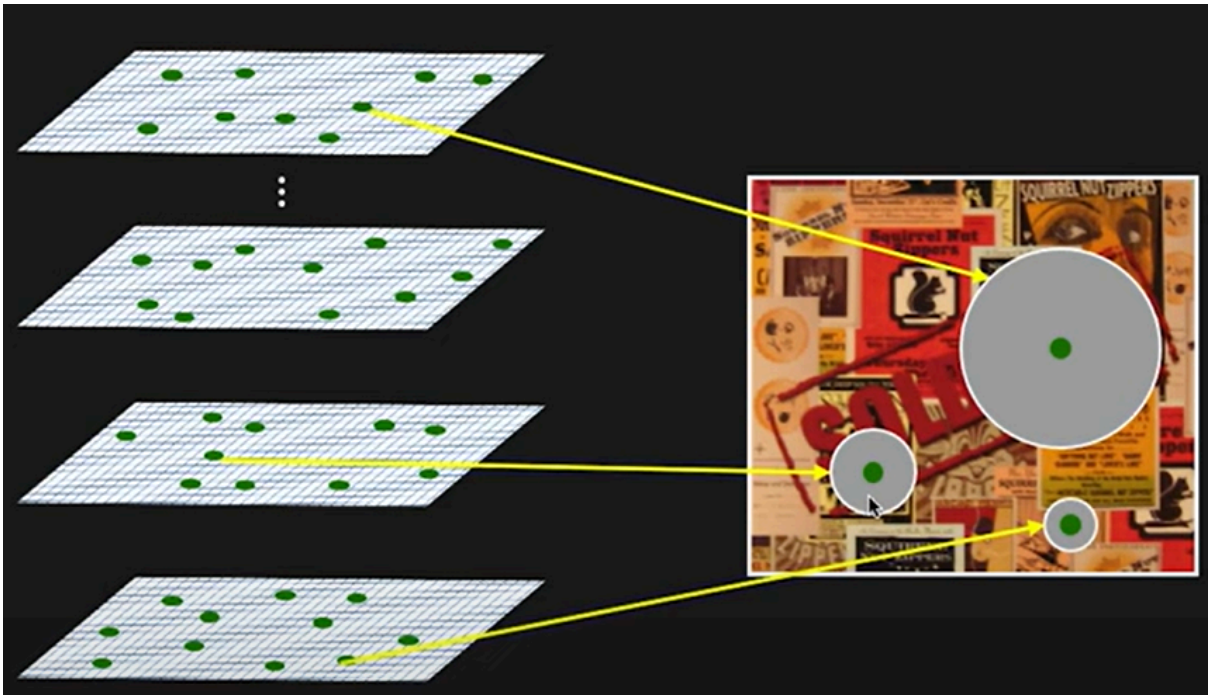
After having done that, we have interest points **x** = {x, y, sigma}, that we clean up removing weak estrema:

- First we filter by illumination thresholding
- Then we do gradient-based filtering: keypoints at low gradient regions or along weak edges are discarded

Now we have only strong, stable keypoints.



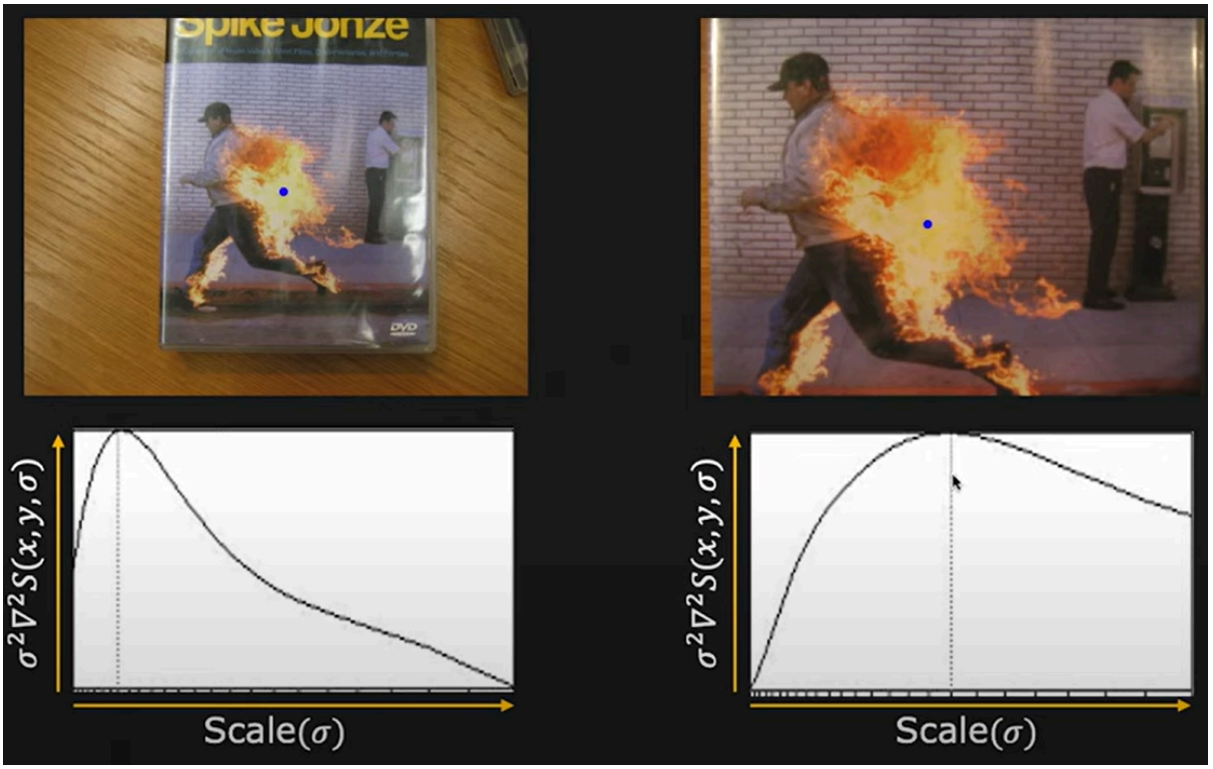
These are now our SIFT features. These features will be in different points (x,y) of the images, but also at different scales (sigma). That can be represented in this way:



As you can see from the photo, higher characteristic scale (sigma) of a point corresponds to a larger blob in the image. All the local features are here:

Scale Invariance

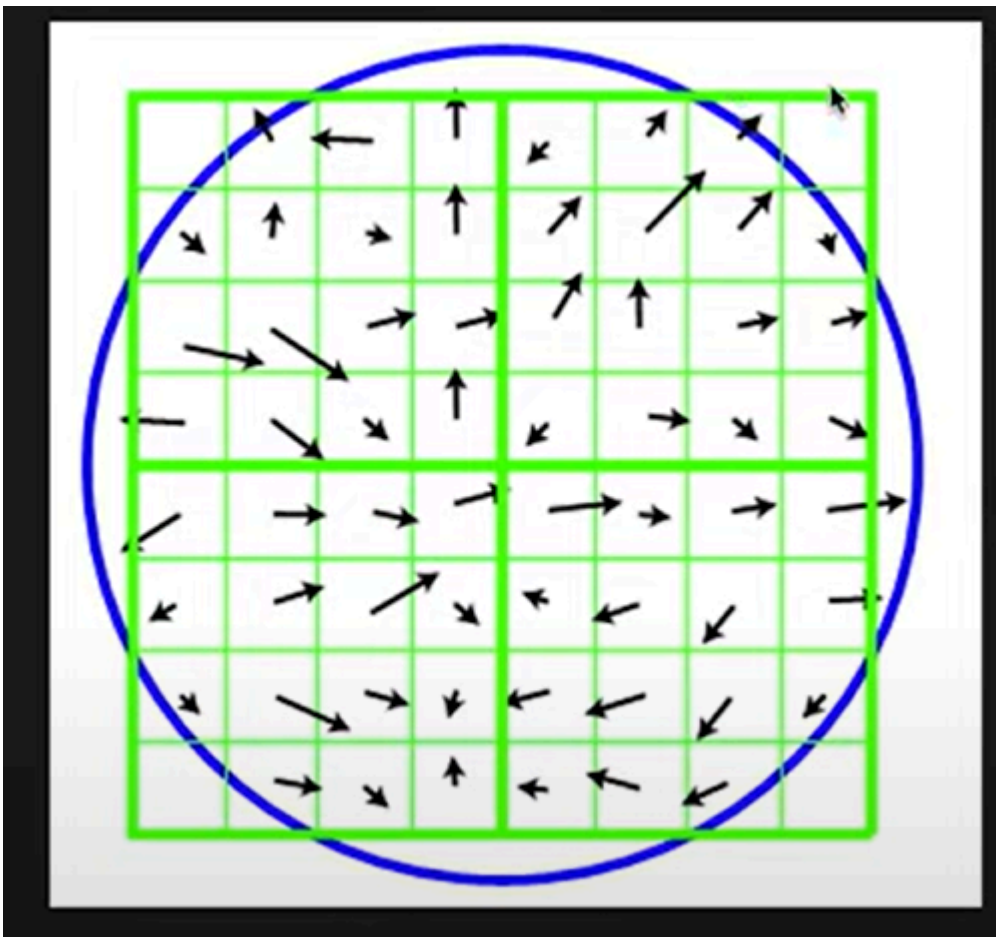
This algorithm provides scale invariance. In fact, consider these two objects below in two different images, one more zoomed. The same keypoint is localized in both images, but obviously at different scales. That means that the NLoG/DoG operator peaks at different sigma.



Because we know this sigma, thanks to what we have done until now, we can normalize, adjusting the scale of one to match the other

Keypoint Orientation Assignment

Now that we have found our keypoints (x,y, sigma), we have to assign an orientation to each of them. Assigning an orientation, we provide rotation invariance, in the same way as assigning a sigma we provide scale invariance. For doing so, we compute gradient magnitude and orientation for each keypoint.



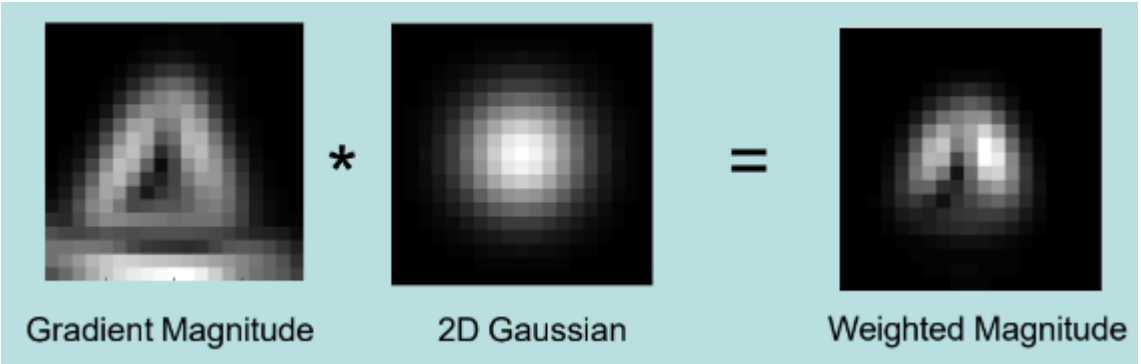
Imagine that this above is the area that the blob occupies. We can place a grid in there and for each pixel in the grid we compute the gradient, that gives us the magnitude and orientation of the edge in that pixel.

Image gradient directions

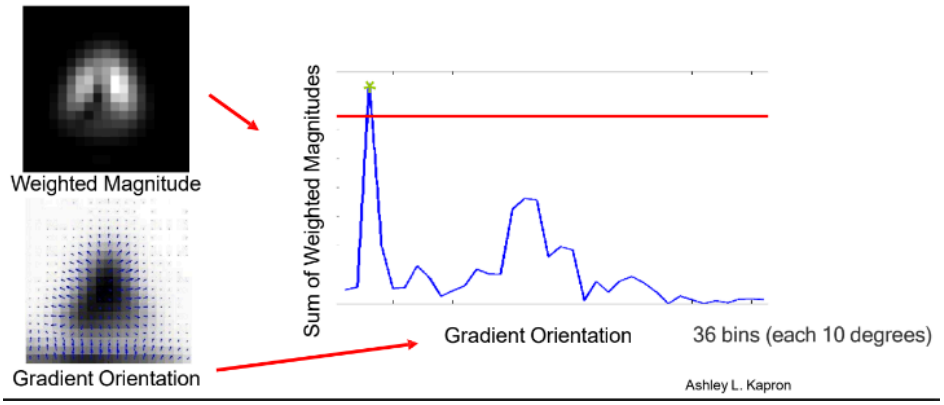
$$\theta = \tan^{-1}\left(\frac{\partial I}{\partial y} / \frac{\partial I}{\partial x}\right)$$

Now we want to find the principal orientation within these pixel orientations, in order to assign it to the entire blob. For doing so we create a histogram in which each bin is a different direction. The peak of the histogram will be the principal orientation of the blob.

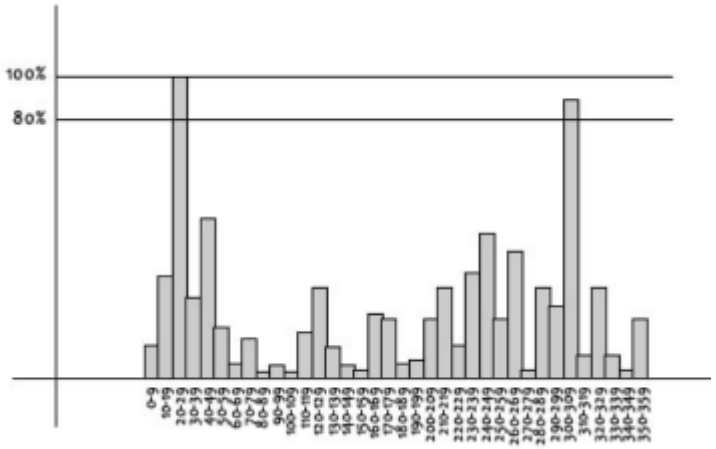
This is practically how we create the histogram: first let's get the weighted magnitude of the blob



Then we build an histogram of orientations in terms of sum of weighted magnitude. Precisely, the 360 degrees of orientation are broken into 36 bins each of 10 degrees. This is for the x axis. How is the y axis calculated? We put the sum of weighted magnitude on the y axis. We compute the histogram considering the keypoints and all the pixels around it.



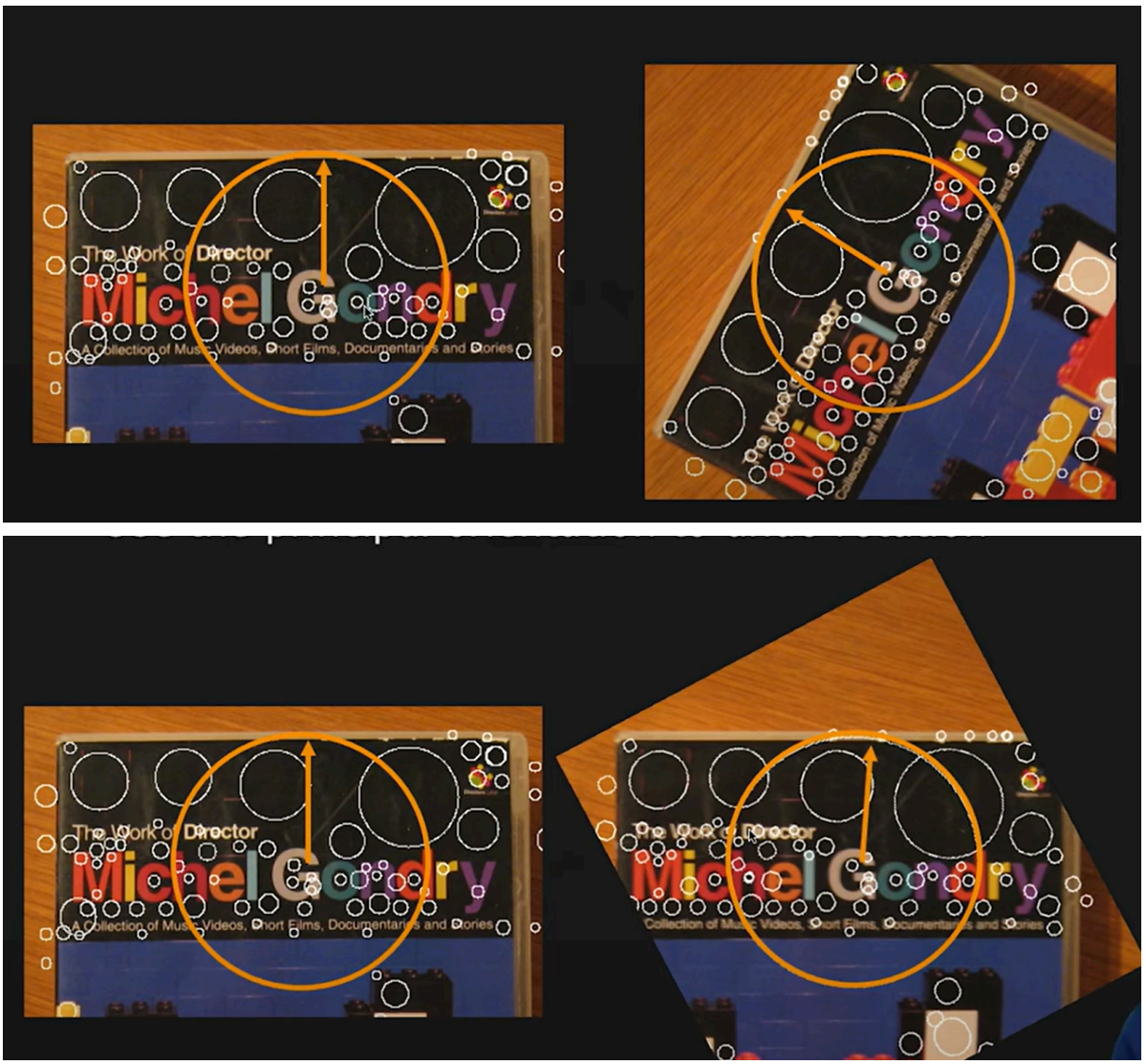
We obtain the graph above. We can clearly identify the peak in the histogram. Considering the case below, we see that the histogram peaks in the third bin (20-29 degrees). That means that summing up all the weighted magnitudes that correspond to orientations that are in the range [20-29], we see that this sum is the maximum of all the image.



This peak orientation and the corresponding sum of magnitude are assigned to the keypoint. Moreover, peaks above 80% of the highest peak are converted into new keypoints. These new keypoints have the same location and scale (x,y,sigma) as the original, but the orientation is different, since it's the one corresponding to the bin that has peaked. This means that **orientation can split up one keypoint into multiple keypoints**.

Rotation Invariance

Why is this providing us rotation invariance? Because given two images representing the same object but rotated differently, wecan undo the rotation since we know the principle orientation of the feature

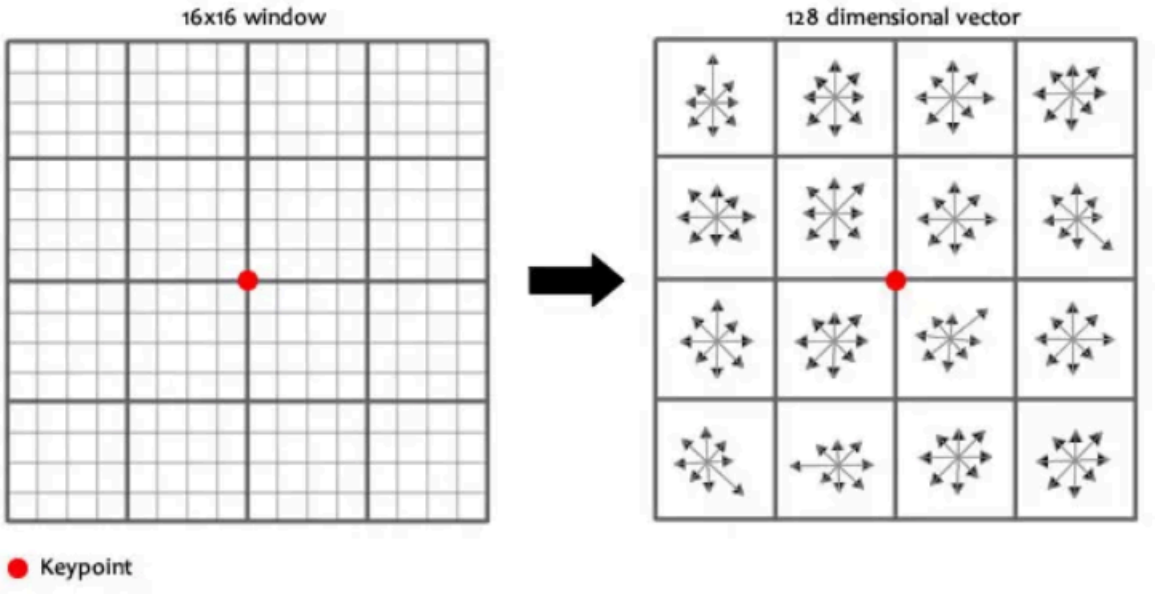


SIFT Descriptor

Now that we have assigned location (x, y), scale (sigma) and orientation (theta), we can say we have successfully detected the point (x,y, sigma, theta). Now we want to describe it. To build the descriptor those are the steps:

1. Take the gaussian blurred image associated with the keypoint's scale. This will ensure invariance to scaling.
2. Take the image gradient over a 16x16 square windows around the detected feature in position (x,y)
3. Rotate the gradients in the 16x16 neighborhood in order to align with the dominant orientation (theta). In this way the descriptor will be invariant to image rotation.
4. Now divide the 16x16 window into 4x4 cells

5. Compute an orientation histogram with 8 bins for each 4x4 cell, using the sum of the weighted gradient magnitude
6. The SIFT descriptor is a length 128 vector, since we have 4 x 4 = 16 histograms each with 8 bins -> 16 * 8 = 128.



Comparing SIFT Descriptors

We are essentially comparing two array of data, so we can do it with:

- L2 Distance: subtracting them bin for bin

$$d(H_1, H_2) = \sqrt{\sum_k \left(H_1(k) - H_2(k)\right)^2}$$

- Normalized Correlation: if it's 1, you have a perfect match

$$d(H_1, H_2) = \frac{\sum_k [(H_1(k) - \bar{H}_1)(H_2(k) - \bar{H}_2)]}{\sqrt{\sum_k (H_1(k) - \bar{H}_1)^2} \sqrt{\sum_k (H_2(k) - \bar{H}_2)^2}}$$

where: $\bar{H}_i = \frac{1}{N} \sum_{k=1}^{N} H_i(k)$

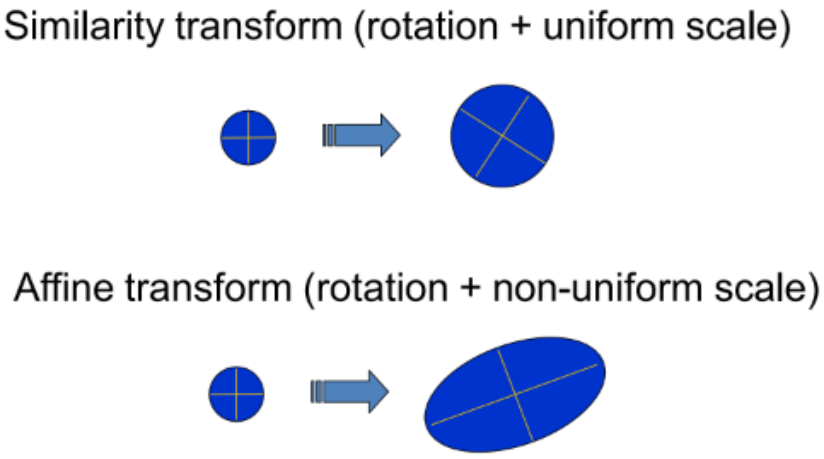
- Intersection: larger the intersection, better the match

$$d(H_1, H_2) = \sum_k \min\left(H_1(k), H_2(k)\right)$$

Properties of SIFT

The SIFT descriptor properties:

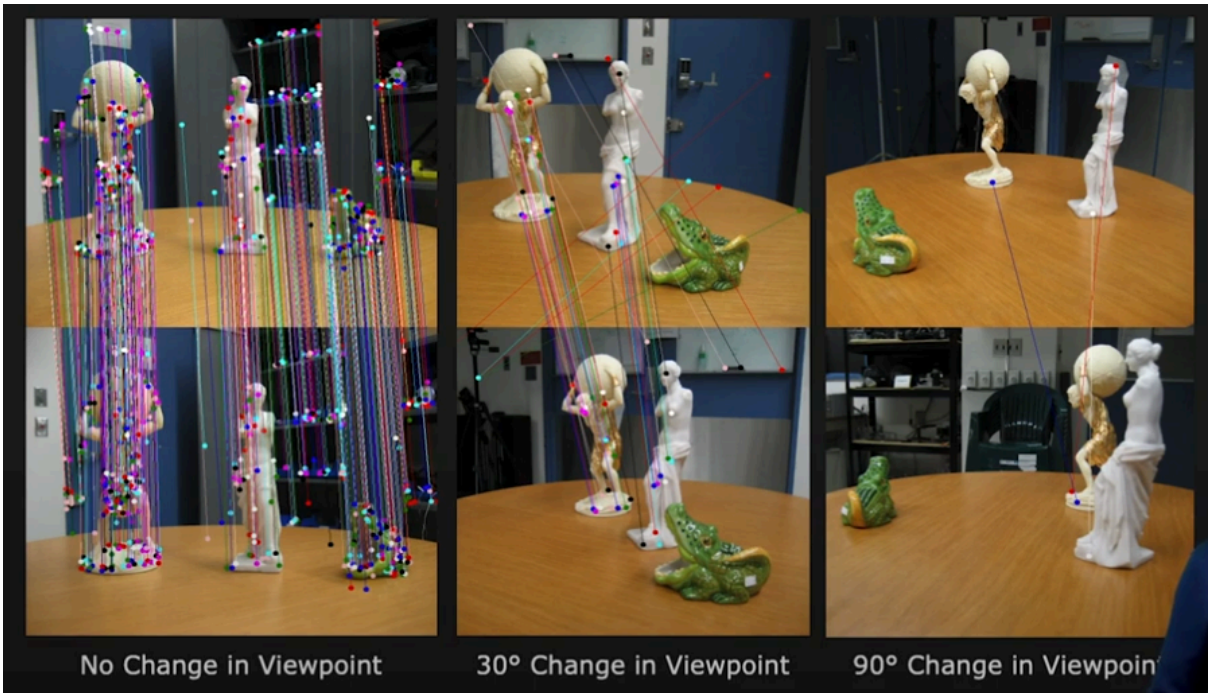
1. Invariant to scaling
2. Invariant to rotation
3. Robust to clutter and partial occlusions
4. Invariant/robust to illumination variation since we worked with the orientation not taking into account the magnitude of the gradient
5. Slightly robust to affine transformation and noise (empirically found)



We can do a stability test, rotating, scaling, changing intensity and transforming in order to see how much keypoints survive in percentage.

SIFT fails for large variation of the viewpoint

SIFT is reliable for only small changes in viewpoint, so you can't use it for 3D objects



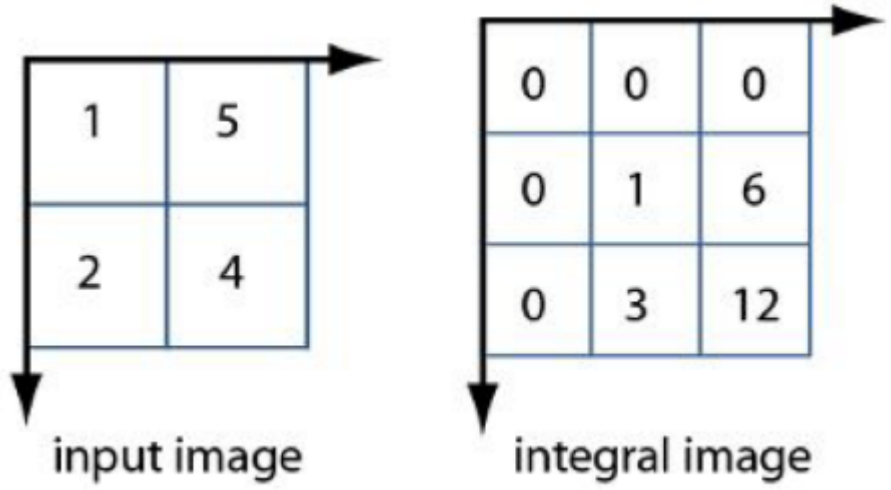
Speeded-Up Robust Features (SURF)

Is a feature detection framework inspired by SIFT but faster.

Integral Images

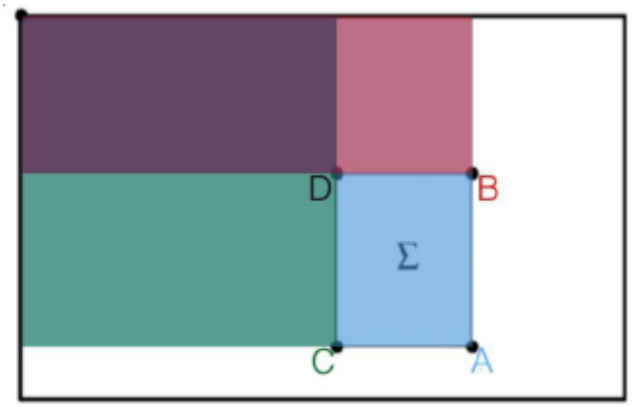
The value of the integral image in a point **x** = (x, y) is the sum of all the points in the rectangle that goes from the origin to **x** = (x, y)

$$I_{\Sigma}(\mathbf{x}) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i,j)$$



We use integral images since they are efficient in calculating regions of the image, *and so efficient to detect blobs*: just 4 memory accesses and three operation

$$\Sigma = A - B - C + D$$



SURF Detector

Scale-Space Extrema Detection

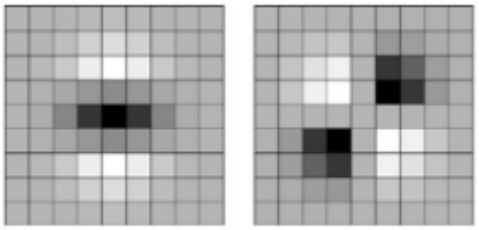
SURF detects blob-like structure at locations where the determinant of the Hessian is maximum.

$$H(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix}$$

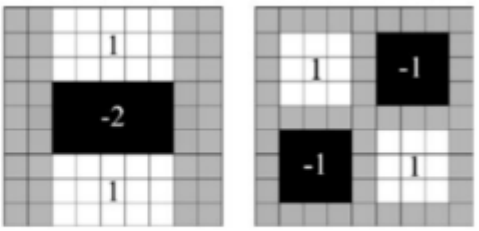
Where L_{xx} is the convolution of the Gaussian second order derivative with the image I in point **x.** Computing the Hessian is slow, so we can approximate this procedure using box filters:

$$det(H_{approx}) = D_{xx}D_{yy} - (wD_{xy})^2$$

Where D_{xx} is the approximation of the Gaussian second order partial derivative in the x-direction



The Gaussian second order partial derivative in y- and xy-direction.



Box filter approximations of the Gaussian second order partial derivatives.

Using box filters instead of LoG/DoG simplifies the computation a lot.

To match points across different scales we build a pyramidal scale space. So we find the maximum of the hessian determinant both in scale and in space.

In this way we can capture keypoints of varying sizes.

Orientation Assignment

In order to achieve rotation invariance, SURF assigns an orientation to each keypoints based on the dominant direction of the gradient around the keypoint. This is done by calculating the Haar wavelet responses in x and y directions in a circular neighborhood of the keypoint and summing the responses to find the dominant orientation.

Feature Vector Extraction - Keypoint Descriptor

Now we know the dominant orientation of the keypoint. To extract the feature vector we consider a square region centered around the keypoint and oriented along the dominant orientation found in the previous step. This region is divided into square sub-region and for

each of them Haar wavelet responses are computed. The responses are summed up to create the descriptor that represents the intensity changes around the keypoint

$$\mathbf{v} = \begin{bmatrix} \Sigma d_x \\ \Sigma d_y \\ \Sigma |d_x| \\ \Sigma |d_y| \end{bmatrix}$$

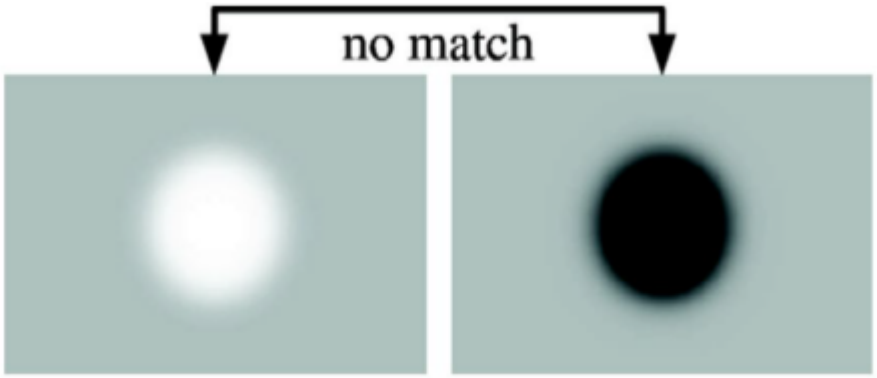
With d_x being the “horizontal” Haar wavelet response while d_y is the “vertical”. This is a 64 dimensional vector, whereas SIFT uses a 128 dimensional vector.

Keypoint Matching - Laplacian Indexing

With keypoints detected and described, now we need to match keypoints between images. The SURF algorithm, for fast indexing during the matching phase, uses the sign of the Laplacian to distinguish between:

- bright spot: positive laplacian, indicating that the keypoint is located in a bright region with dark surrounding
- dark spot: negative laplacian, indicating that the keypoint is located in a dark region with bright surrounding

and it’s a meaningful metric to divide the set of all interest points. In fact, when searching for matches between keypoints in two images, only keypoints with the same laplacian sign are compared, halving the number of comparisons needed.



SURF vs SIFT

In the approach:

- SIFT uses DoG for scale-space representation and orientation histogram for descriptor

- SURF uses Hessian for scale-space representation, using Integral images for faster computation of the Hessian response, and wavelet in horizontal and vertical directions for orientation assignment and descriptor

In the computational efficiency:

- SIFT is computationally intensive due to the creation of the pyramid and the orientation histograms, although is more robust and accurate than SURF
- SURF is faster since it uses integral images and wavelets, although is less accurate in some scenarios

In the descriptor size:

- SURF descriptors are 128-dimensional, whereas SURF descriptors are 64-dimensional, making SURF more efficient in terms of storage

Binary Descriptors

What's the problem with SIFT

- Expensive to compute
- Patenting Issue

So even if SIFT works well and is a gold standard we want something easier. Binary descriptors generate small binary strings that are easier to compute and compare.

The idea behind is simple:

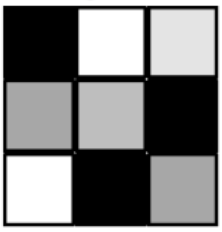
- Select a patch around a keypoint
- Select a set of pixel pairs in that patch
- For each pair compare the intensities:

$$b = \left\{ \begin{array}{ll} 1 & \text{if } I(s_1) < I(s_2) \\ 0 & \text{otherwise} \end{array} \right\}$$

- Concatenate all the b's to a bit string

Consider this example:

image area



index (for pairs)

1	2	3
4	5	6
7	8	9

pairs: $\{(5,1),\ (5,9),\ (4,6),\ (8,2),\ (3,7)\}$

tests: $b=0 \quad b=0 \quad b=0 \quad b=1 \quad b=1$

result: $B = 00011$

Then, to compare two images, we just compare their descriptors. We must:

1. Use the same pairs in both images
2. Maintain the same order in which the pairs are tested

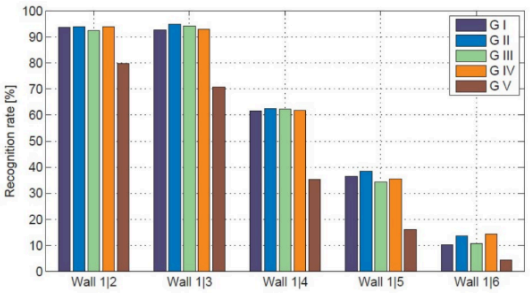
Advantages:

- Compact descriptor, is long as the number of pairs that we chose to consider
- Fast to compute: just an intensity value comparison
- Trivial and fast to compare: using the Hamming distance you can compare two binary strings

$$d_{\text{Hamming}}(B_1, B_2) = \text{sum}(\text{xor}(B_1, B_2))$$

The first binary image descriptor was Binary Robust Independent Elementary Features (BRIEF), with 256 bits for the descriptor. To deal with noise the operations are performed on a smoothed image. BRIEF provides five different geometries as sampling strategies.

- G I: Uniform random sampling
- G II: Gaussian sampling
- G III: s_1 Gaussian; s_2 Gaussian centered around s_1
- G IV: Discrete location from a coarse polar gird
- G V: s_1=(0,0) ; s_2 are all location from a coarse polar gird



Oriented FAST and Rotated BRIEF (ORB)

Is an extension to BRIEF that:

- adds rotation compensation
- learns the optimal sampling pairs

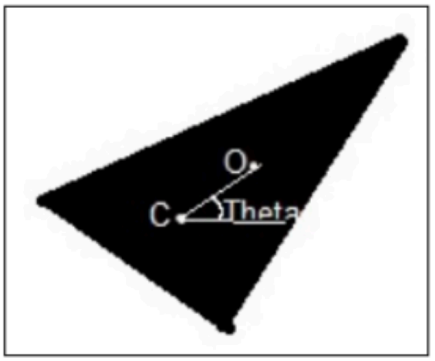
Rotation Compensation

ORB estimates the center of mass and the main orientation of the area/patch, using the image moment[16].

$$m_{pq} = \sum_{x,y} x^p \, y^q \, I(x,y)$$

From this, we compute

- Center of mass $C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$
- Orientation $\theta = \text{atan2}(m_{01}, m_{10})$



Now that we know C and theta, we can rotate the coordinates of all pairs around C by an angle theta.

$$s' = T(C, \theta) \, s$$

Now we can use the transformed pixel coordinates, having achieved invariance to rotation in the plane.

Learning Sampling Pairs

Pairs should:

- be uncorrelated: each new pair adds new information to the descriptor
- have high variance: it makes a feature more discriminative

We want to optimize in this sense the selection of the 256 pairs using a training database

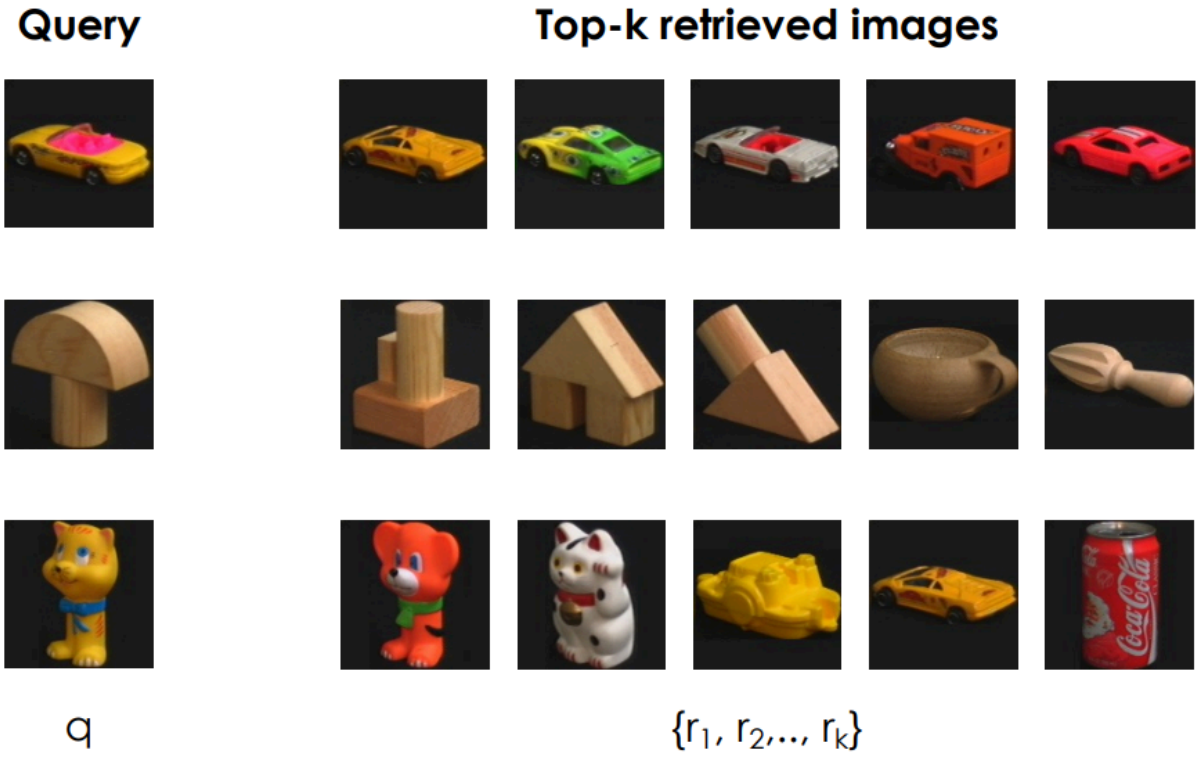
ORB vs SIFT

1. ORB is 100x faster than SIFT
2. ORB uses only 256 bit, while SIFT 4096 (128 * 4 * 8 bit)
3. ORB is not scale invariant, but is achievable via an image pyramid
4. ORB is rotation invariant in the plane
5. ORB has similar matching performance as SIFT

[16] p and q are non-negative integers representing the order of the moment

Image Retrieval

Given an image we want to search in the databases images that are close to the considered:



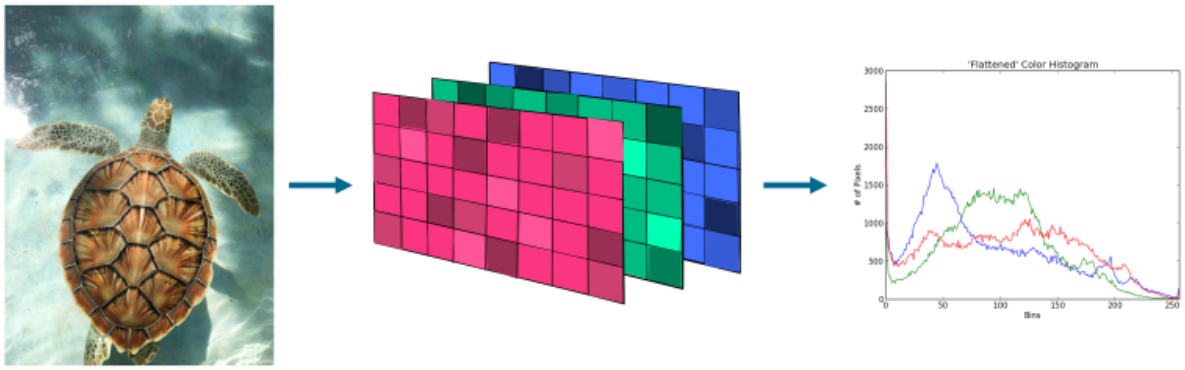
Challenges: background clutter, illumination, deformation, occlusion, intraclass variance.

For doing image retrieval, we need to consider:

1. What is the measure of similarity?
2. How do we represent the images?

Color Histogram

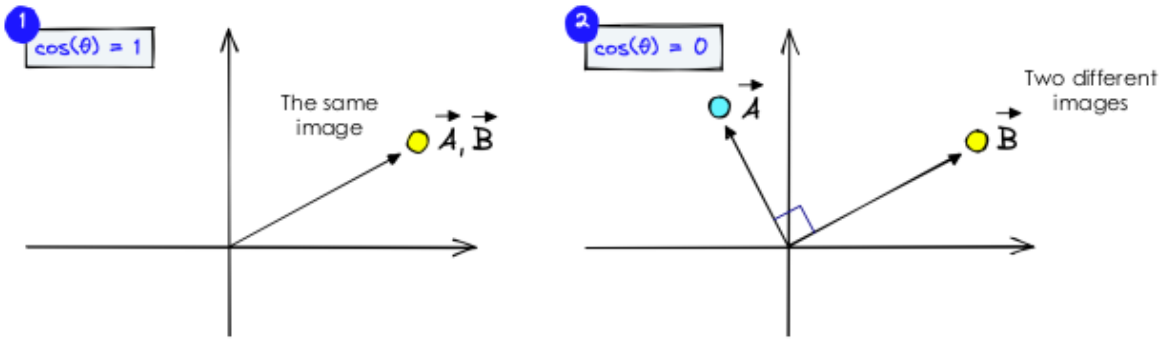
Given an RGB image, we can represent it with three matrices. Every pixel in every matrix has a value in [0, 255]



Each matrix can be visualized as an histogram, with bins representing intensity ranges and their height being how many pixels of the matrix lies in that intensity range. Considering that an histogram can be represented as a vector, for an RGB image we have three vectors that we can concatenate, getting a single vector. Now we have a vector from an image, how can we compare vectors in order to compare images?

$$cos(\theta) = \frac{A \cdot B}{||A||\ ||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sum_{i=1}^{n} B_i^2}$$

The smaller theta, the similar two vectors are. In fact for theta = 0 -> cos(theta) = 1 implies that we are dealing with the same image, while theta = pi/2 -> cos(theta) = 0 meaning the two are very different images.



In [this repository] you can find an exercise for color histograms.

Bag of Visual Words (BOVW)

We will extract two things:

- keypoints, invariant to scaling and rotation
- descriptors: vector representation of an image patch obtained from feature detector such as SIFT, ORB or SURF

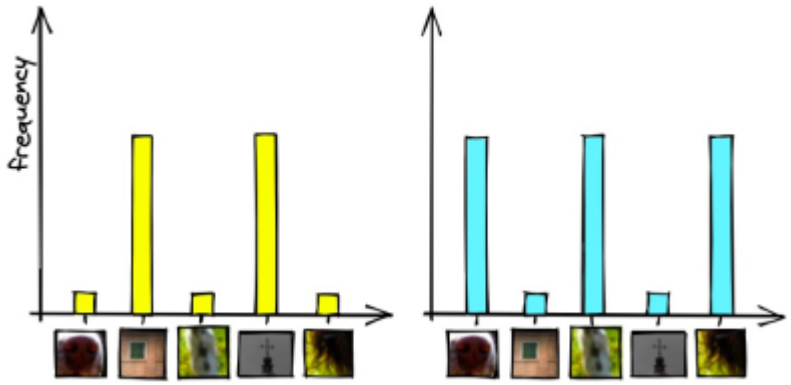
Vector Quantization

Consider the following case:

Are all these features really relevant? No, because they are representing the same thing: the dog's nose. How can we reduce the number of visual features representing the same object by using only one feature for it? Namely, how can we perform vector quantization? Using k-means[17]: the algorithm will iterate through each visual feature and check which centroid (visual word) is nearest. At the end of the algorithm, we will have that all the visual features will be associated to the nearest of the k centroid (visual word).

Now we have a predefined number (k) of visual words. The set of all visual words is called codebook, dictionary or vocabulary. Now, having a codebook of visual words, we can count how many times a certain visual word occurs in each image, building a frequency vector for each image. We can represent the frequency vector with an histogram:

[17] if k is too small, visual words could be not representative of all image regions, if k is too large, there could be too many visual words with little-no of them being shared between images (making comparisons very hard or impossible)



Not all the visual words will appear in all the images (e.g. a picture of a dog will likely not contain visual words of a window). So we have a sparse vector: almost all 0 apart for some values.

Tf-idf (Term frequency - inverse document frequency)

Are all visual words equally important? No of course, for instance:

Patch 1 doesn't tell us anything useful to detect that the image represents a church. To solve this, what we can do is to reduce the importance of visual words that are more present, since it means that they are not able to distinguish objects. For doing so we use the Tf-idf (term-frequency inverse document frequency) function:

$$tf\text{–}idf_{t,d} = tf_{t,d} * idf_t = tf_{t,d} * log\frac{N}{df_t}$$

Number of times that visual word t occurs in the image d

Number of images containing visual word t

That reduces the importance of visual words that are very present. We have:

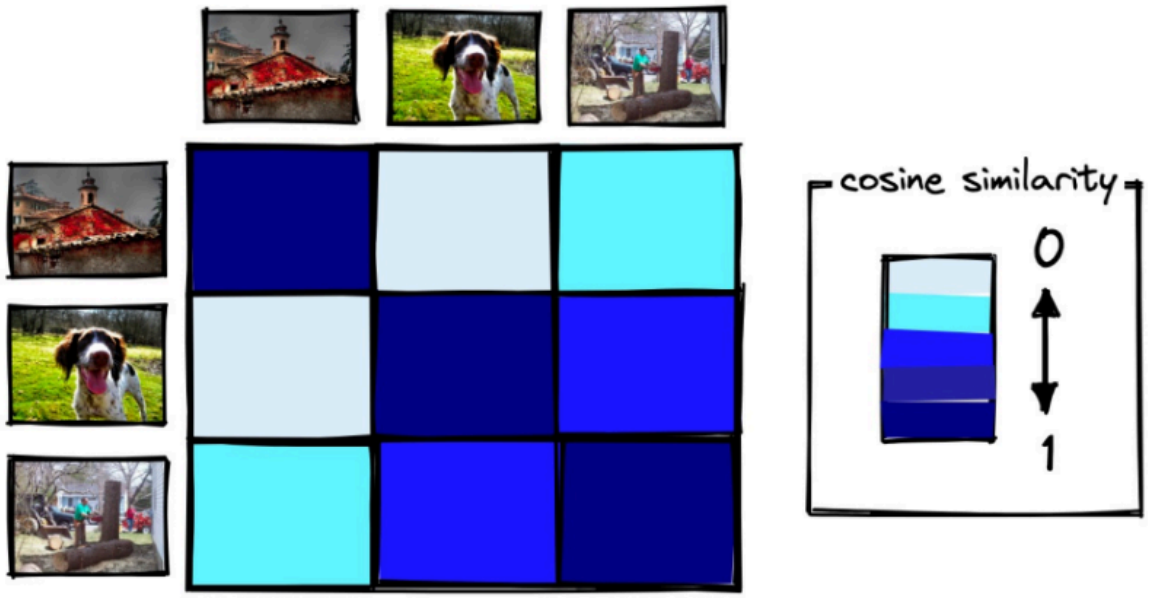
- $tf_{t,d}$ that is the number of times that visual word t occurs in d, extractable from the frequency vector
- N that is the total number of images
- df_t that is the number of images containing visual words t
- $log(N/df_t)$ that measures how common visual word t is across all images in the database. So it's low if the visual word d occurs in many images, high otherwise.

The logarithmic term ensures that the impact of idf_t doesn't grow linearly with the term frequency.

Measuring Similarity

Finally, to measure similarity between vectors, we can use any distance metrics between two vectors:

- Cosine similarity
- Euclidean distance
- Dot product similarity



In this repository you can find an exercise for BoVW.

Transformation and Image Stitching

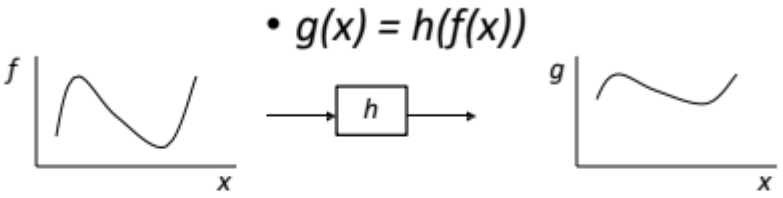
2/04/2024

If we want to create a mosaic we need to know what this transformation is and how to compute it using feature matches.

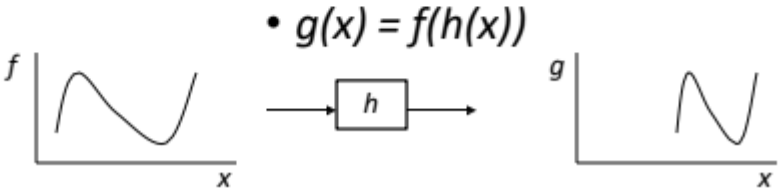
Image Warping

With warping we are changing the domain of the image, so (x,y). This is the opposite of what we have done so far with image filtering, where we were changing the “y axis” so the function f(x,y). In one dimension, this is the comparison between image filtering and warping

- image filtering: change *range* of image

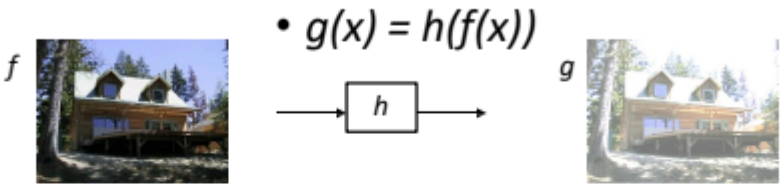


- image warping: change *domain* of image

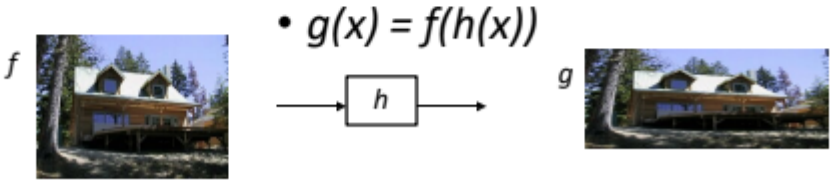


In two dimension:

- image filtering: change *range* of image



- image warping: change *domain* of image



Parametric global warping means to use a transformation T s.t. p' = T(p):

- Parametric: can be described by just a few numbers, the parameters
- Global: is the same for any point p of the image

It's a linear transformation so we have:

$$\mathbf{p}' = \mathbf{T}\mathbf{p} \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{T} \begin{bmatrix} x \\ y \end{bmatrix}$$

Examples of parametric warps:

translation

rotation

aspect

Types of 2D Transformations (Warping)

Linear Transformation

In 2D, so with a 2x2 matrix:

1. Identity

2D Identity?

$$x' = x \qquad y' = y \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2. Rotation

- Rotation by angle θ (about the origin)

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

What is the inverse?

For rotations:

$$\mathbf{R}^{-1} = \mathbf{R}^T$$

3. Scaling

2D Scale around (0,0)?

$$x' = s_x * x \qquad y' = s_y * y \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

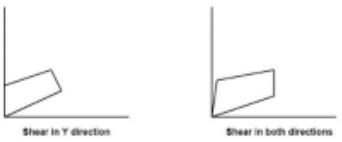
- Uniform scaling by *s*:

$$\mathbf{S} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$$

4. Shear

$$x' = x + sh_x * y$$
$$y' = sh_y * x + y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & sh_x \\ sh_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



5. Mirror

2D Mirror about Y axis?

$$x' = -x$$
$$y' = y$$

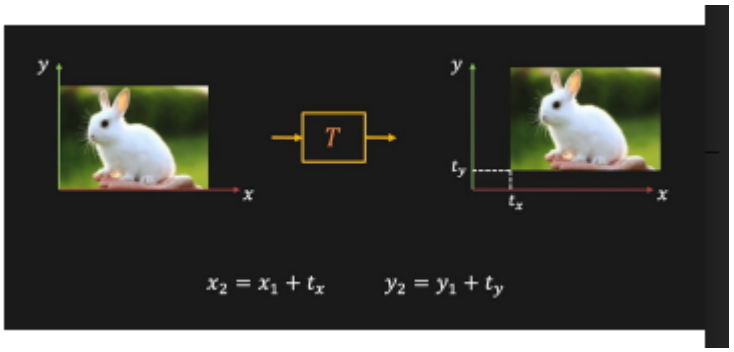
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D Mirror over (0,0)?

$$x' = -x$$
$$y' = -y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Note that translation is not a linear operation on 2D coordinates, so we can't represent it with a 2x2 matrix.



In fact, is impossible to represent this relationship:

2D Translation?

$$x' = x + t_x$$
$$y' = y + t_y$$

in the form x' = A*x

Properties of linear transformations

1. Origin maps to origin
2. Lines map to lines
3. Parallel lines remain parallel
4. Ratios are preserved
5. Closed under composition

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} e & f \\ g & h \end{bmatrix}\begin{bmatrix} i & j \\ k & l \end{bmatrix}}_{T'=T1T2T3}\begin{bmatrix} x \\ y \end{bmatrix}$$

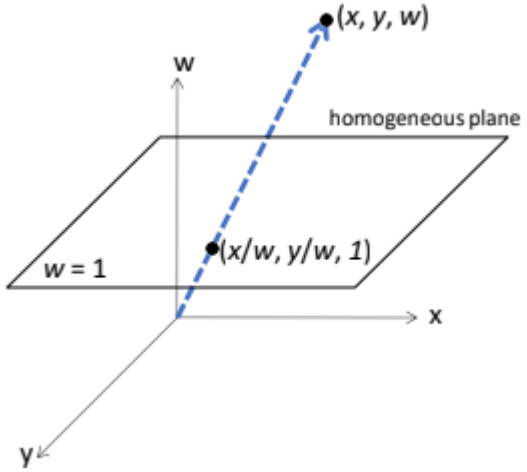
so T' that is a composition of 3 linear transformations is itself a linear transformation

Affine Transformations

We want to represent translation, how can we do it?

$$\boldsymbol{x'} = \boldsymbol{x} + \boldsymbol{t}_x$$
$$\boldsymbol{y'} = \boldsymbol{y} + \boldsymbol{t}_y$$

We can represent it as a 3x3 matrix, so we need to add a 1 to the coordinates [x, y, 1]. These are called Homogeneous Coordinates.



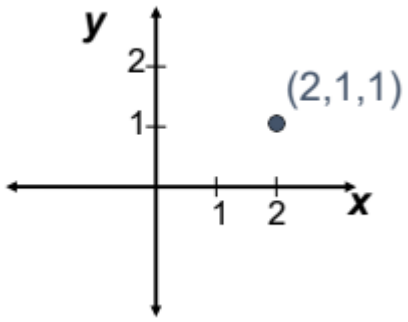
Using Homogeneous coordinates, when we have a [x, y, w] vector, we can re-convert it in 2D in the following way:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} \Rightarrow (x/w, y/w)$$

So that means that a point that in homogeneous coordinates is:

- [x, y, w] is a point at location [x/w, y/w]
- [x, y, 0] represents a point at infinity
- [0, 0, 0] is not allowed

Since we divide by w when going back to 2D, this point here:



is (2,1,1) or (4,2,2) or (6,3,3), (12, 6, 6), …

Now that we have a third dimension we can represent translation:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

Since this transformation has [0, 0, 1] as the last row, it is called affine transformation. These are basic affine transformations:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Translate

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D *in-plane* rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Shear

As with linear transformation, we can combine affine transformation:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$\mathbf{p'} = T(t_x,t_y) \quad R(\Theta) \quad S(s_x,s_y) \quad \mathbf{p}$

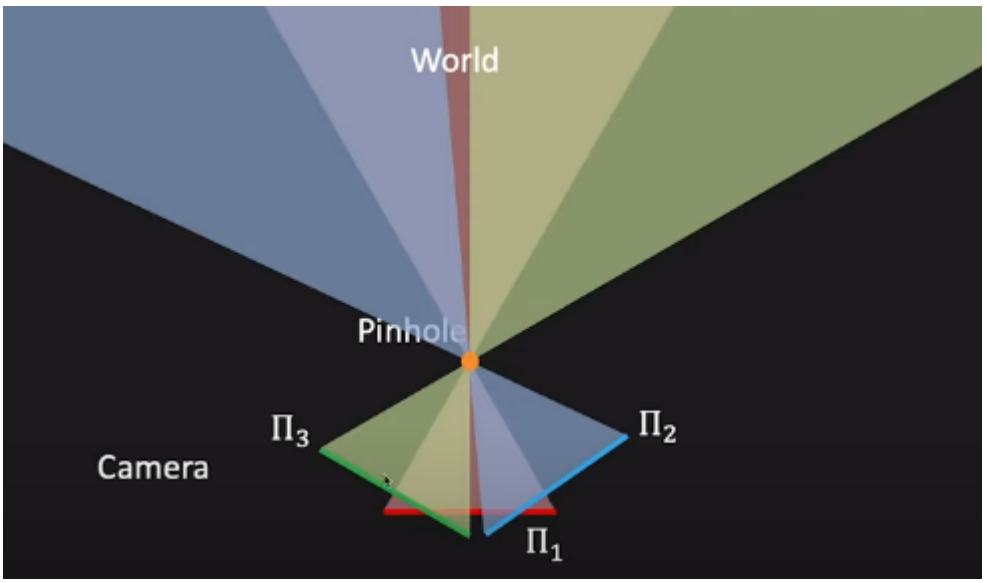
Precisely, we have that affine transformations are combinations of linear transformations and translations.

Properties of affine transformations

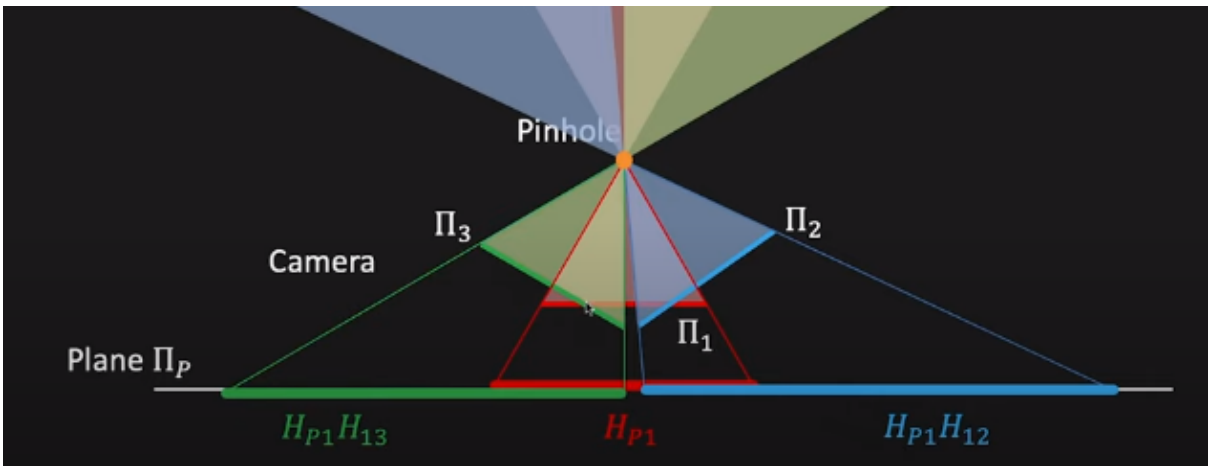
1. Origin DOES NOT NECESSARILY map to origin
2. Lines map to lines, as linear transformations
3. Parallel lines remain parallel, as linear transformations
4. Ratios are preserved, as linear transformations
5. Closes under composition, as linear transformations

Homography

Also called Projective Transformations or Planar Perspective Maps. An homography is a transformation that maps from one plane to another. Let's consider this example:



So we take three images of the world from different planes (perspectives). All these images use the same pinhole. Now, let's consider another plane Π_P. There exists an homography that brings the image from Π_1 to Π_P. And actually there are homographies for all the planes:

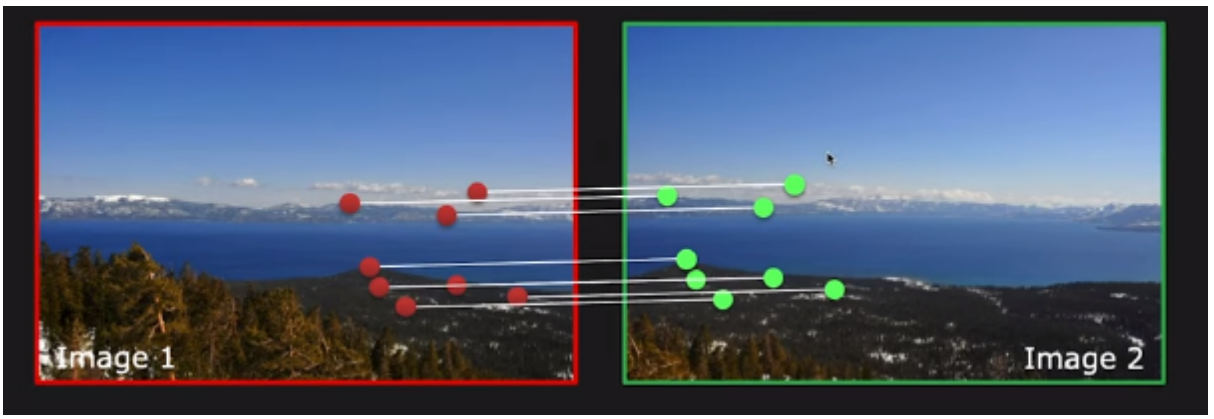


An homography transformation has the following matrix:

$$\mathbf{H} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix}$$

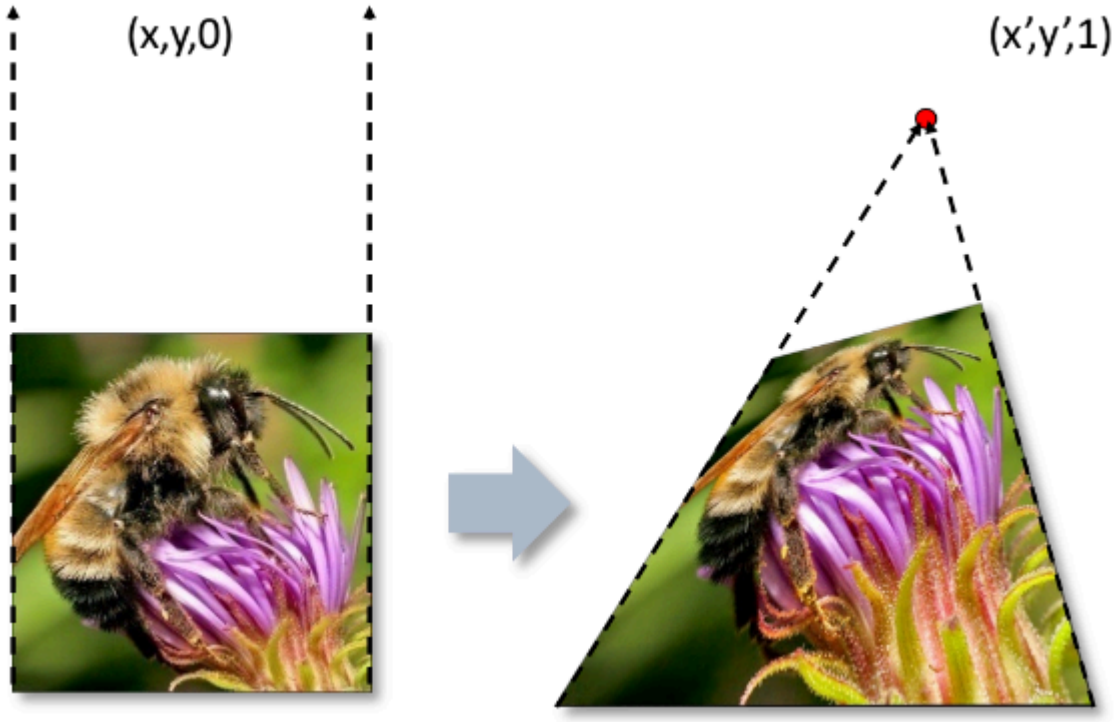
Usage of Homographies

1. Let's consider the following two images:

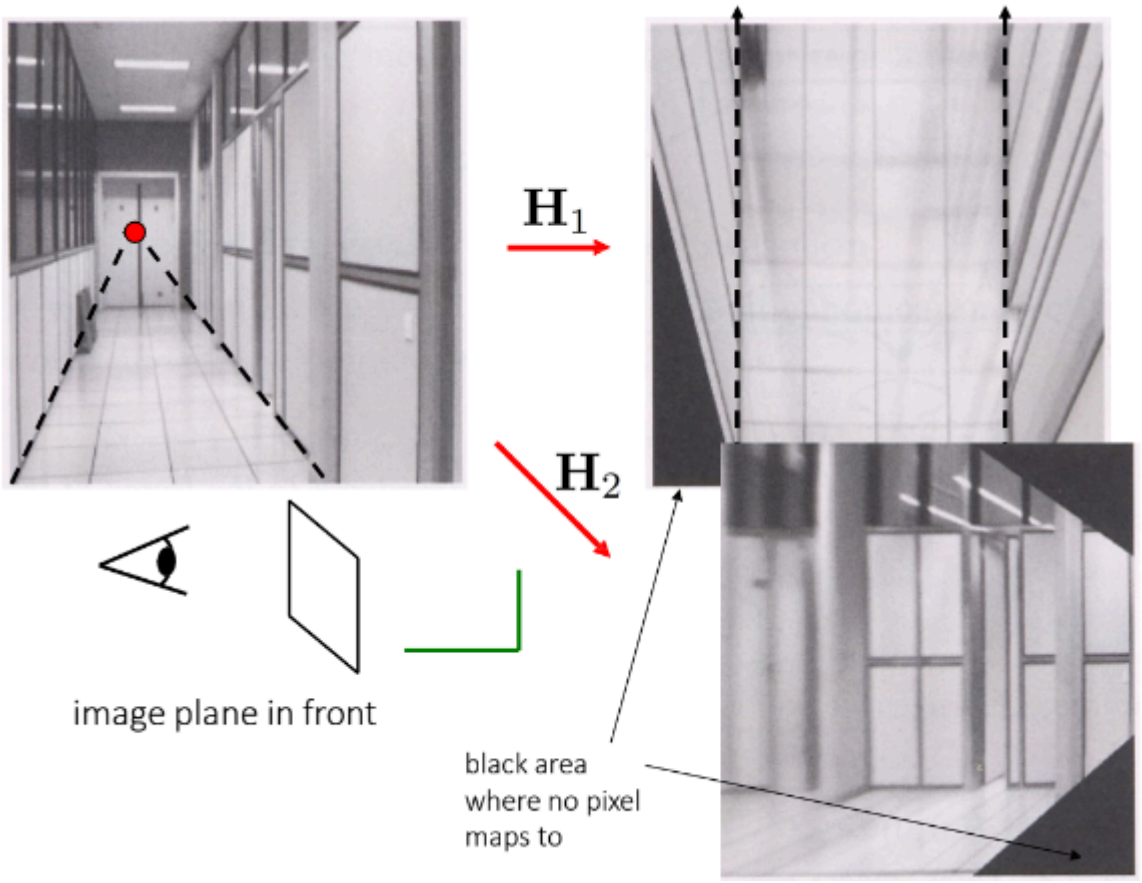


With SIFT we can find a set of matching features between the two images, and then find an homography H that best “agrees” with the matches and allows us to pass from image 1 to image 2 and vice versa.

2. Homographies are used to represent vanishing points or parallel lines that converge at infinity in the original scene in homogeneous coordinates.



This can be used for image warping. Look at the example below:



H1 is a transformation that shows how the two lines become parallel in the new plane. Note that there are black areas where no pixel from the original image maps to, often occurring in the edges. So information that is not present in the original image is of course not present after applying the homography.

Properties of projective transformations (Homographies)

1. Origin DOES NOT NECESSARILY map to origin
2. Lines map to lines, as linear and affine transformations
3. Parallel lines DO NOT NECESSARILY remain parallel, as we have seen above
4. Ratios ARE NOT preserved
5. Closed under composition, as linear and affine transformations

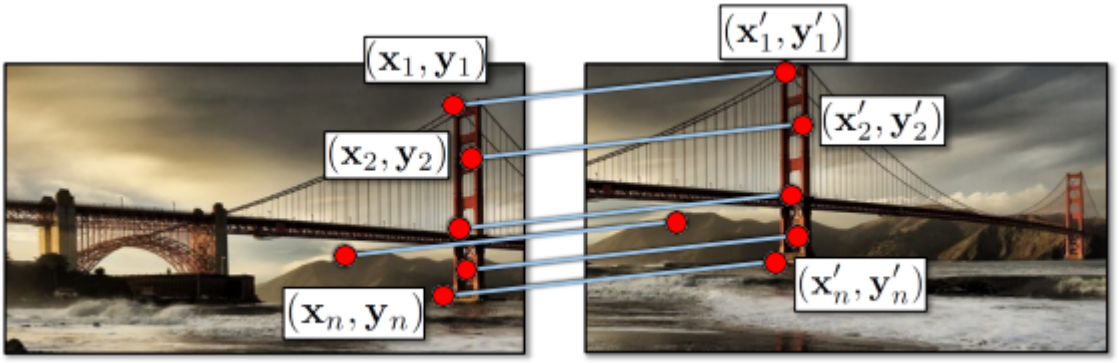
So basically they only guarantee that lines remain lines.

Computing 2D Transformations (Warping)

Now that we have seen several types of transformations, let's see how to actually compute them. Given a set of matches between image A and B, computing the transformation T from A to B means to find the transform T that best "agrees" with the matches.

How to compute Linear Transformation

Let's consider a simple case of linear transformation, a translation:



We can compute the displacement of the i-th match in this way:

$$(\mathbf{x}'_i - \mathbf{x}_i, \mathbf{y}'_i - \mathbf{y}_i)$$

This is basically how much (x_i', y_i') is shifted from its match (x_i, y_i). From this, we can compute:

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

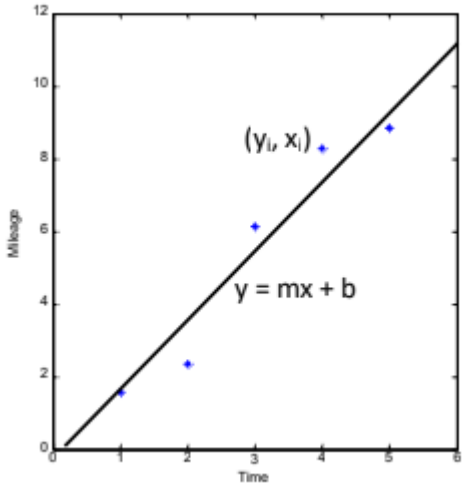
And so now we have find the values for the translation:

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

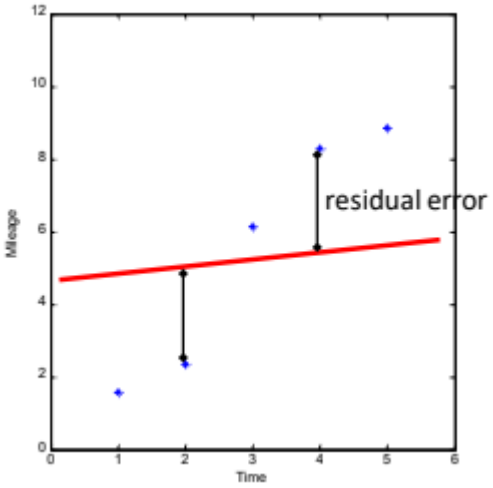
Now, actually, there is a problem: we have 2 equations for each match (x_i, y_i) <----> (x_i', y_i') but just two variables x_t and y_t, since they are the same for each match. Having more equations than unknowns means that we have an "overdetermined" system of equations and we need to use the *least* squares *solution* to solve the system and find x_t and y_t.

Least Squares Solution for Overdetermined Systems

In general, when we talk about least squares we talk about a method to evaluate the error that a model makes trying to predict a certain value that we know exactly. For instance, using a linear regression:



This linear function is trying to approximate the function that represents the distribution of the points.



We can compute the error made by the model with:

$$\text{Cost}(m, b) = \sum_{i=1}^{n} |y_i - (mx_i + b)|^2$$

Writing it with a matrix form:

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \\ x_n & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

If we want to minimize this error, we have to see where the derivative is zero. Since the variables are m and b, we have to calculate the partial derivatives w.r.t to m and b. Now, calling the matrix on the left A, the vector of variables x and the vector of known elements b, we have our system of linear equations A*x = b. As we said, we use least squares solution when the linear system is overdetermined, in order to find a vector x such that:

$\min_x ||A*x - b||^2$

Expanding this expression, we have:

$$\|A\mathbf{x} - \mathbf{b}\|^2 = (A\mathbf{x} - \mathbf{b})^T(A\mathbf{x} - \mathbf{b})$$

$$= (\mathbf{x}^T A^T - \mathbf{b}^T)(A\mathbf{x} - \mathbf{b})$$

$$= \mathbf{x}^T A^T A\mathbf{x} - \mathbf{x}^T A^T \mathbf{b} - \mathbf{b}^T A\mathbf{x} + \mathbf{b}^T \mathbf{b}$$

Now, to find the minimum, we take the derivative with respect to x and set it equal to zero:

$$\frac{d}{d\mathbf{x}}\|A\mathbf{x} - \mathbf{b}\|^2 = 0$$

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^T A^T A\mathbf{x} - \mathbf{x}^T A^T \mathbf{b} - \mathbf{b}^T A\mathbf{x} + \mathbf{b}^T \mathbf{b}) = 0$$

$$2A^T A\mathbf{x} - 2A^T \mathbf{b} = 0$$

$$A^T A\mathbf{x} = A^T \mathbf{b}$$

So, we find:

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}$$

Note that A^TA must be invertible. This **x** = (m, b) is the solution.

Residuals

There is another way of seeing the least squares solutions, and it's with residuals:

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

In this case, the goal becomes of minimizing the sum of squared residuals:

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

We can also write the least squares formulation as a matrix equation

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

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

So this is how we find x_t and y_t that solves the linear systems and compute the translation.

How to compute Affine Transformation

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

Analogously to linear transformation, we compute residuals and minimize the sum of them squared:

- 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:

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

Calculate partial derivatives w.r.t. (a,b,c,d,e,f) and set to 0.

Written in matrix form:

$$\underbrace{\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ & & & \vdots & & \\ x_n & y_n & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_n & y_n & 1 \end{bmatrix}}_{\mathbf{A}_{2n \times 6}} \underbrace{\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}}_{\mathbf{t}_{6 \times 1}} = \underbrace{\begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ x'_n \\ y'_n \end{bmatrix}}_{\mathbf{b}_{2n \times 1}}$$

How to compute Homographies

Our goal is to unwarp an image. Given a pair of points, namely a match:

$$\begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \cong \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Rearranging the terms:

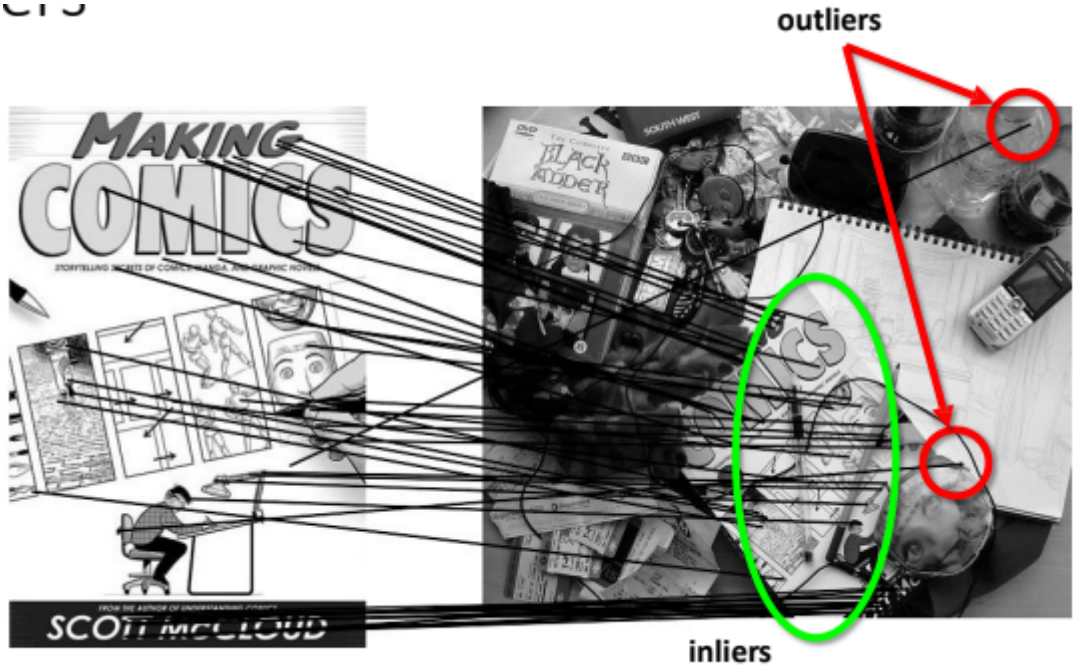
$$
\begin{aligned}
x_i'(h_{20}x_i + h_{21}y_i + h_{22}) &= h_{00}x_i + h_{01}y_i + h_{02} \\
y_i'(h_{20}x_i + h_{21}y_i + h_{22}) &= h_{10}x_i + h_{11}y_i + h_{12}
\end{aligned}
$$

$$
\underset{2n \times 9}{\mathbf{A}} \quad \underset{9}{\mathbf{h}} = \underset{2n}{\mathbf{0}}
$$

$$
\begin{bmatrix}
x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 & -x_1' \\
0 & 0 & 0 & x_1 & y_1 & 1 & -y_1'x_1 & -y_1'y_1 & -y_1' \\
 & & & & & \vdots & & & \\
x_n & y_n & 1 & 0 & 0 & 0 & -x_n'x_n & -x_n'y_n & -x_n' \\
0 & 0 & 0 & x_n & y_n & 1 & -y_n'x_n & -y_n'y_n & -y_n'
\end{bmatrix}
\begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix}
= \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}
$$

And we can solve it with a least squares problem, minimizing $||A*h - 0||^2$. Precisely, the solution, namely the optimal h, is the eigenvector of A^TA that has the smallest eigenvalue.

What could go wrong during the search for a solution? Outliers.



Outliers lead us to the next topic, which is RANSAC.

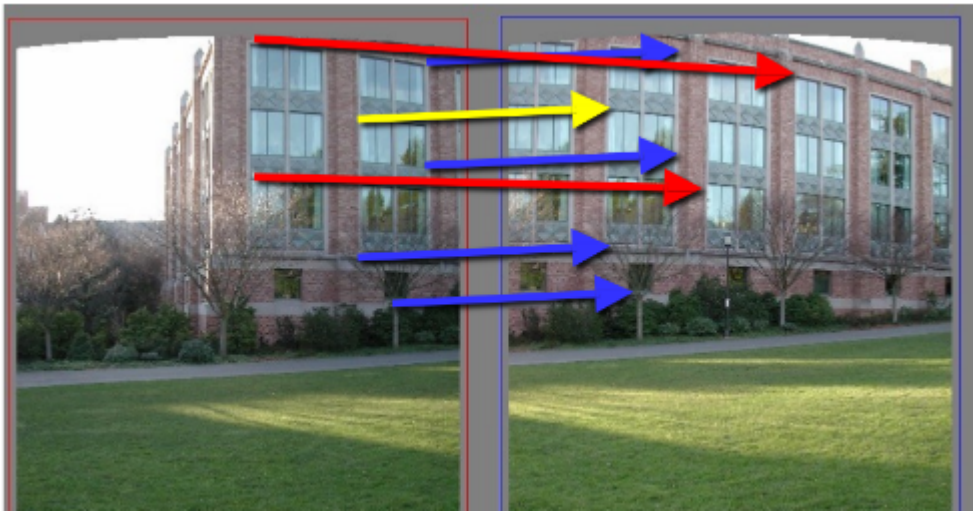
RANSAC (RANdom SAmple Consensus)

“All good matches are alike; every bad match is bad in its own way.”. This is the basic idea of RANSAC to have robustness to outliers. Basically we are assuming that the outliers (hopefully a small number) will disagree with each other, while we know that the inliers will certainly agree with each other. Note that RANSAC only has guarantees if there are < 50% outliers.

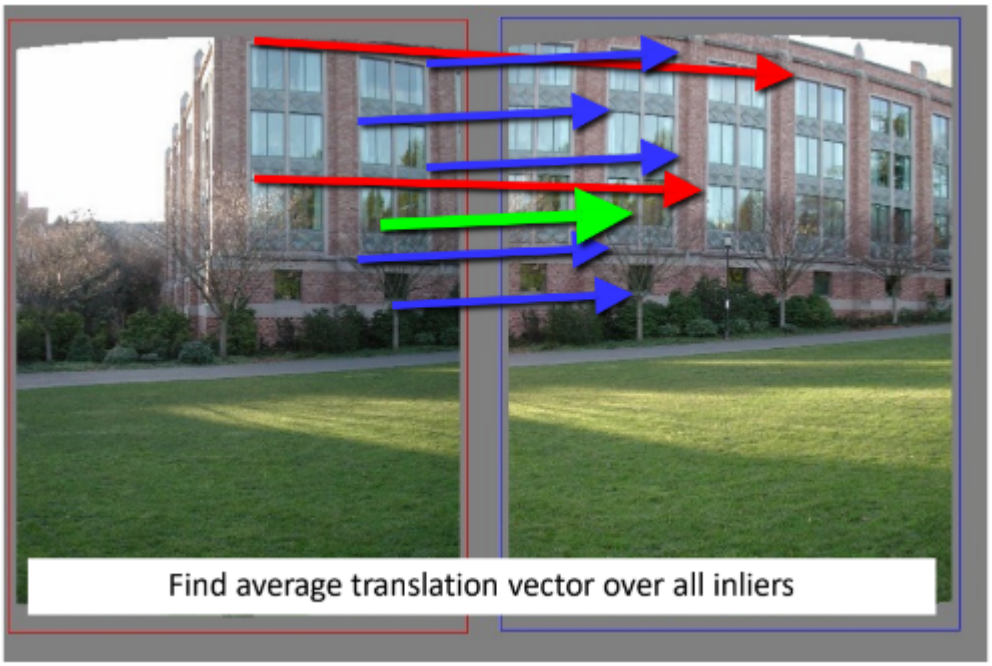
RANSAC for Translation

Given the previous assumption, we can, for all possible lines, count the number of points that agree with each line we choose (i.e. the inliers), meaning that the distance is small, and then select the line with the largest number of inliers. Precisely, the steps are the following:

1. Select one match at random (yellow arrow) and count its inliers. This is equivalent to choose a line and count how many points are within a certain distance



We see that the translation associated with the yellow arrow has 4 inliers (in blue) and 2 outliers, and it’s the one with the number of inliers. After having found the best line, we find the average translation vector over all inliers (in green, below).



This is our translation, robust to outliers.

General Version of RANSAC

1. Randomly choose s samples (matching points) from your data (typically the minimum samples size that lets you fit a model)
2. Fit the model to those samples
3. Count the number M of inliners in all your data (not just s) that fit the model. Basically you take each point in the first image and you plug it into the model. Check whether the output of the model is closer or not to the actual coordinates of the point in the second image. If this distance is below ε then this point is an inlier
4. Repeat steps 1-3 N times
5. Choose the model with the largest number of inliners M

Once you found the best model, you can re-train it with all the inliers found, instead of using only s samples. In this way this model is more refined.

How big should be N? Considering s samples, an outlier ratio “e” and a probability “p” of having the right answer:

$$N \geq \frac{\log(1-p)}{\log(1-(1-e)^s)}$$

RANSAC for Homography

Specifying it for homography, we got:

1. Randomly choose four feature pairs
2. Compute homography H
3. Compute the inliers, considering as them each point s.t. dist(p_i', Hp_i) < epsilon
4. Keep the largest set of inliners
5. Re-compute least-squares H estimate on all of the inliers

RANSAC Pros and Cons

- Pros:
    - Simple and General
    - Applicable to many different problems
    - Often works well in practice
- Cons:
    - Parameters to tune
    - Sometimes too many iterations are required
    - Can fail for low inlier ratios
    - We can often do better than brute-force sampling

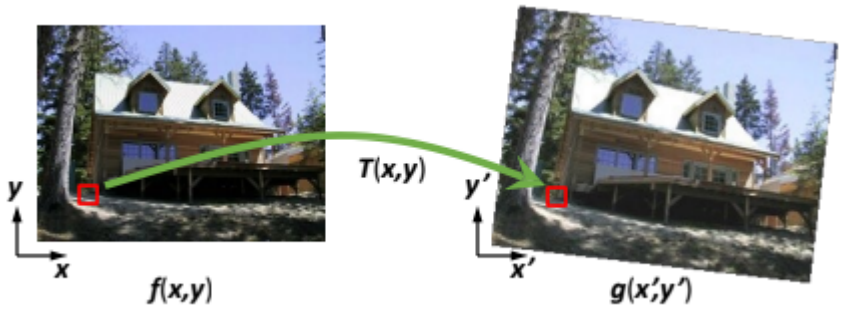
Image Alignment Algorithm

We have a set of images of the same scene ***from roughly the same viewpoint*** but by rotating the camera, with fields of view of the images overlap. We want to stitch these images together to create a panorama. Now that we know how to find the optimal h for homographies, we can do that:

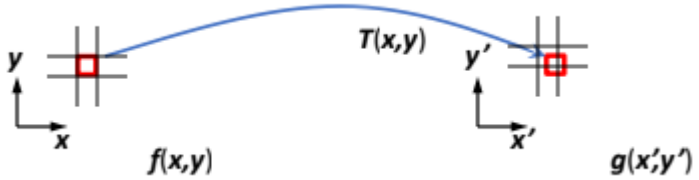
1. Compute the features for image A and B
2. Match features between A and B
3. Compute homography between A and B using least squares on set of matches: this gives you the transformation, with which you warp one of the two image to the reference coordinate frame corresponding to the other image
4. Blend the images together to get a single seamless image

Warping

What is warping? Given a source image f(x,y):



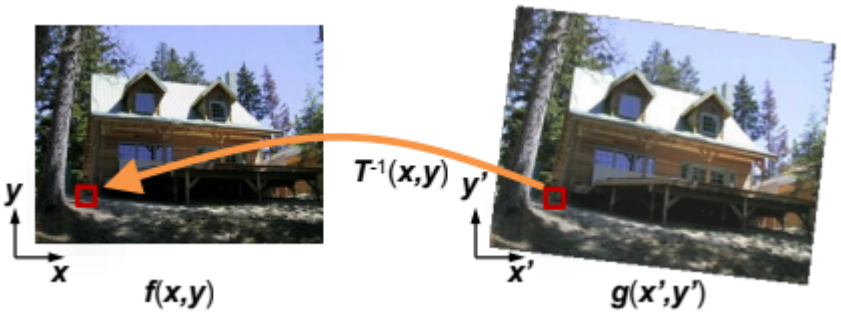
We send each pixel (x,y) to its corresponding location (x', y') = T(x,y) in the destination image g(x',y'). What if a pixel lands between two pixels? We add contributions to several pixels, normalizing later. This is called splatting.



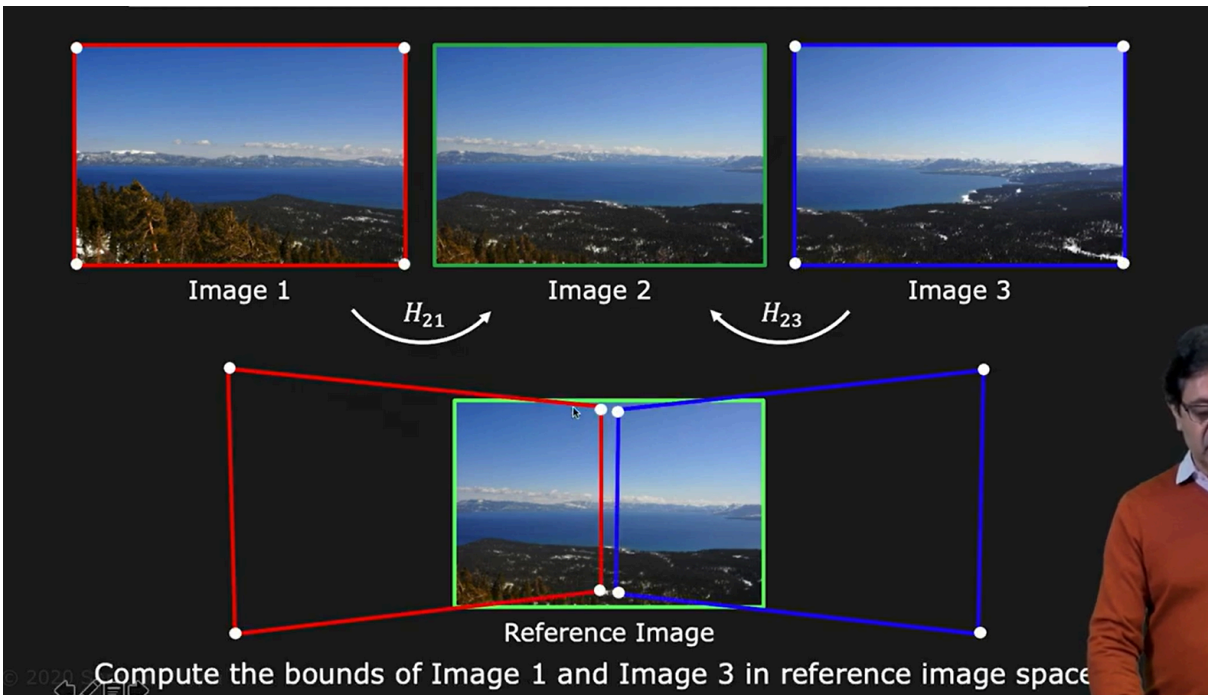
There is another problem: we might get pixels in g that are not filled by pixels in f, so holes in the transformed image. We solve this problem with inverse warping.

With inverse warping, there is no possibility of having holes in g. First we take F and we apply the forward warp to the four corners, getting the four corners of g, its bounding box. Now we want to fill the internal cells. For doing so apply the inverse warping to (x,y), getting the brightness value of the pixel (x,y) in f. We plug this value into the pixel (x,y) in g. If the inverse warping lands in the middle of several pixel, we can obtain this value interpolating with:

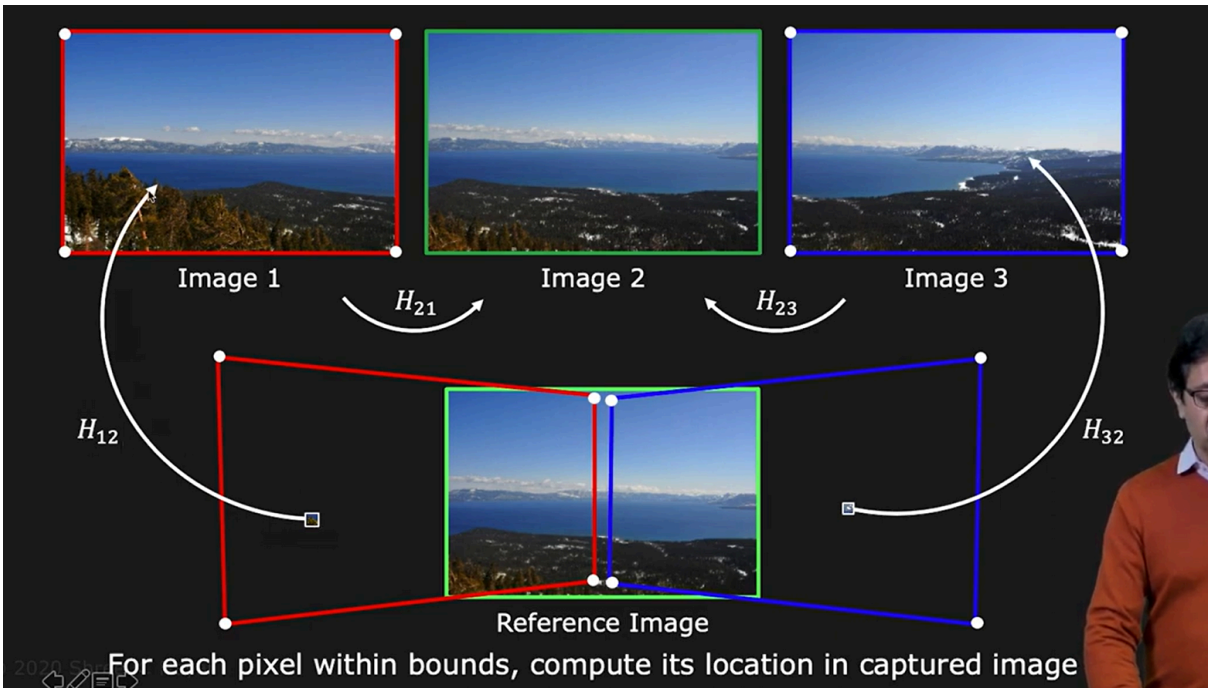
- NN
- Bilinear
- Bicubic
- Sinc



Example: first we calculate the bounding box with the forward warping



Then, for each pixel within the bound, we compute the inverse warping in order to obtain the (eventually interpolated) value of the pixel of the image, that we assign to the warped image



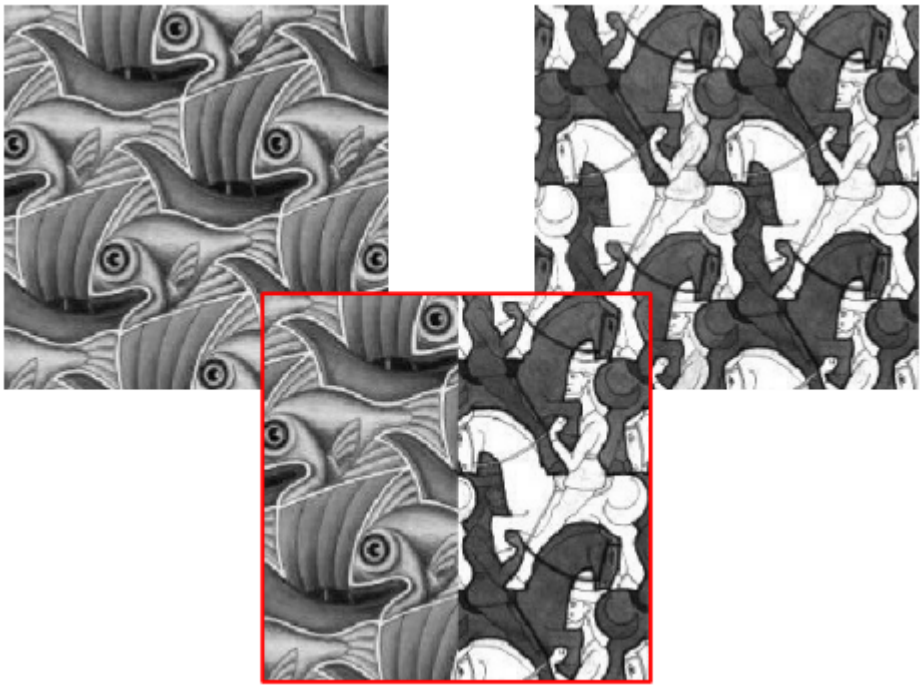
Blending

Now we have an understanding of how to align images, thanks to RANSAC to compute the transformations with robustness w.r.t. to outliers and to Warping. These below are aligned images:

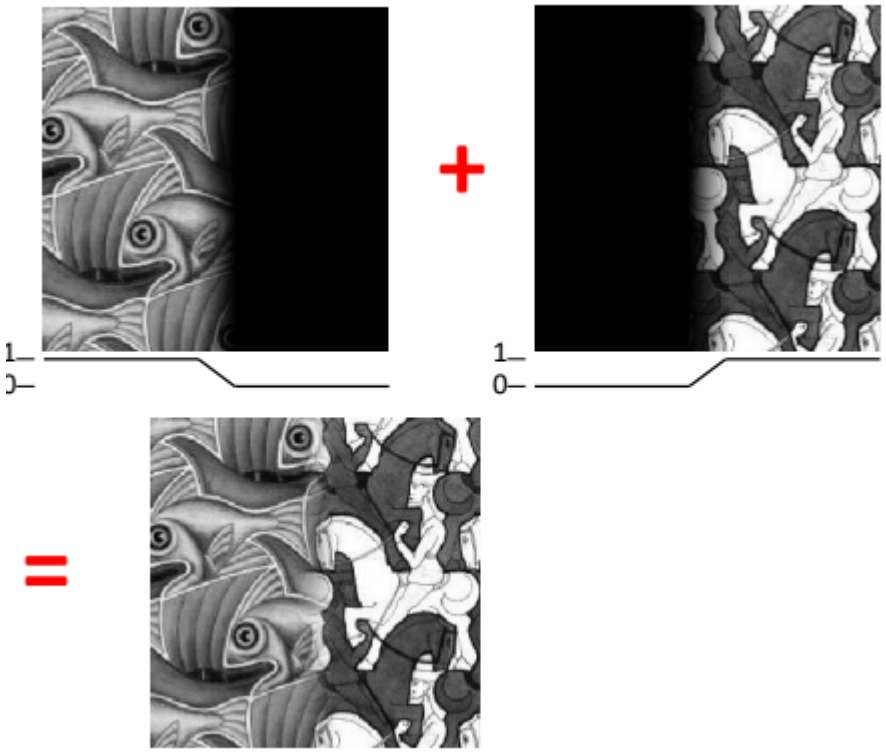
As you can see there are hard seams due to vignetting, exposure differences, etc. How can I now seamlessly blend them together to get a single image?

Feathering

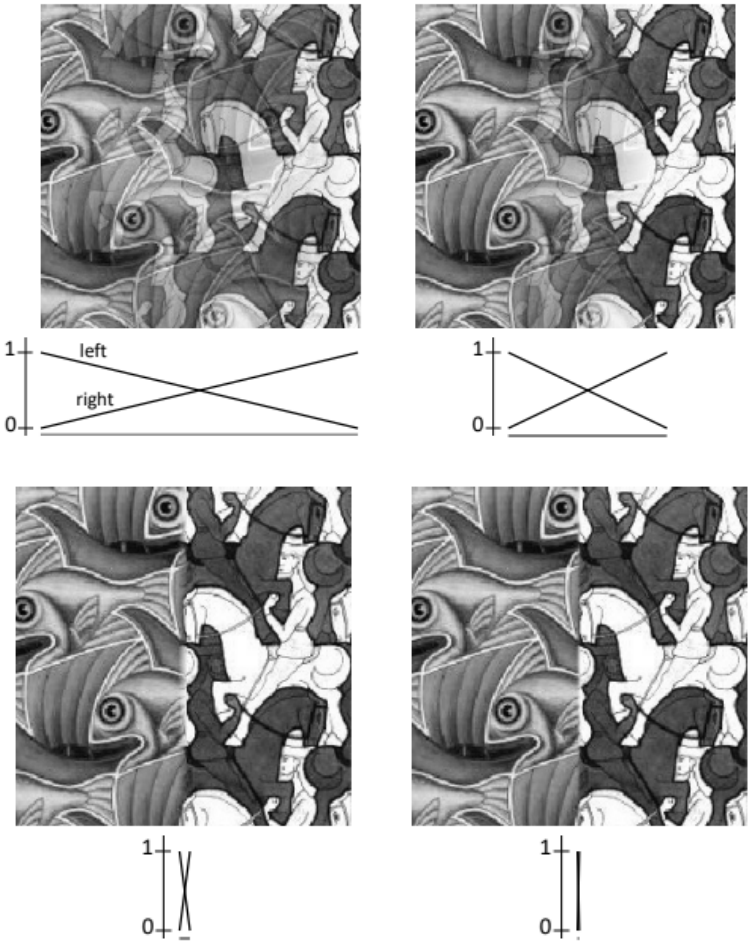
To blend two images together, like these one:



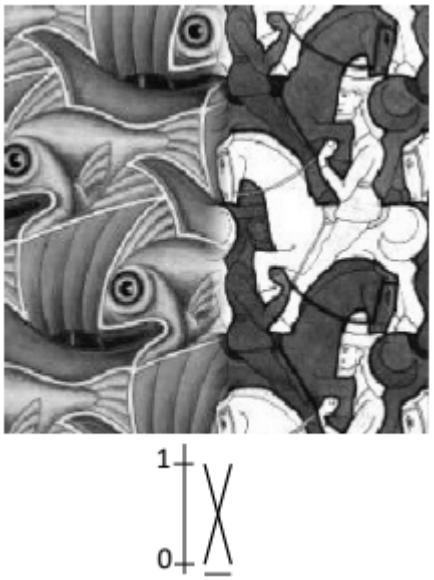
We can do feathering. with weighting functions for images



This technique uses a window size that influences the blending, in fact:



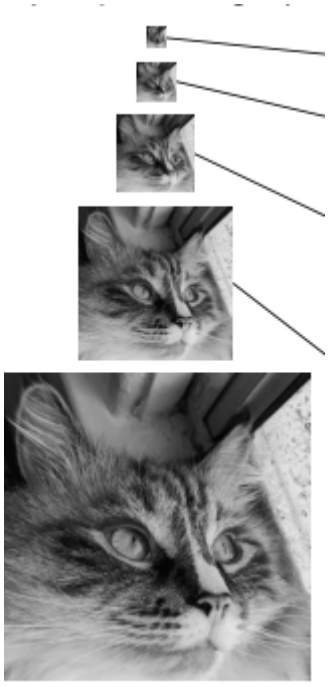
An optimal window size will give a smooth but not ghosted blending:



This does not always work. Another approach to blend is the pyramid blending.

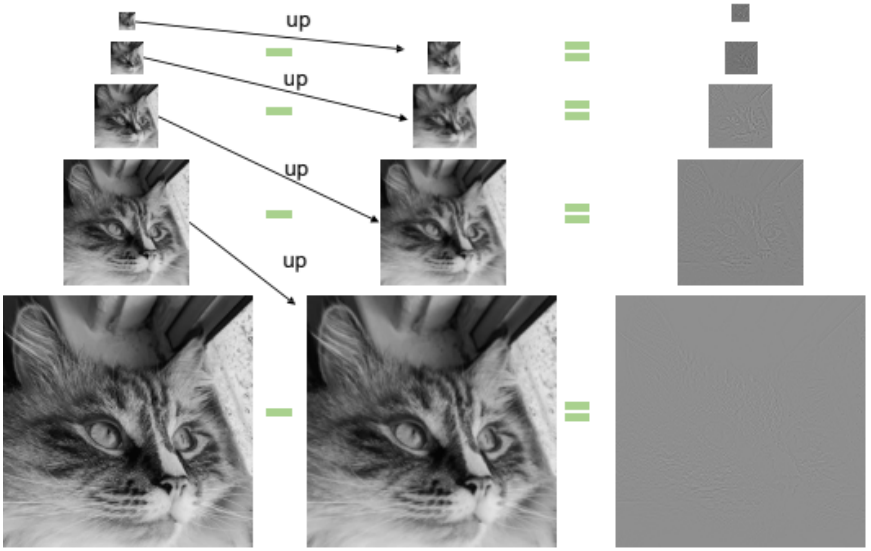
Pyramid Blending

First we create a Gaussian Pyramid

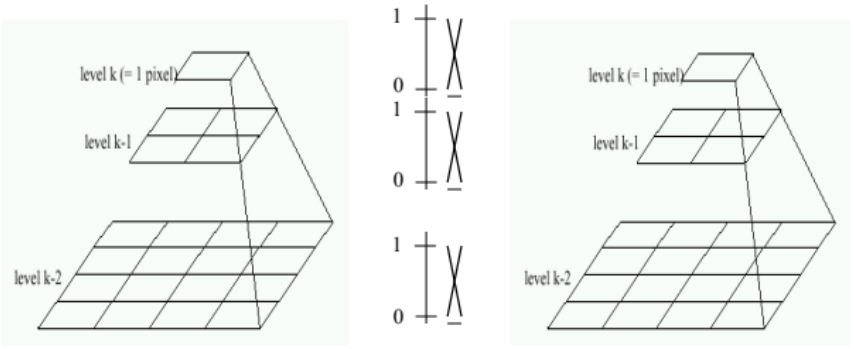


Then we obtain the Laplacian Pyramid[18]:

[18] To obtain the Laplacian Pyramid of an image, we compute the Gaussian Pyramid, then we upsample each level of the pyramid (except the first) and subtract it to the previous level, in order to obtain the information that went lost during the process of blurring and downsampling



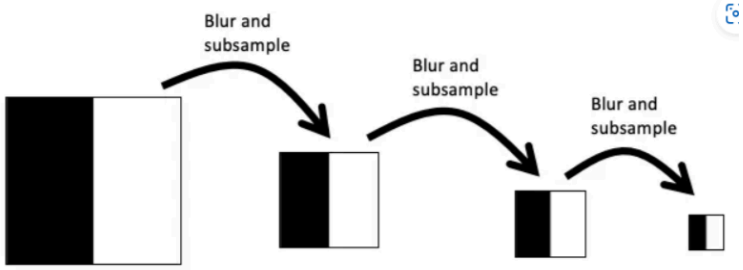
Since we have two images to blend, we will have two laplacian pyramid:



Left pyramid blend Right pyramid

Now we can blend the two pyramids together, getting a blended pyramid. Precisely, the approach is the following:

1. Build Laplacian pyramids LA and LB for images A and B
2. Build a Gaussian pyramid GR from the mask

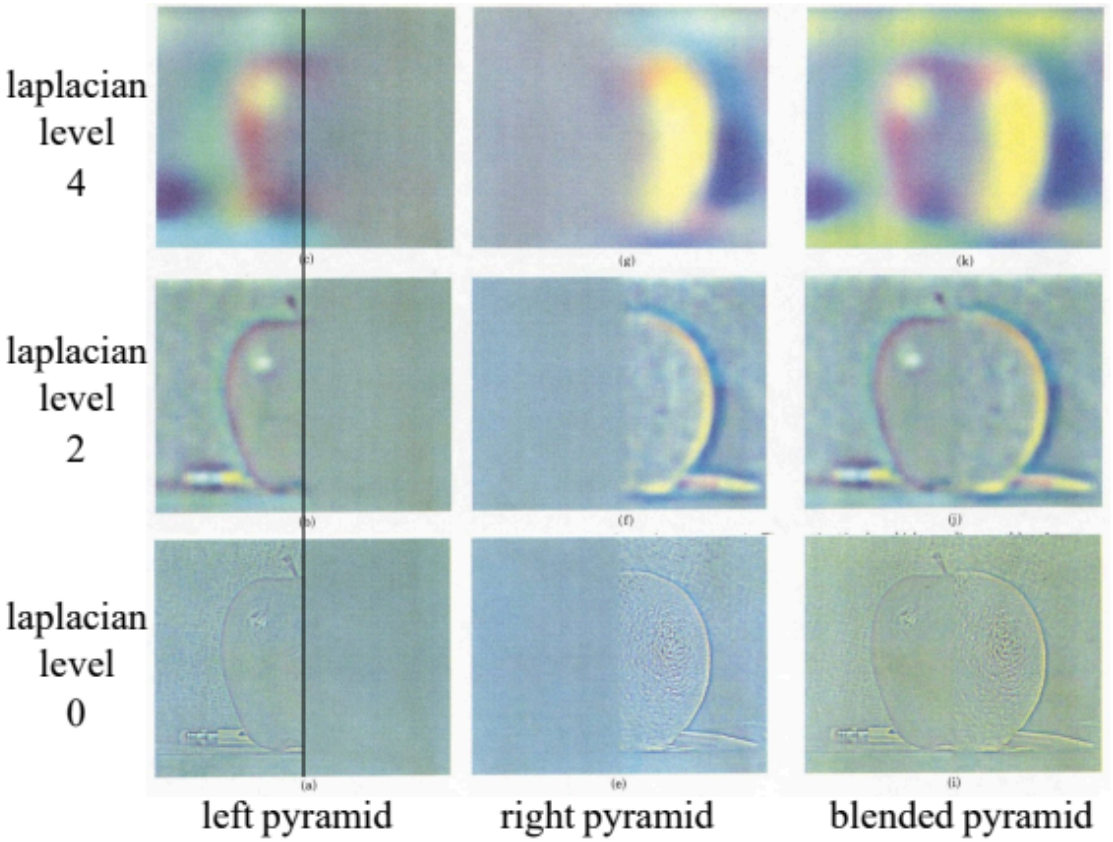


What is this mask? Is a grayscale image indicating where and how the blending should occur.

3. Form a combined pyramid LS from LA and LB using nodes of GR as weights, namely:

$$LS(i,j) = GR(I,j,)*LA(I,j) + (1-GR(I,j))*LB(I,j)$$

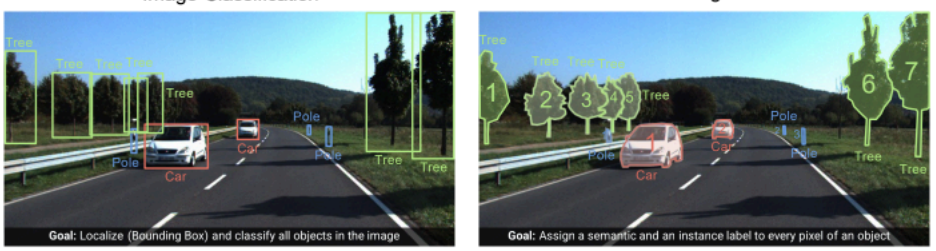
4. Collapse LS to get the final blended image



Recognition

9/04/2024

These are the topic we will deal with today:



1. Image classification: we want to assign a class to an image, in this case it's a "Street Scene". We just want to describe the scene

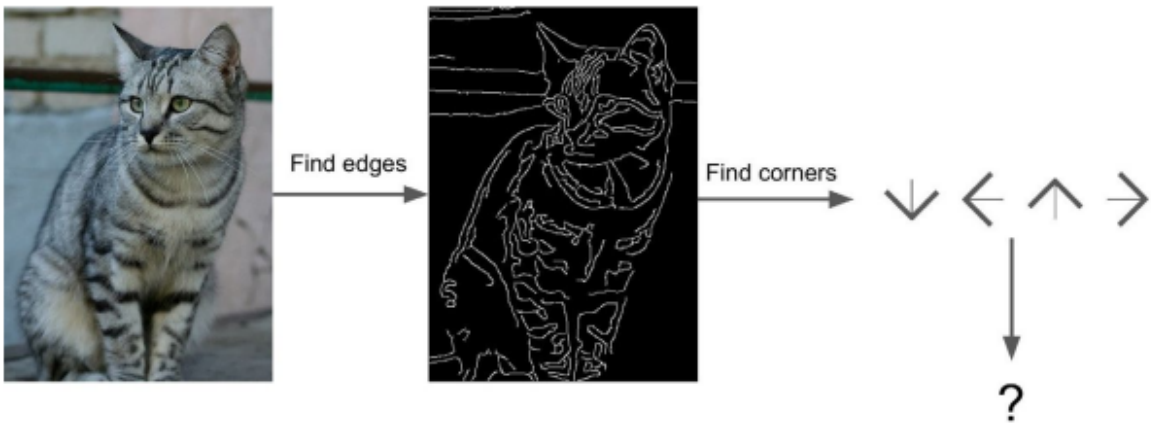
2. Semantic Segmentation: assign a label for each pixel in the image. So pixel x is assigned to class “Sky”, for instance.

3. Object Detection: classify each of the objects and then localize them.

4. Instance Segmentation: it combines aspects of object detection and semantic segmentation. It assigns both a semantic and an instance label to each pixel of the object, so it detects and locates instances of the objects but also provides a precise segmentation mask for each instance.

Image Classification

Useful for image retrieval, for instance. Previously of Deep Learning, it was done like this:



In the beginning one of the attempts was to identify the object through the use of corners, but it’s hard to describe an object with corners. With Machine Learning, we solved this problem easily.

Simple Model for Image Classification

Nearest Neighbor

- Choose image distance:

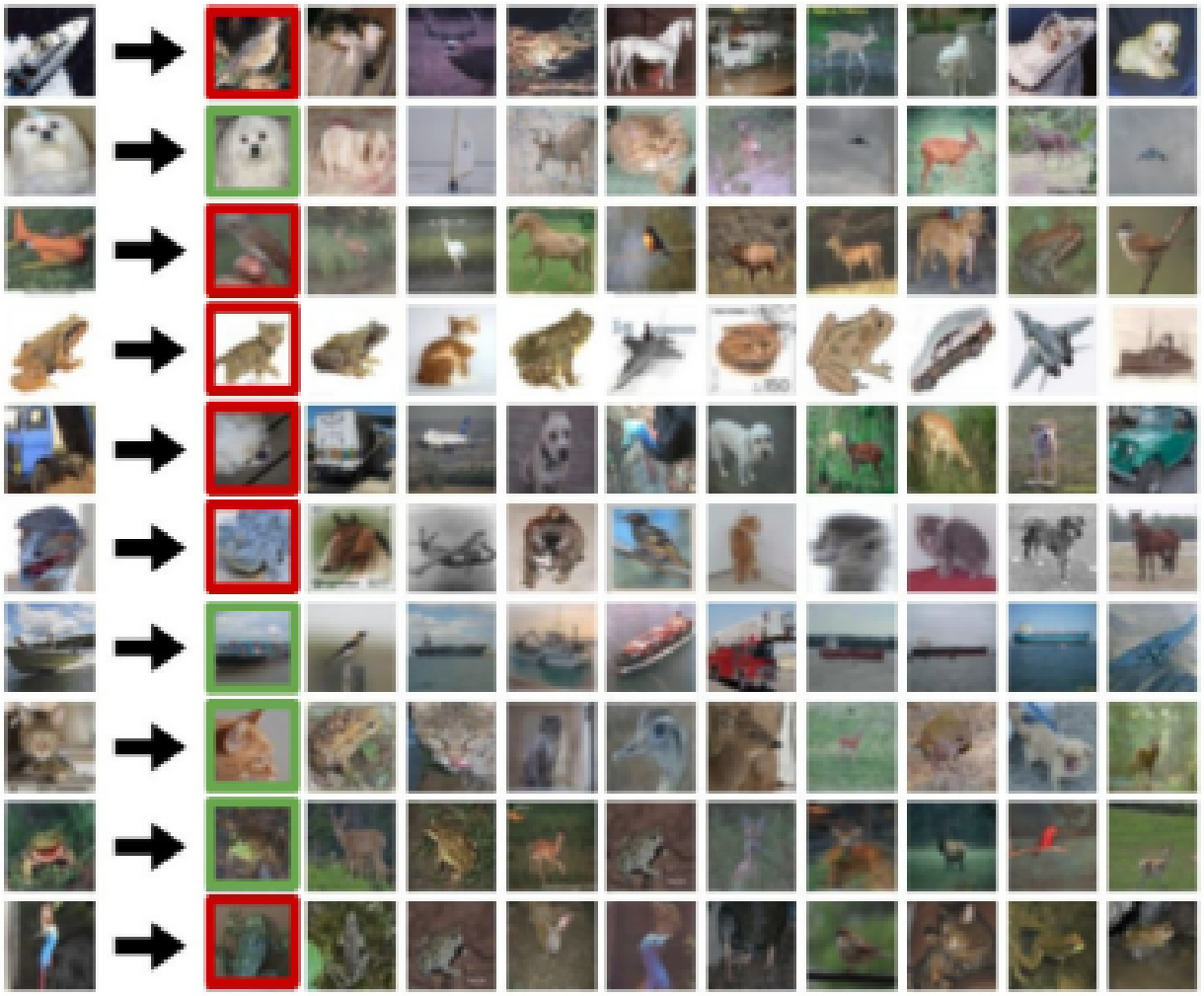
$$d(\mathbf{I}_1, \mathbf{I}_2) = \sum_{\mathbf{p}} |I_1(\mathbf{p}) - I_2(\mathbf{p})|_1$$

- Given $\mathbf{I}$, find nearest neighbor:

$$\mathbf{I}^* = \underset{\mathbf{I}' \in \mathcal{D}}{\operatorname{argmin}}\ d(\mathbf{I}, \mathbf{I}')$$

- Return class of $\mathbf{I}^*$

The problem with this is that it's slow and basically is bad since the pixel distance is not informative on the content of the image. In fact, see the following example using NN on the Caltech 101 dataset:



As you can see basically the model is choosing an image that is similar in shape and color to the one we want to find. This is not good, we want to find something that is invariant to this, especially when it comes to color.

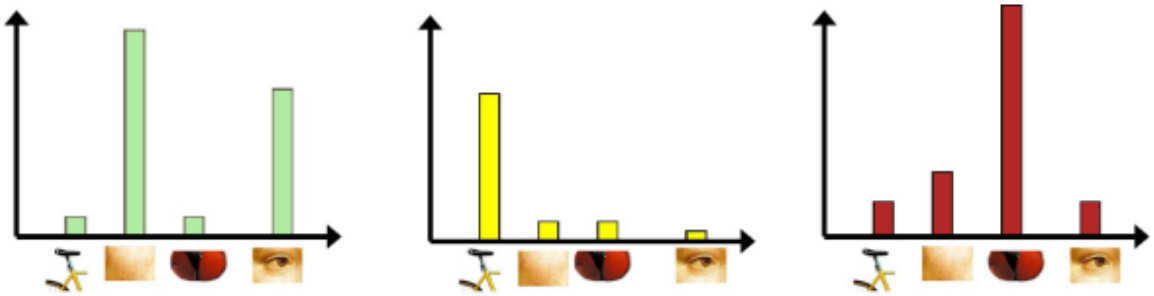
Bag-of-Words (BoW)

It uses histograms of local features. BoW has an orderless document representation, and this property is useful for images.

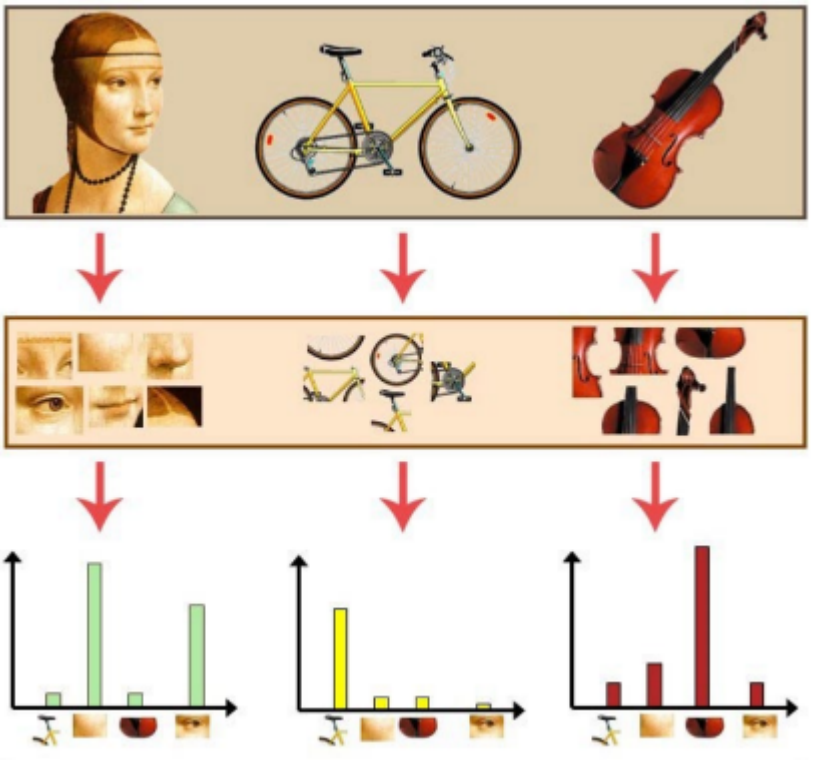
The steps are the following:

1. Extract Features, e.g. with SIFT
2. Learn a visual vocabulary, using an unsupervised algorithm like k-means, that will group similar features together in a cluster. Each cluster represents a visual word.
3. Quantize Features into Visual Words using the vocabulary. Basically we associate a patch to the nearest neighbor visual word

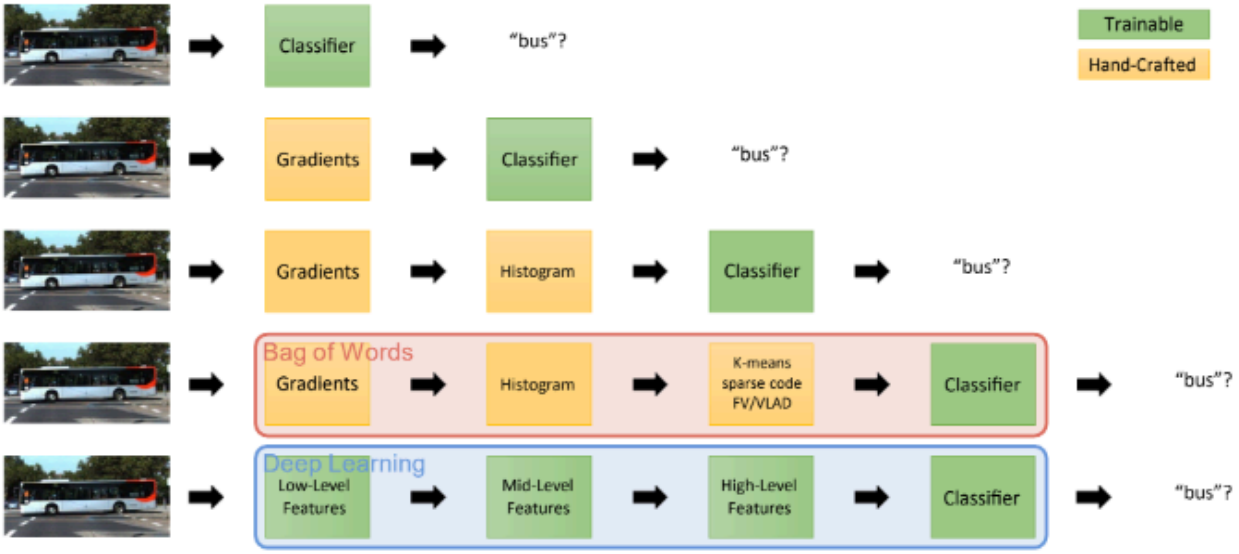
4. Now we can represent an image as an histogram of visual word frequency, as if it was a document. These above are three different images.



5. Train a Classifier, e.g. SVM, Random Forest, Neural Network. In this way we can predict the class of the image having the color histograms as input.

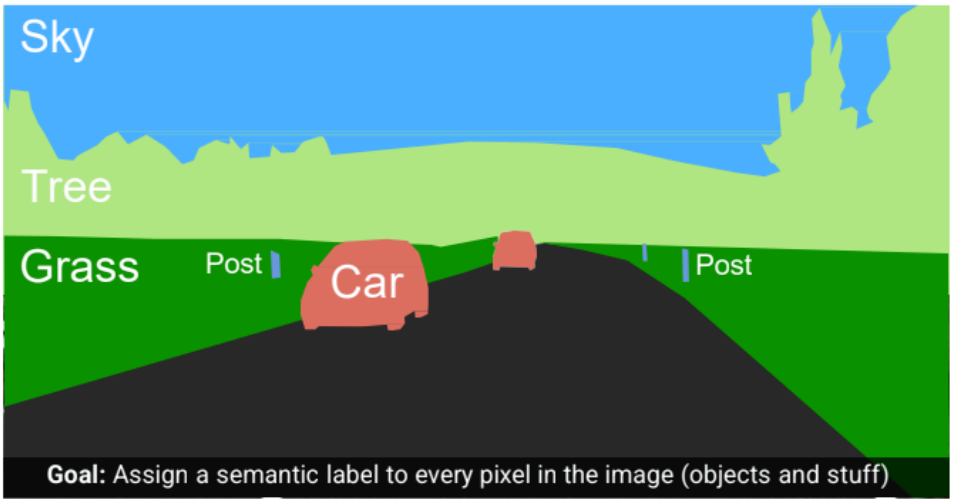


The problem with BoW is that it has many hand-designed components and little learning. The final solution is to use DL and learn the model from data.



With DL we don't have human intervention, but everything is trainable and there are no human hand-crafted components.

Semantic Segmentation



One of the first methods was based on fully convolutional networks, basically we are extending the methods for image classification doing classification per pixel for each class that we want to segment.

Object Detection

Localize means we want to extract the bounding box. Understanding where an object is placed is important, for instance, to avoid incidents in auto-driving cars.

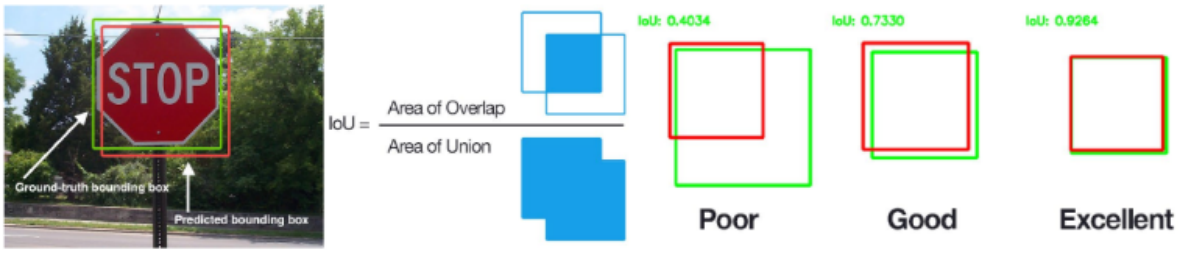
Problem setting: we have an image (we can also have a video, frame by frame) in input, while the output can be of 2 different types:

1. 2D bounding boxes
2. 3D bounding boxes

These bounding boxes will have a category label and a confidence. If we move to 3D locations object detection is not really trivial.

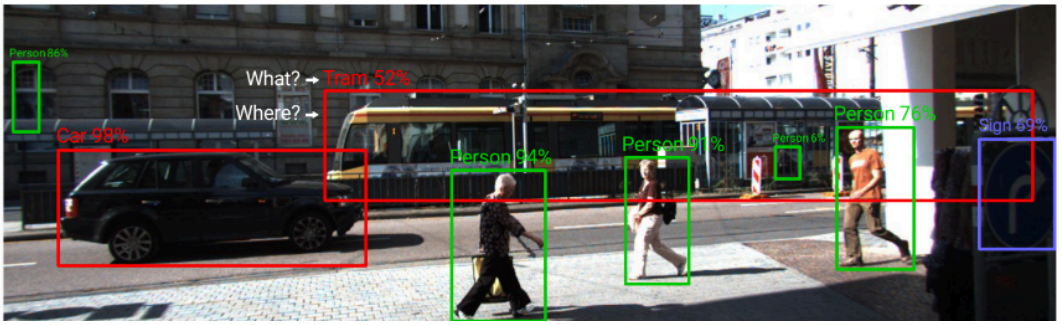
How can we evaluate the performance of an object detector? I have an area that I recover, the red is the bounding box that I recover as output, while the green is the ground truth. I can evaluate the red box w.r.t. to the green, using IoU[19]

[19] Intersection-over-Union



The detection is considered successful if IoU is >= 0.5. This is done for one object. How can we evaluate more than one object? With the Average Precision Metric:

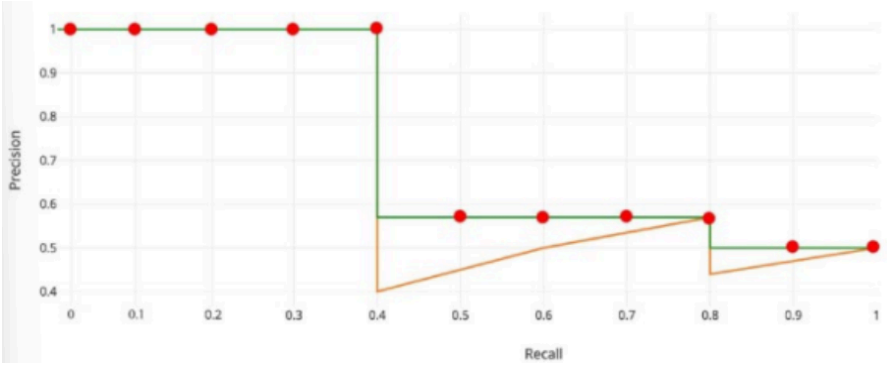
1. Run the detector with varying thresholds. Different thresholds will generate different sets of detected objects



2. For each detected object by the detector, match the bounding box found with the ground truth, calculating the IoU. If it's >= 0.5, it is considered a match. We have at most 1 match per object.
3. Count:
    - TP: object correctly detected, IoU >= 0.5
    - FP: wrong detections
    - FN: object not detected (IoU < 0.5
4. Compute Average Precision (AP)

$$\text{Avg. Prec. } AP = \frac{1}{11} \sum_{R \in \{0,\ldots,1\}} \max_{R' \geq R} P(R')$$

Average Precision (AP) is the area under the precision-recall curve. It is calculated by averaging precision values over a set (in this case 11) of recall thresholds. For instance, keeping constant TP and varying FP and FN, for Recall = 0 (so FN -> infinite), we have that Precision is 1, since FP is neglectable.

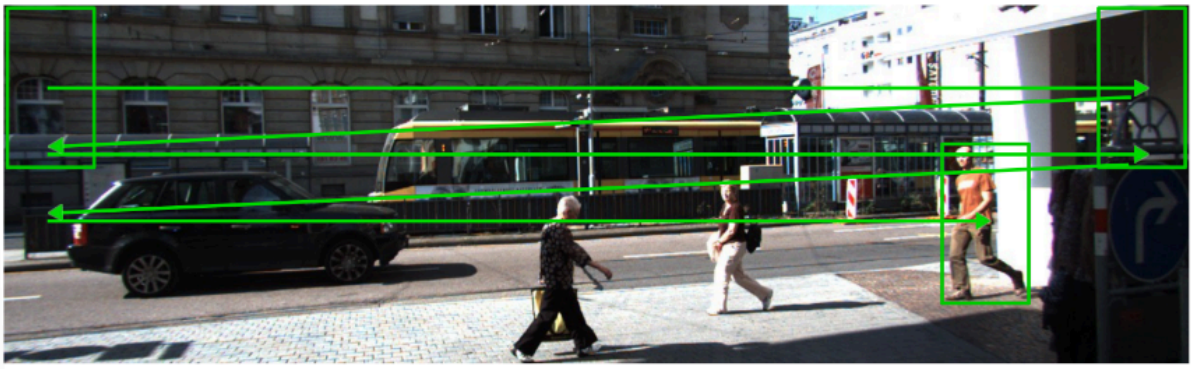


$$\text{Precision } P = \frac{TP}{TP + FP}$$

$$\text{Recall } R = \frac{TP}{TP + FN}$$

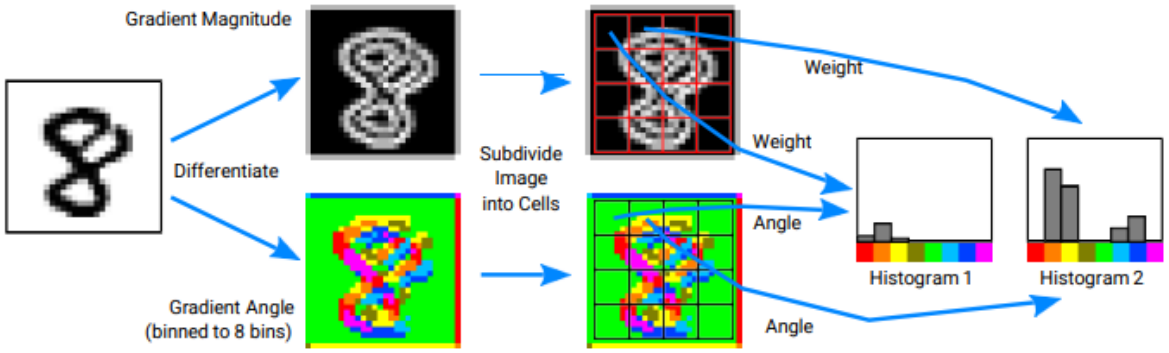
Sliding Window Object Detection:

1. We run sliding windows of fixed size over the image, extracting features for each window.
2. We classify each window as an object or background, using SVM, random forest, …
3. Iterate over multiple aspect ratios/scales and perform non-maxima suppression.



Histograms of Oriented Gradients:

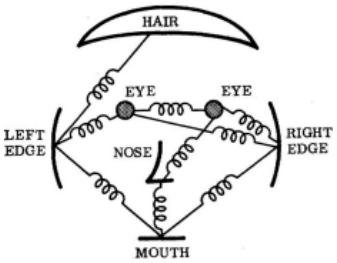
Another technique to detect objects in an image is to use HoG. We represent patches with histograms for magnitude and angle.

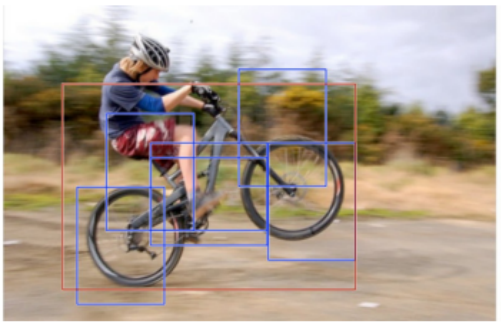


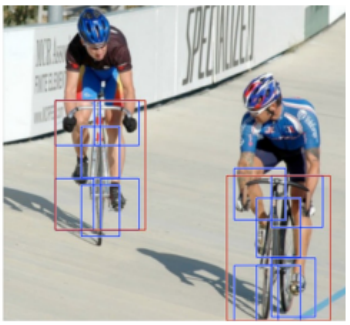
This is invariant to small deformation: translation, scale, rotation, perspective

Part Based Models:

The idea here is to model an object based on its part (e.g., for a human, parts might include the head, arms, legs, etc.). The typical configuration and spatial relationships between these parts are also considered. In this way we allow invariance. However, inference becomes slower than HoG.



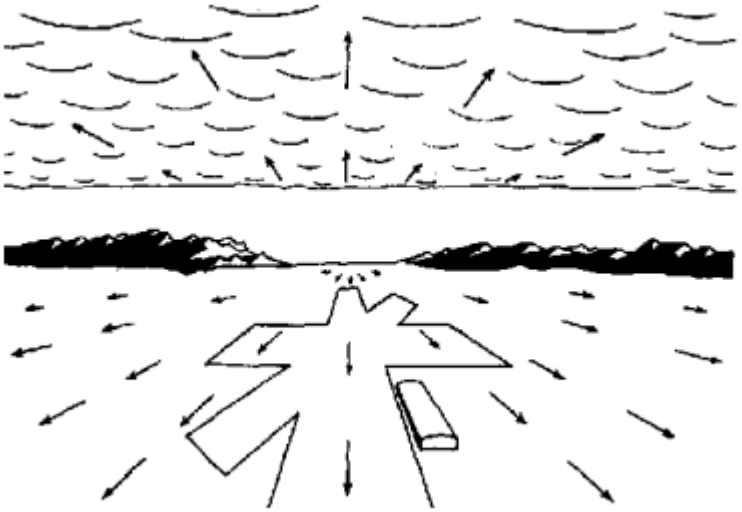




Optical Flow

10/04/2024

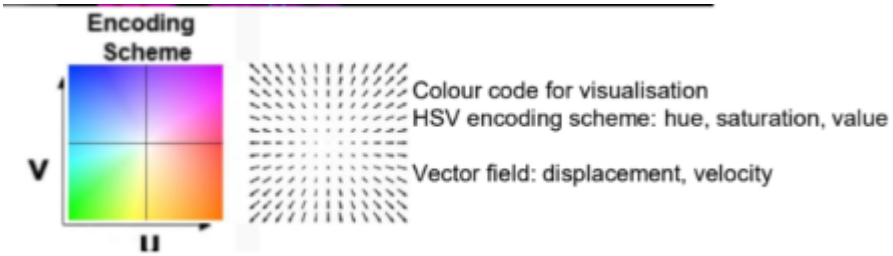
Optical flow is the apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer and a scene.



Optical flow is used to understand the motion of an object frame by frame.

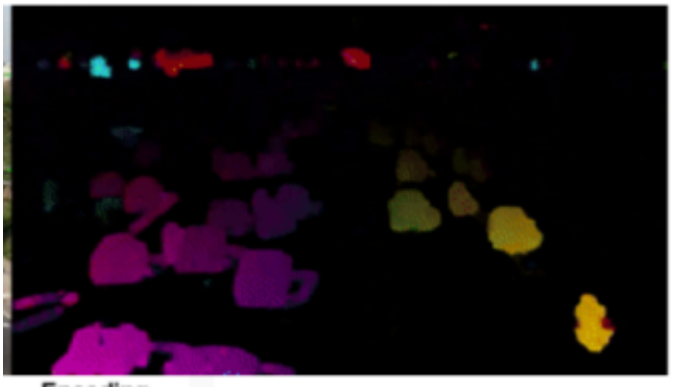
HSV encoding

We can this color encoding to visualize the optical flow:



In this way the optical flow of this image:

Can be represented in this way:



Stereo vs Optical Flow

These two approaches for visual motion analysis have the following differences:

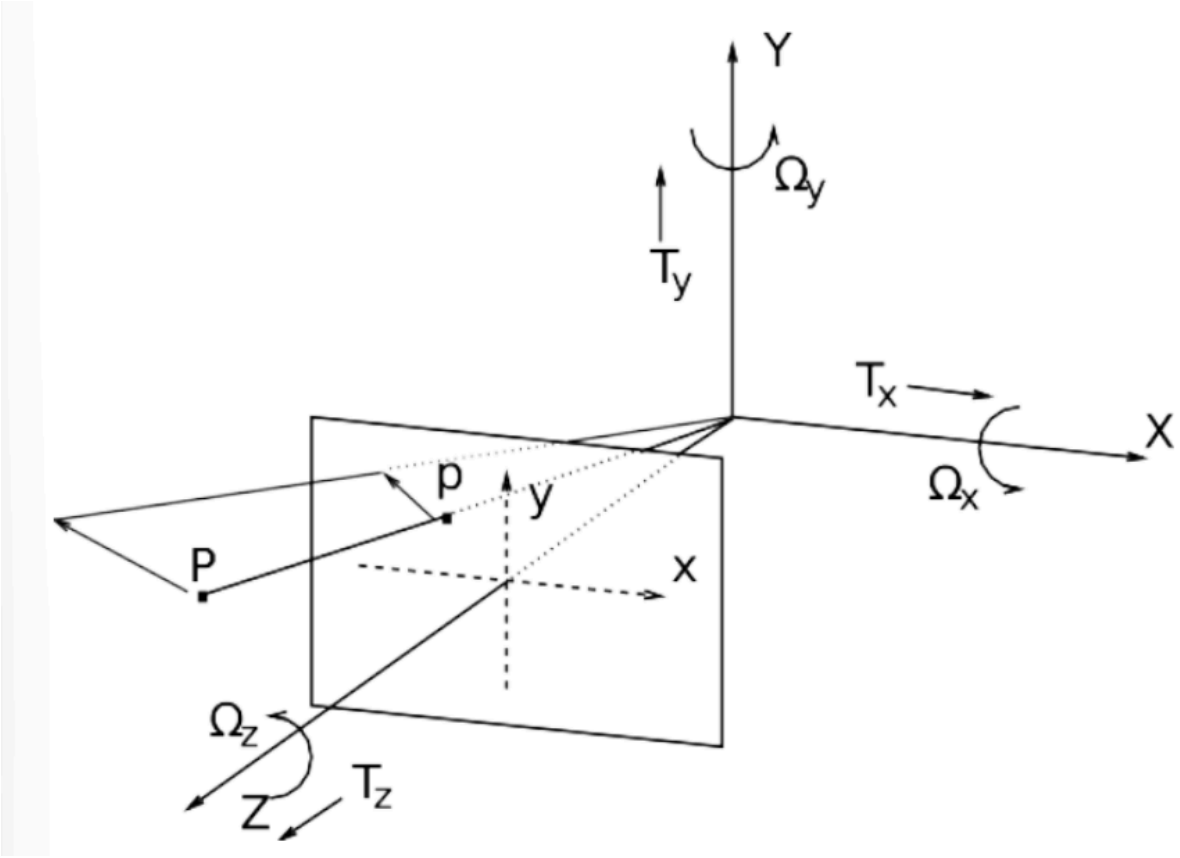
1. Stereo Vision uses 2 images at the same time, while Optical Flow compares two images but in two different timesteps
2. Stereo Vision consider only camera motion, while Optical Flow consider both camera and object motion
3. Stereo Vision is a 1D estimation problem, focusing on disparity along the horizontal direction to determine depth, while Optical Flow is a 2D estimation problem.

Motion Field vs Optical Flow Field

motion: a point moving in 3D, its projection onto the image via perspective projection is the motion field. We can't directly measure it since in the image we have only brightness patterns, and we can measure only brightness patterns. The motion of brightness patterns in the image is the optical flow. Sometimes motion = optical flow, but generally no.

We can't directly measure optical flow uniquely at each pixel by simply looking at the brightness variation of that pixel, but we can come up with an optical flow constraint equation that constrains the optical flow at a pixel.

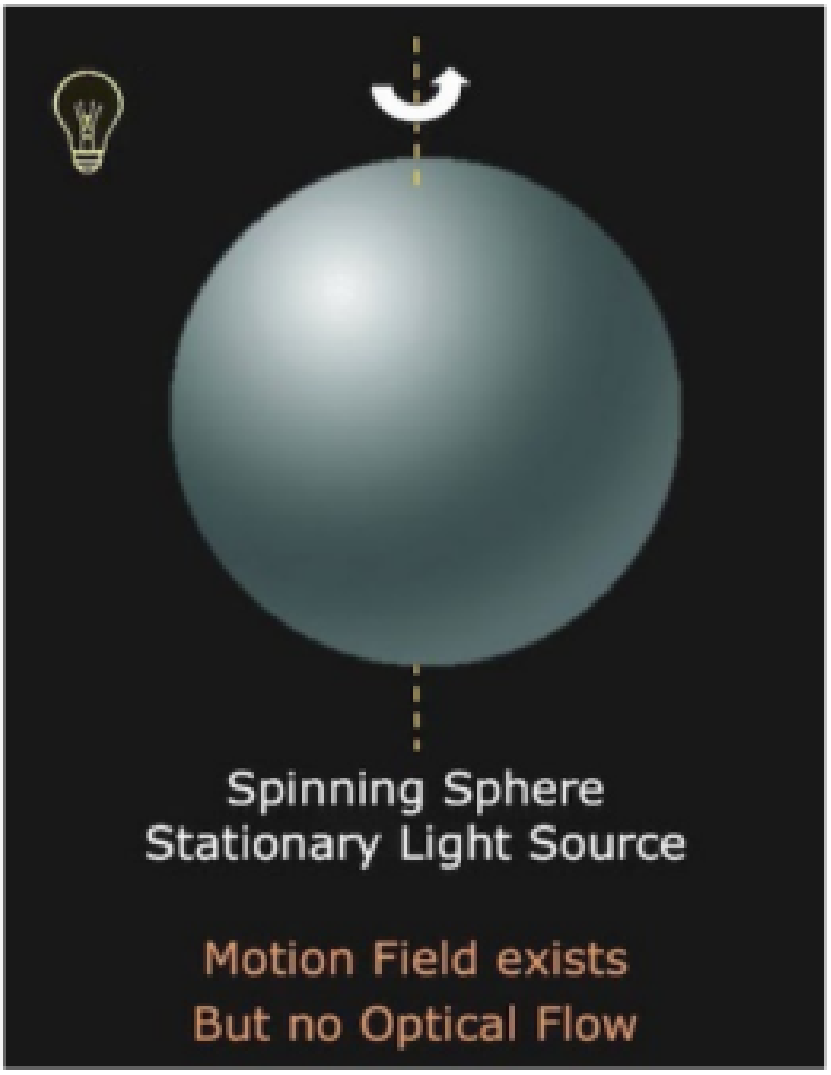
Motion Field: when a 3D point or the camera (or both) are moving, projecting this 3D motion in the image plane results in a 2D motion



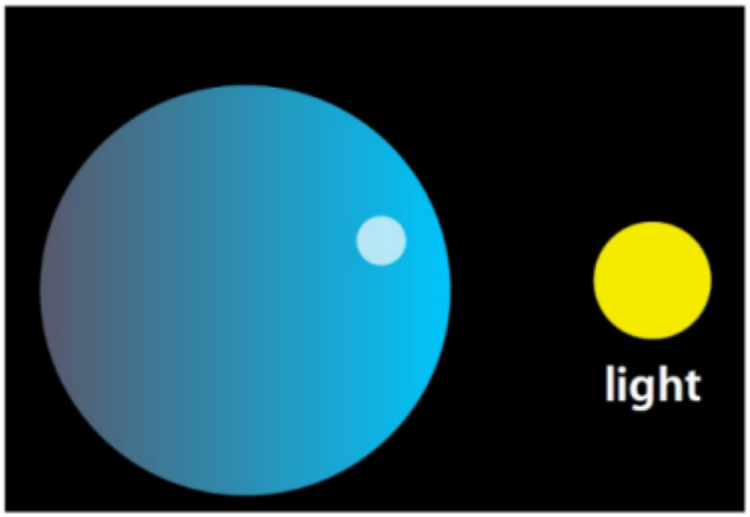
Optical Flow, on the other hand, is a 2D velocity describing the apparent motion in the image, namely the displacement of pixels looking similar. Sometimes we can estimate the Motion Field with Optical Flow Field, sometimes they are completely different. The optical flow tells us something, **maybe ambiguous**, about the 3D structure of the world and the motion of objects and the observer.

Thought Experiment

Given two images, the only thing that we have to understand the motion of an object in the image is the values of the pixels. Let's do this thought experiment to understand the difference between motion field and optical flow field.



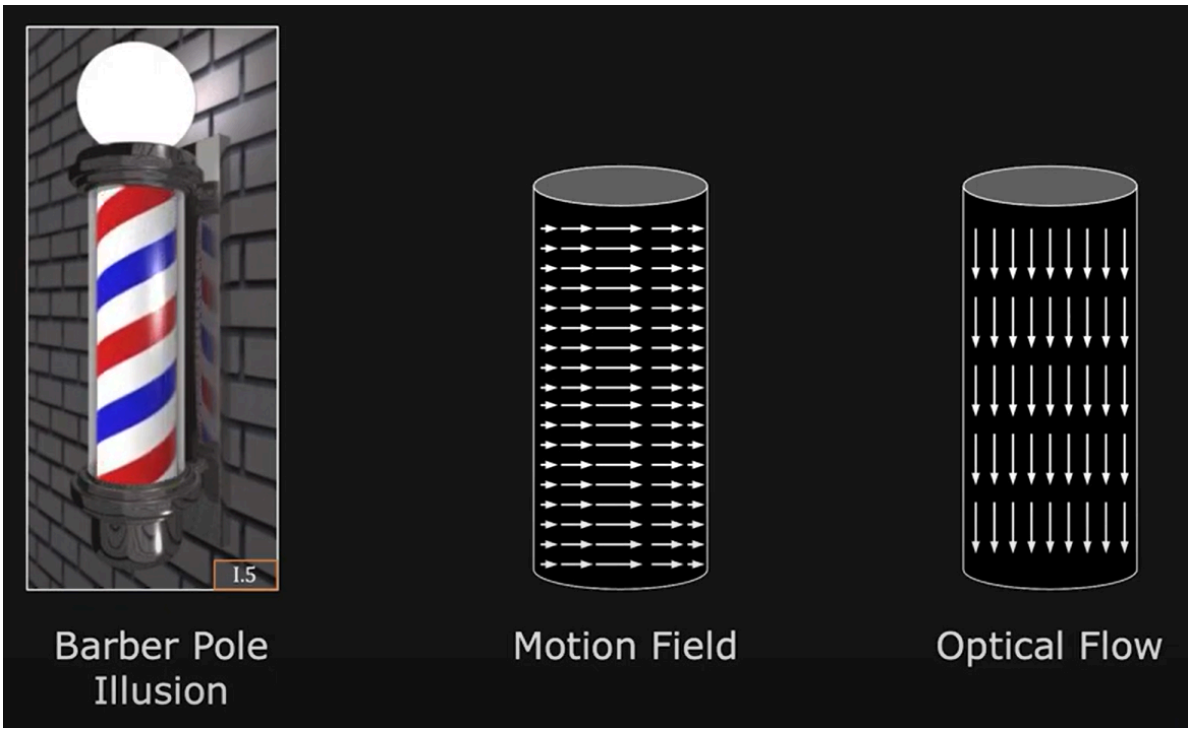
This ball is Lambertian, meaning that the apparent brightness of the surface to an observer is the same regardless of the observer's angle of view. That means that when rotating with stationary light source, no change of the intensity gradient is present, so no Optical Flow exists, while a Motion Field is of course present. In this other example is the opposite:



We have a stationary specular ball and a moving light source. Since the light moves we have a gradient change and so an Optical Flow, but the blue ball is not moving so the Motion Field is null.

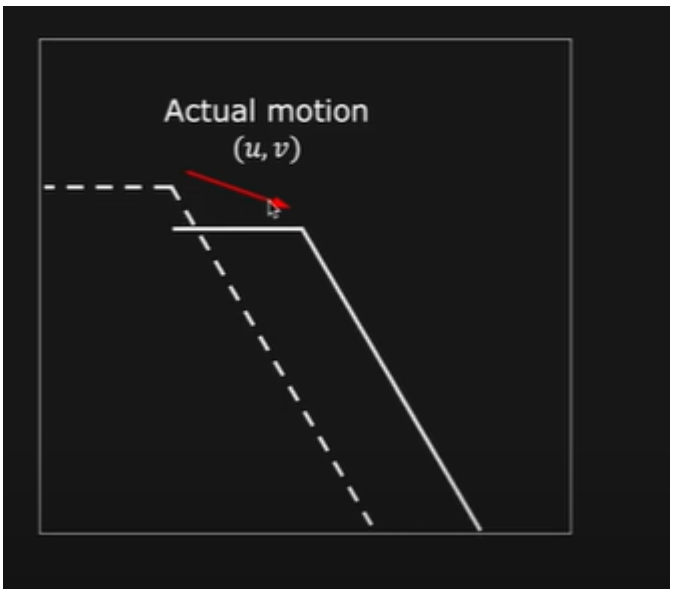
Barber's Pole Illusion

The Barber's pole illusion is another example in which motion field and optical flow are different:

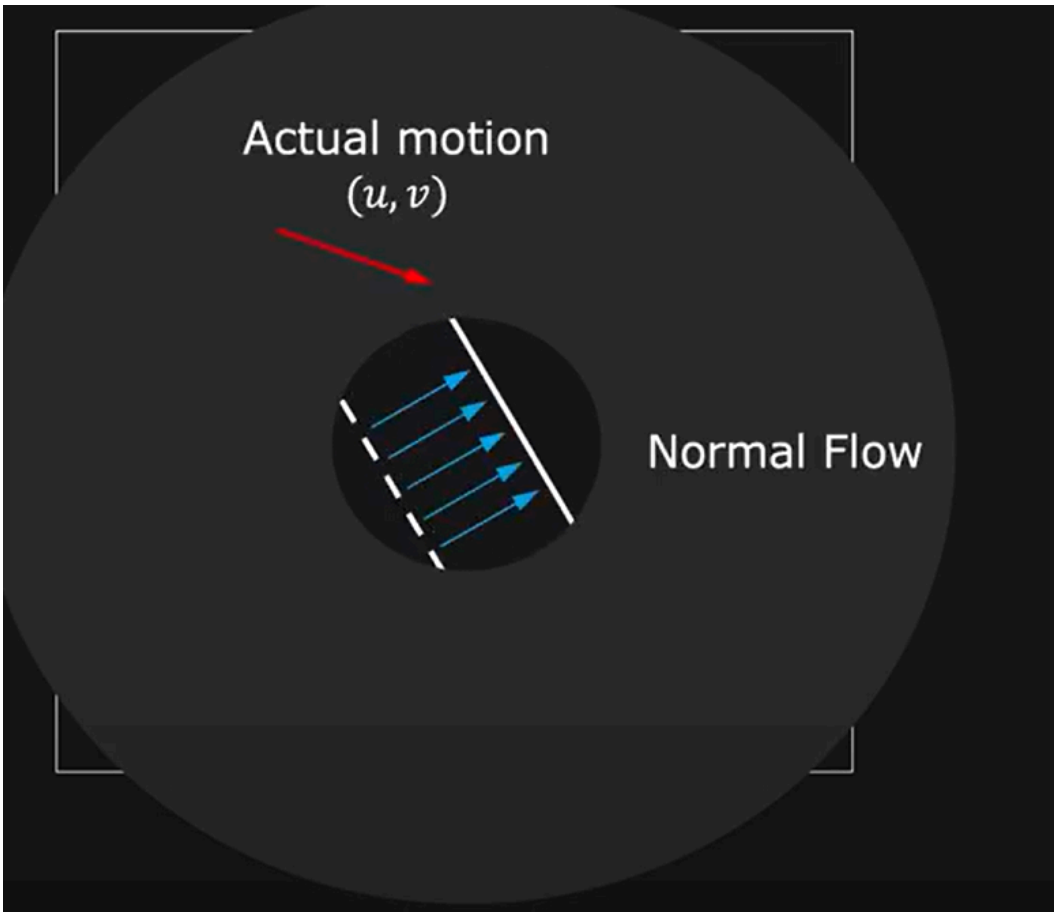


The Aperture Problem

A single observation is not enough to determine the optical flow. The problem is that the apparent motion of an object can be ambiguous when observed through a small aperture, such as a small window of pixels. *Basically the aperture problem is a phenomenon where only the motion perpendicular to an edge can be detected when motion is observed through a small aperture*. This makes it impossible to determine the actual direction of motion. Another example to explain it:

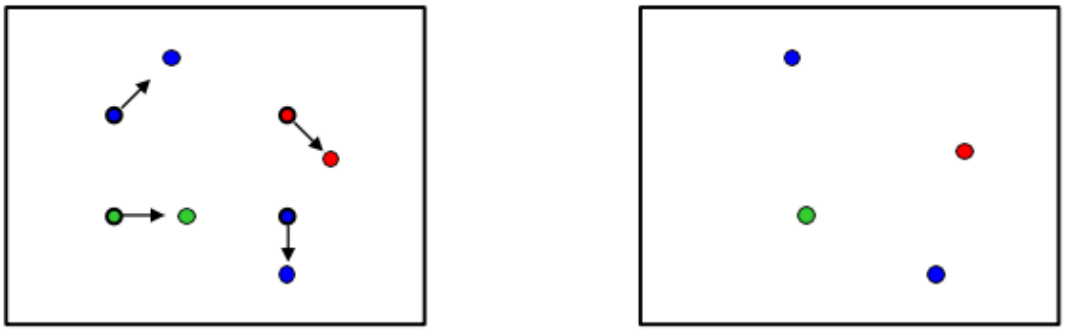


But in optical flow we are not looking the entire images but local regions, so we look at a small local patch, called apertures



So locally we can only see the normal component of the flow, not the actual one.

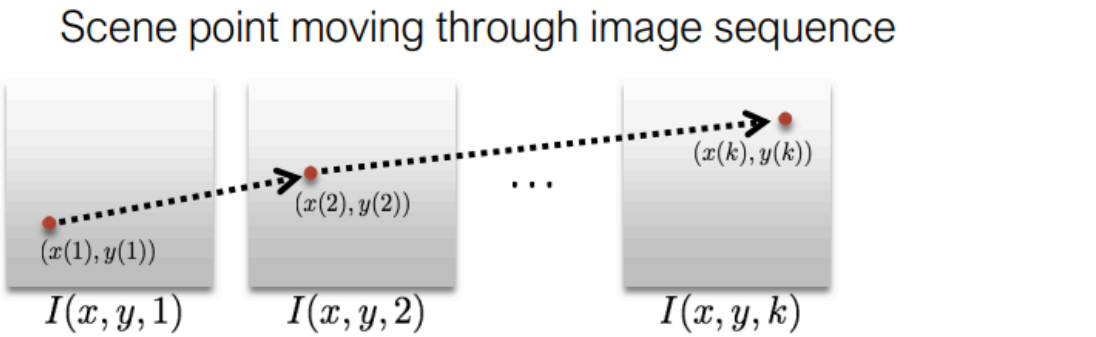
So, in general, the problem description is:



$I(x,y,t)$ $I(x,y,t')$

Given two subsequent frames, estimate the apparent motion field u(x,y), v(x,y) between them. For doing so the approach is to **look for <u>nearby pixels</u> with the <u>same color</u>**. The key assumptions of this approach are:

1. Brightness constancy **(<u>same color</u>)**

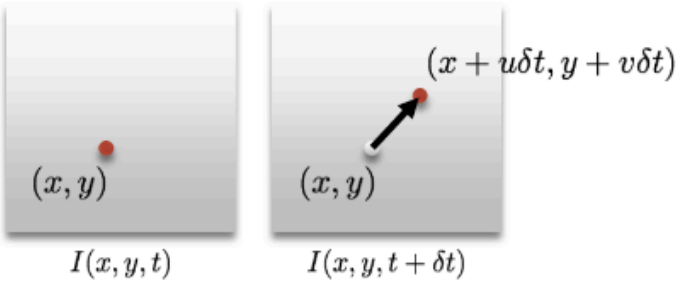


Assumption: brightness of the point will remain the same

$$I(x(t), y(t), t) = C$$

constant

2. Small motion **(<u>nearby pixels</u>)**: in one frame points do not move very fast, namely if you do a small space-time step δt, the brightness between the two images frames is the same



Optical flow (velocities): (u, v) Displacement: $(\delta x, \delta y) = (u\delta t, v\delta t)$

For a *small space-time step*

$$I(x + u\delta t, y + v\delta t, t + \delta t) = I(x, y, t)$$

the brightness between two consecutive image frames is the same

Using Taylor series expansion, for a small step we have

$$I(x + u\delta t, y + v\delta t, t + \delta t) = I(x, y, t)$$

$$\sim$$

$$I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t = I(x, y, t)$$

Then, dividing by δt we get:

$$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} = 0$$

Brightness Constancy Equatio

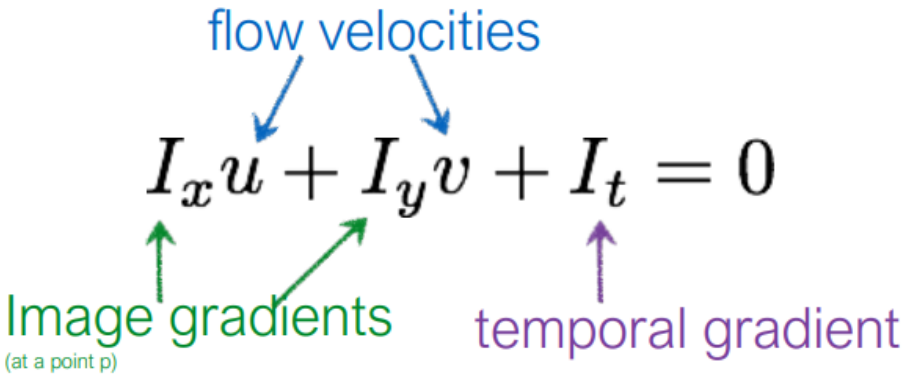
$$\underset{\text{(x-flow)}}{I_x u} + \underset{\text{(y-flow)}}{I_y v} + I_t = 0$$

shorthand notation

$$\underset{(1 \times 2)}{\nabla I^\top} \underset{(2 \times 1)}{\boldsymbol{v}} + I_t = 0$$

vector form

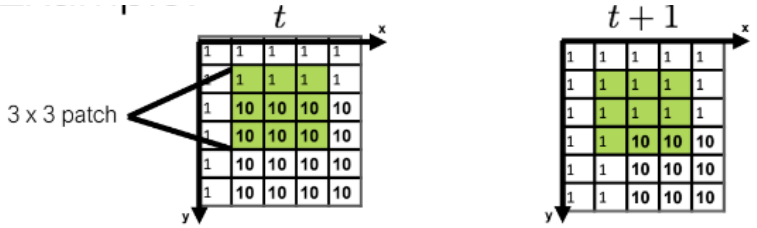
Considering the shorthand notation:



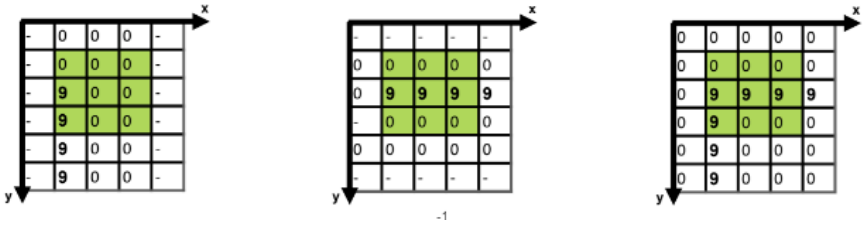
To get the **image gradients** I_x and I_y we can use:

- Forward difference
- Sobel filter
- DoG
- …

To get the **temporal gradient** I_t we can do frame differencing

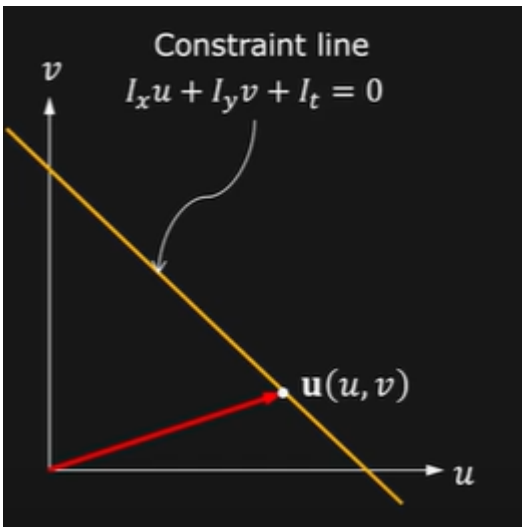


$$I_x = \frac{\partial I}{\partial x} \qquad I_y = \frac{\partial I}{\partial y} \qquad I_t = \frac{\partial I}{\partial t}$$

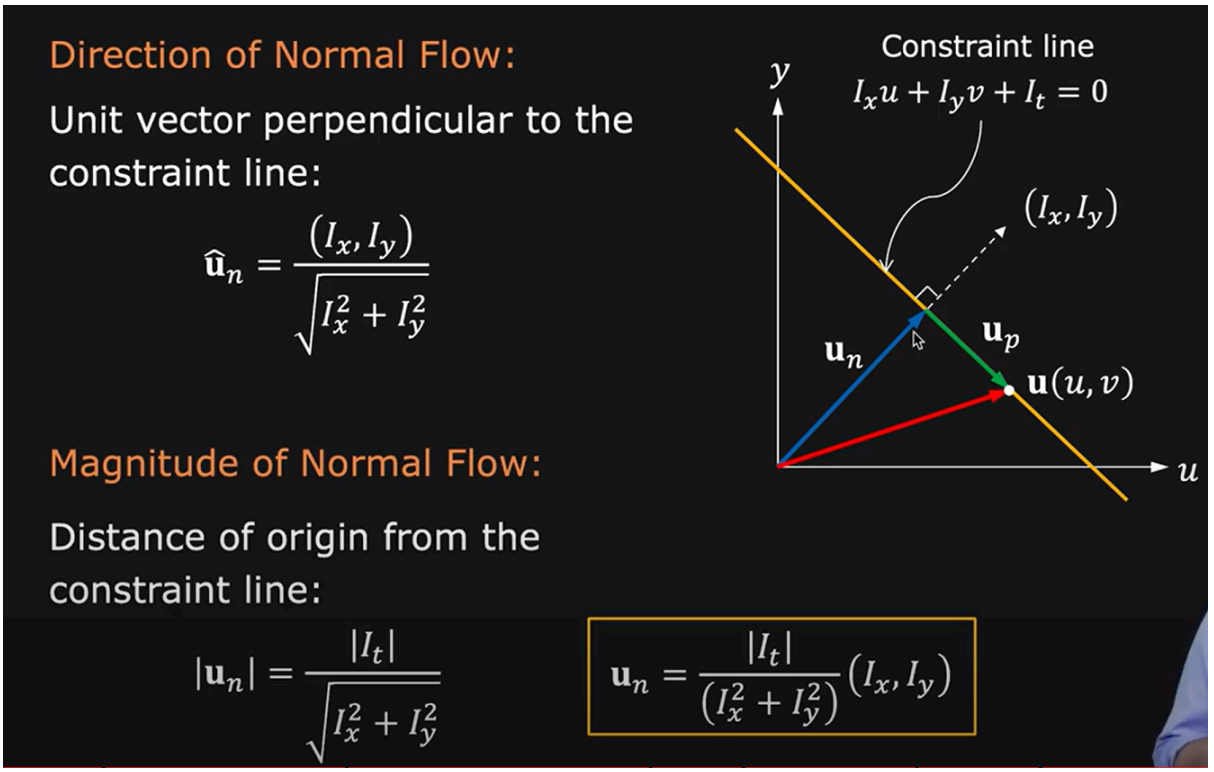


To get **flow velocities** u and v we need to solve the brightness constancy equation

Since we have two variables and just one equation, we have an infinite number solution, i.e. all the points in this line:



Optical flow can be split into two components, the normal flow and the parallel flow.



However, we cannot determine the parallel flow u_p so we can't find the line in this by finding its normal and parallel component

To find a solution, we can use the Lucas-Kanade or the Horn Schunck algorithm.

3. Spatial coherence: points move like their neighbors.

Lucas-Kanade Optical Flow

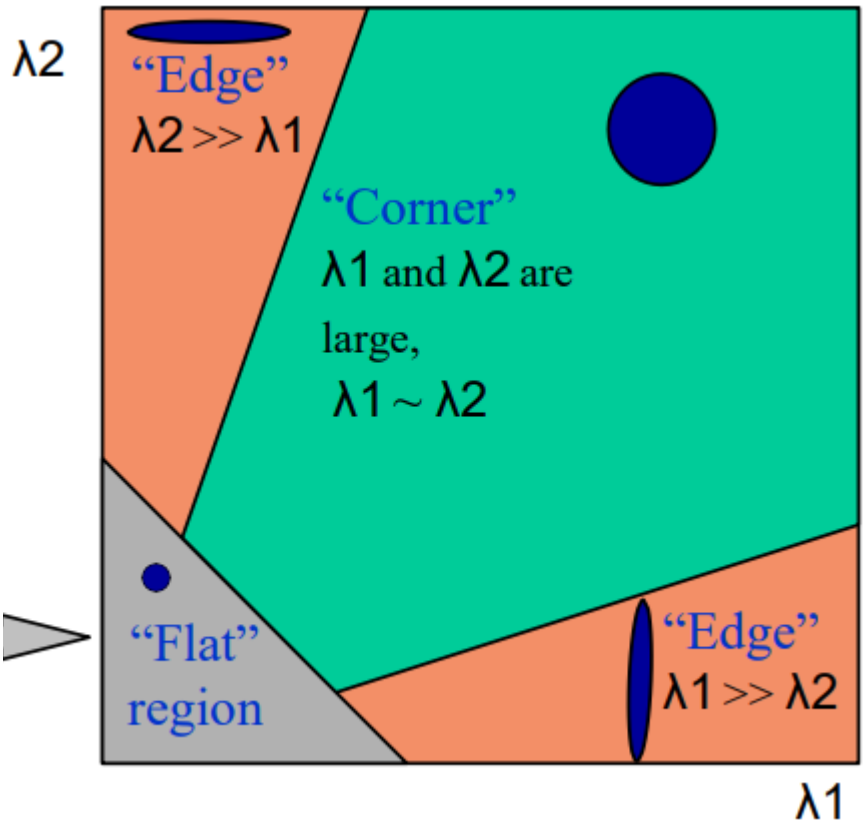
First let's assume local brightness constancy: the intensity of a pixel does not change between consecutive frames. Using a 5x5 image patch (n=5), we have $n^2 = 25$ equations:

$$I_x(\boldsymbol{p}_1)u + I_y(\boldsymbol{p}_1)v = -I_t(\boldsymbol{p}_1)$$
$$I_x(\boldsymbol{p}_2)u + I_y(\boldsymbol{p}_2)v = -I_t(\boldsymbol{p}_2)$$
$$\vdots$$
$$I_x(\boldsymbol{p}_{25})u + I_y(\boldsymbol{p}_{25})v = -I_t(\boldsymbol{p}_{25})$$

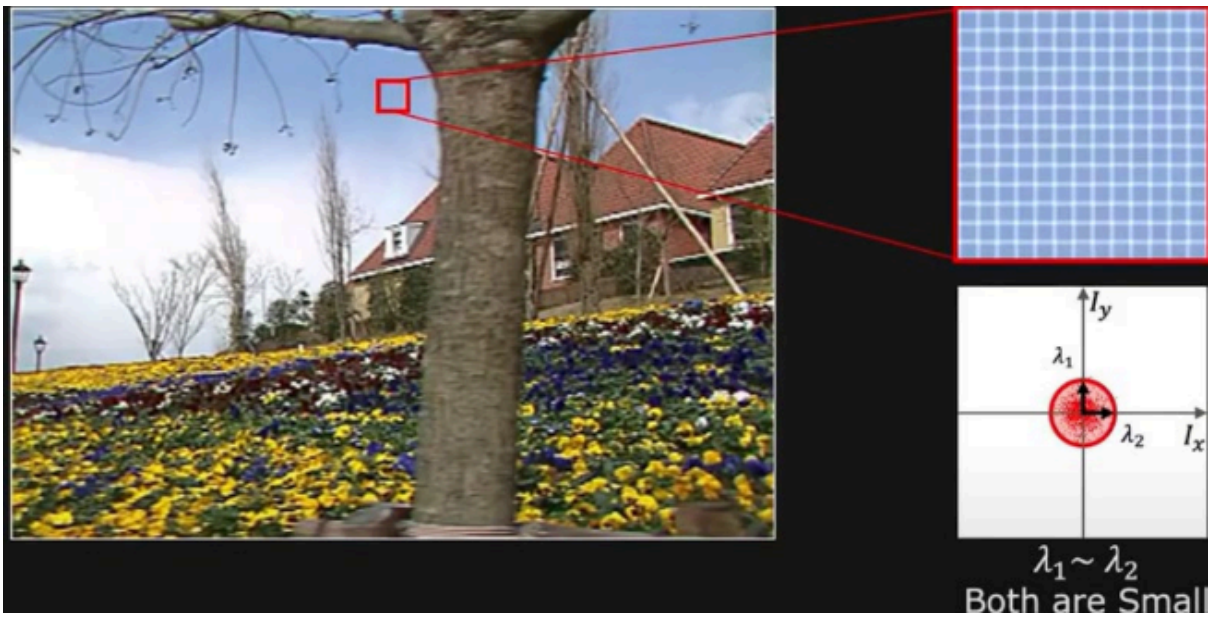
That means that the 5x5 patch has constant flow. Now we have more equations (25) than variables (2), so we can use least square solution, solving Au = b, to find u = (u,v). The solution is:

$$\mathbf{u} = (A^T A)^{-1} A^T B$$

So we should have (A^TA), the second moment matrix, invertible (det != 0) and well conditioned (its eigenvalues should not be too small and the max should not be too large w.r.t. to the other). Eigenvectors and eigenvalues of (A^TA) are related to edge detection and magnitude

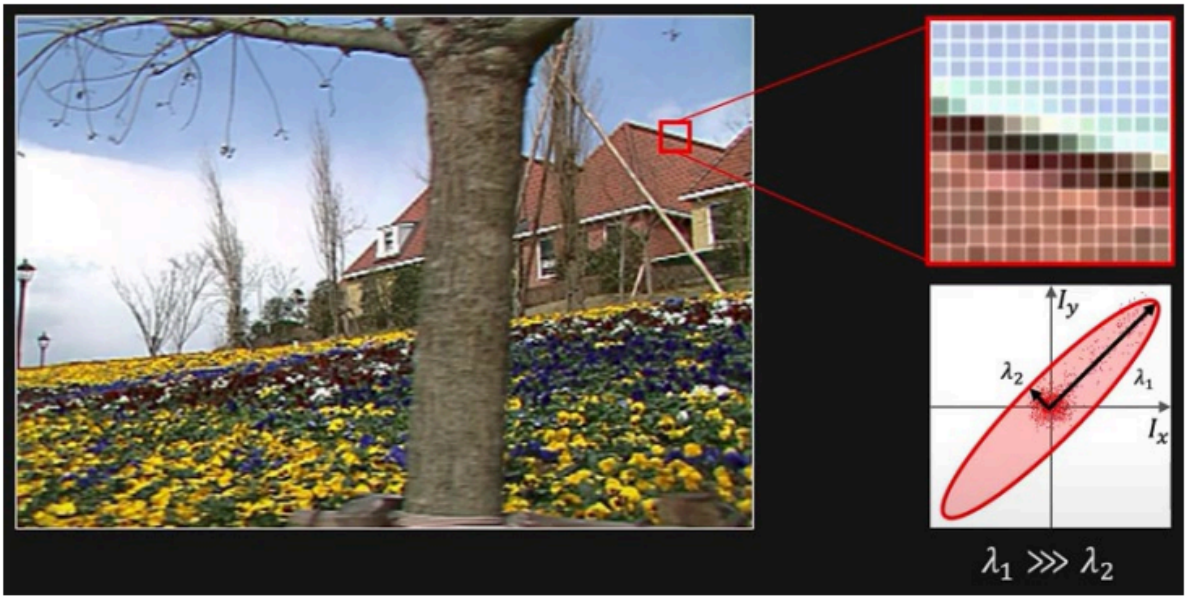


Let's consider this example:



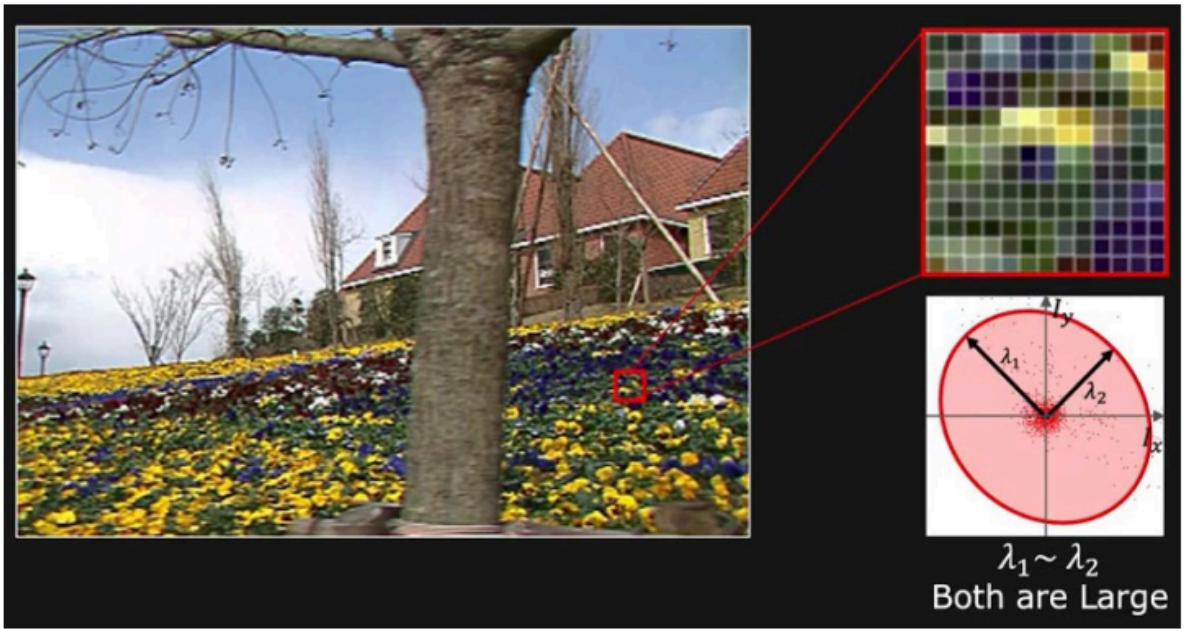
This is a flat region so the eigenvalues are both small -> Second Moments Matrix bad conditioned and so equation of rall pixels in the window are more or less the same and we cannot reliably compute the flow (u,v)

When it comes to edges, on the other hand, the gradient is prominent in one direction



We cannot reliably compute the flow here too.

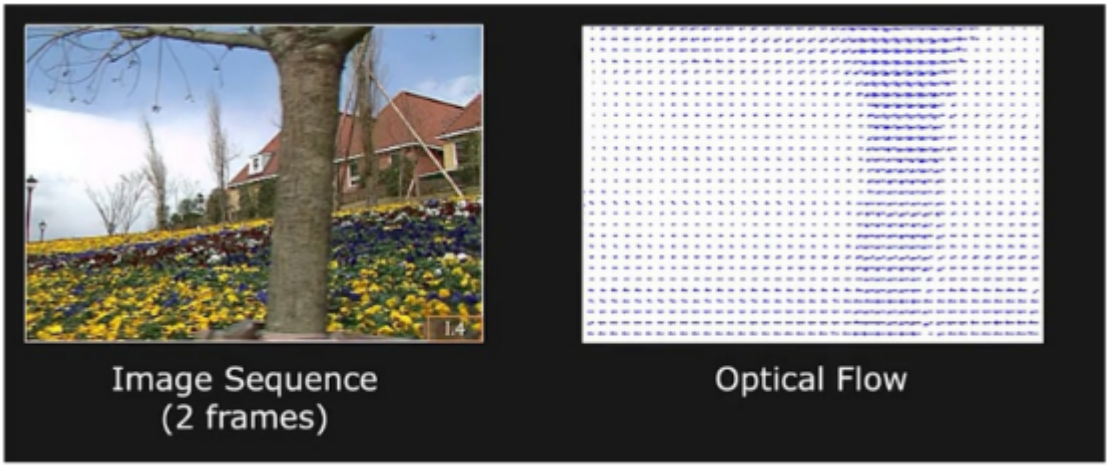
In high-texture region and corner, both eigenvalues are large and here the SMM is well conditioned and we can compute the OF



So basically **Lucas-Kanade** works best in corners and high texture regions.

Steps

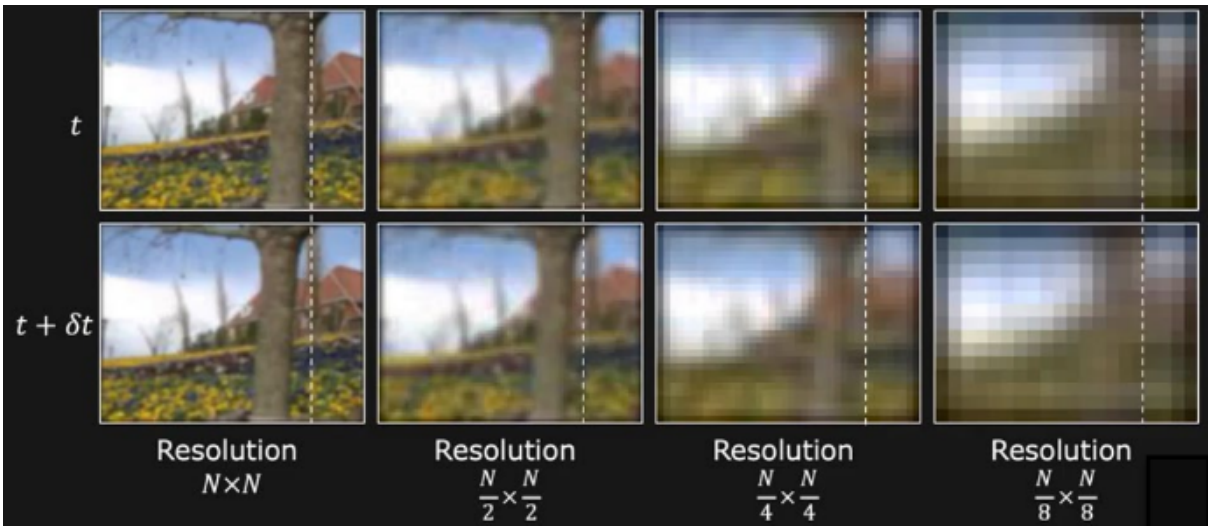
1. Choose a local neighborhood of pixels (window) and smooth with a Gaussian.
2. Calculate I_x, I_y and I_t considering time t and t+dt
3. For each couple of frames obtain equational system and find the least square solution
4. Plot the optical flow vectors:



Coarse-to-fine Flow Estimation

Small motion: points can't move very far. This is not valid all the time. In case of large motion we can't use it. For instance, consider this motion:

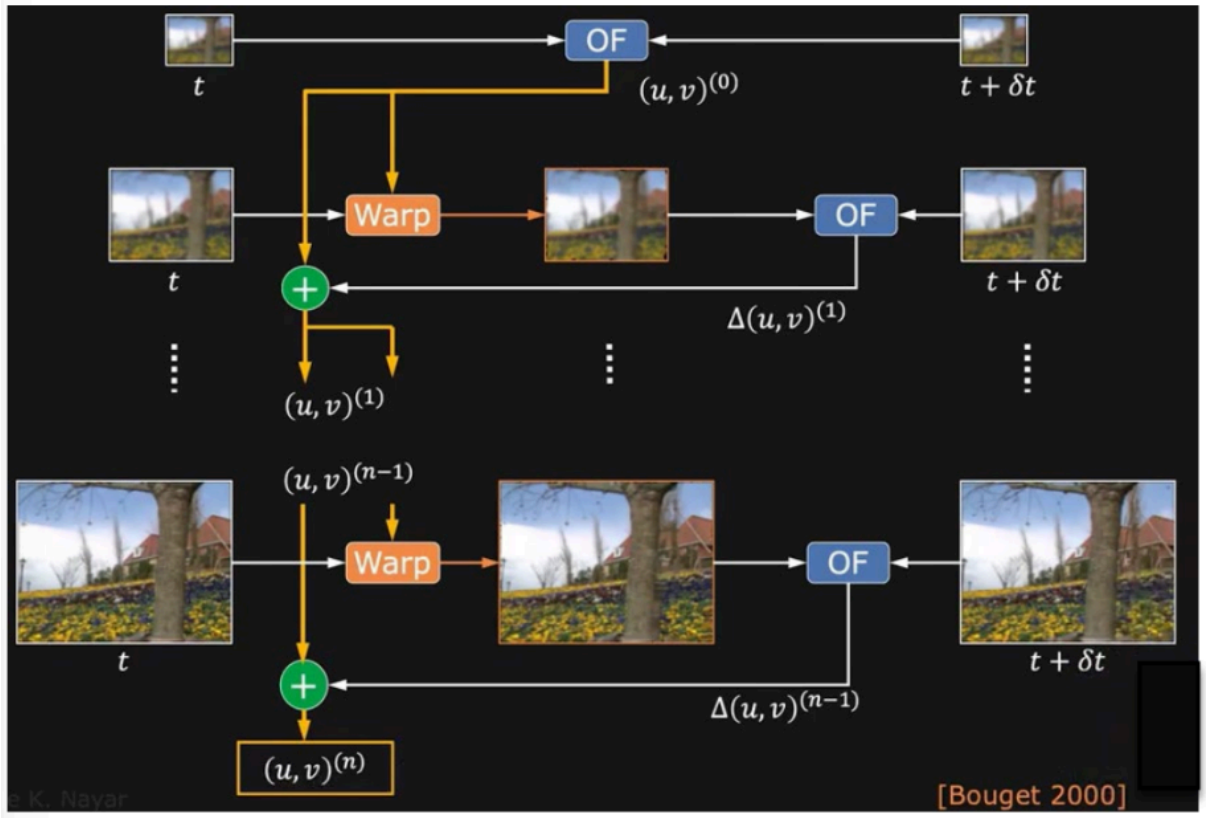
This is clearly too much large, but if we reduce the resolution we get:

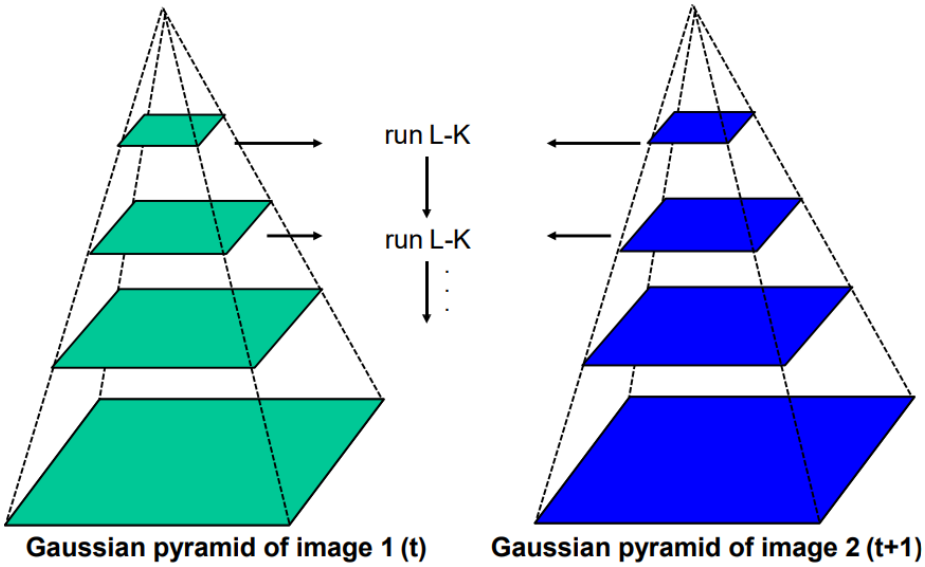


so at a certain point, by continuing decreasing the resolution, we will get a motion that is <= than one pixel and there the small assumption is valid again. Now, with less resolution, we can estimate the optical flow with low resolution and then go back.

This is the algorithm: starting from the lowest resolution, we compute the OF with LK. Then we go to the higher level of the pyramid, warping the image at time t with the OF obtained at the level before. Now this warped image is compared with the image at time t + 1, to get the

OF between them. This OF is summed with the previous to get a new OF that is used at the next level of the pyramid. This process is done up until the highest level of the pyramid, where we get the final optical flow.





Horn-Schunck Optical Flow

This method assumes:

- Smoothness of pixels: the optical flow field is smooth, meaning neighboring pixels have similar flow vectors. This is enforced by a regularization term that penalizes large differences in the flow between neighboring pixels.
- Constant of brightness for small steps

We want to minimize the following energy functional[20]:

$$E(u,v) = \iint \underbrace{(I(x+u(x,y),y+v(x,y),t+1)-I(x,y,t))^2}_{\text{quadratic penalty for brightness change}} + \lambda \cdot \underbrace{\left(\|\nabla u(x,y)\|^2 + \|\nabla v(x,y)\|^2\right)}_{\text{quadratic penalty for flow change}} dx\,dy$$

with regularization parameter λ and gradient $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)$.

The reliance on a quadratic penalty for smoothness is a double-edged sword. While it helps produce smooth and coherent motion fields, it also struggles with scenarios where there are discontinuities in the flow field. Penalizing this changes too much, it causes oversmoothing.

We can't minimize the energy functional directly since the energy is highly non-convex and has many local optima. So we linearize ***assuming small steps***:

$$\begin{aligned} & I(x+u(x,y),y+v(x,y),t+1) \\ \overset{x,y,t}{\approx}\ & I(x,y,t) + I_x(x,y,t)(x+u(x,y)-x) \\ & + I_y(x,y,t)(y+v(x,y)-y) + I_t(x,y,t)(t+1-t) \\ =\ & I(x,y,t) + I_x(x,y,t)\,u(x,y) + I_y(x,y,t)\,v(x,y) + I_t(x,y,t) \end{aligned}$$

So we get:

$$E(u,v) = \iint (I_x(x,y,t)u(x,y) + I_y(x,y,t)v(x,y) + I_t(x,y,t))^2 + \lambda \cdot \left(\|\nabla u(x,y)\|^2 + \|\nabla v(x,y)\|^2\right) dx\,dy$$

And its discretized version

[20] Note that the integral is covering both addends

$$
\begin{aligned}
E(\mathbf{U},\mathbf{V}) &= \sum_{x,y} \left(I_x(x,y)\,u_{x,y} + I_y(x,y)\,v_{x,y} + I_t(x,y)\right)^2 \\
&+ \lambda \cdot \left((u_{x,y} - u_{x+1,y})^2 + (u_{x,y} - u_{x,y+1})^2 + (v_{x,y} - v_{x+1,y})^2 + (v_{x,y} - v_{x,y+1})^2\right)
\end{aligned}
$$

This object is quadratic in the flow maps U,V and **thus has a unique optimum**. We differentiate w.r.t. to **U** and **V** and set the gradient to 0, getting a huge but sparse linear system that can be solved using standard techniques. Problem: as in LK, the linearization works only for small motions.

The minimization of the energy functional is typically solved using an iterative approach, with update rules derived from the Euler-Lagrange equations:

1. Precompute image gradients $I_y \quad I_x$
2. Precompute temporal gradients I_t
3. Initialize flow field $\boldsymbol{u} = \mathbf{0}$ $\boldsymbol{v} = \mathbf{0}$
4. While not converged

   Compute flow field updates for each pixel:

$$
\hat{u}_{kl} = \bar{u}_{kl} - \frac{I_x\bar{u}_{kl} + I_y\bar{v}_{kl} + I_t}{\lambda^{-1} + I_x^2 + I_y^2} I_x \qquad \hat{v}_{kl} = \bar{v}_{kl} - \frac{I_x\bar{u}_{kl} + I_y\bar{v}_{kl} + I_t}{\lambda^{-1} + I_x^2 + I_y^2} I_y
$$

For large motions, we can do a coarse-to-fine estimation.

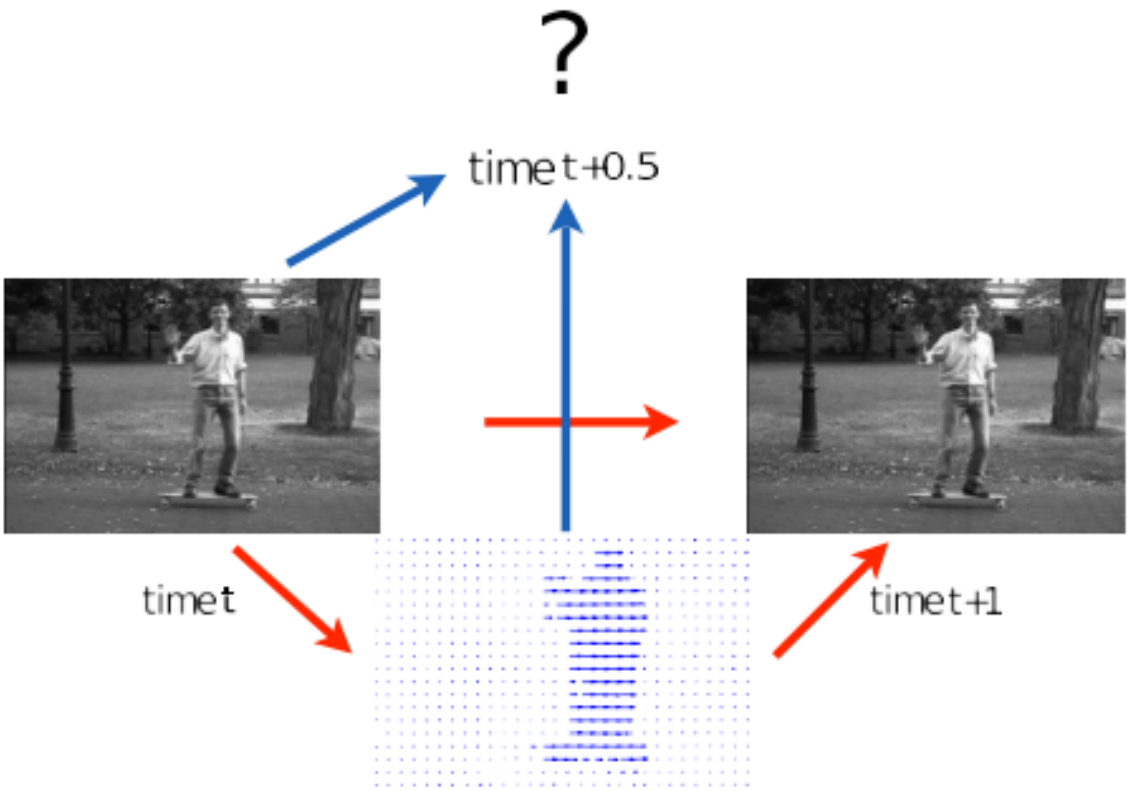
Robust Estimation of Optical Flow - Probabilistic Interpretation

The Horn-Schunck optimization problem can be interpreted as Maximum A Posteriori (MAP) inference in a Markov Random Fields (MRF), with the Gibbs energy

$$
\begin{aligned}
E(\mathbf{U},\mathbf{V}) &= \sum_{x,y} \left(I_x(x,y)\,u_{x,y} + I_y(x,y)\,v_{x,y} + I_t(x,y)\right)^2 \\
&+ \lambda \cdot \left((u_{x,y} - u_{x+1,y})^2 + (u_{x,y} - u_{x,y+1})^2 + \right. \\
&\qquad \left. (v_{x,y} - v_{x+1,y})^2 + (v_{x,y} - v_{x,y+1})^2\right)
\end{aligned}
$$

Application of Optical Flow

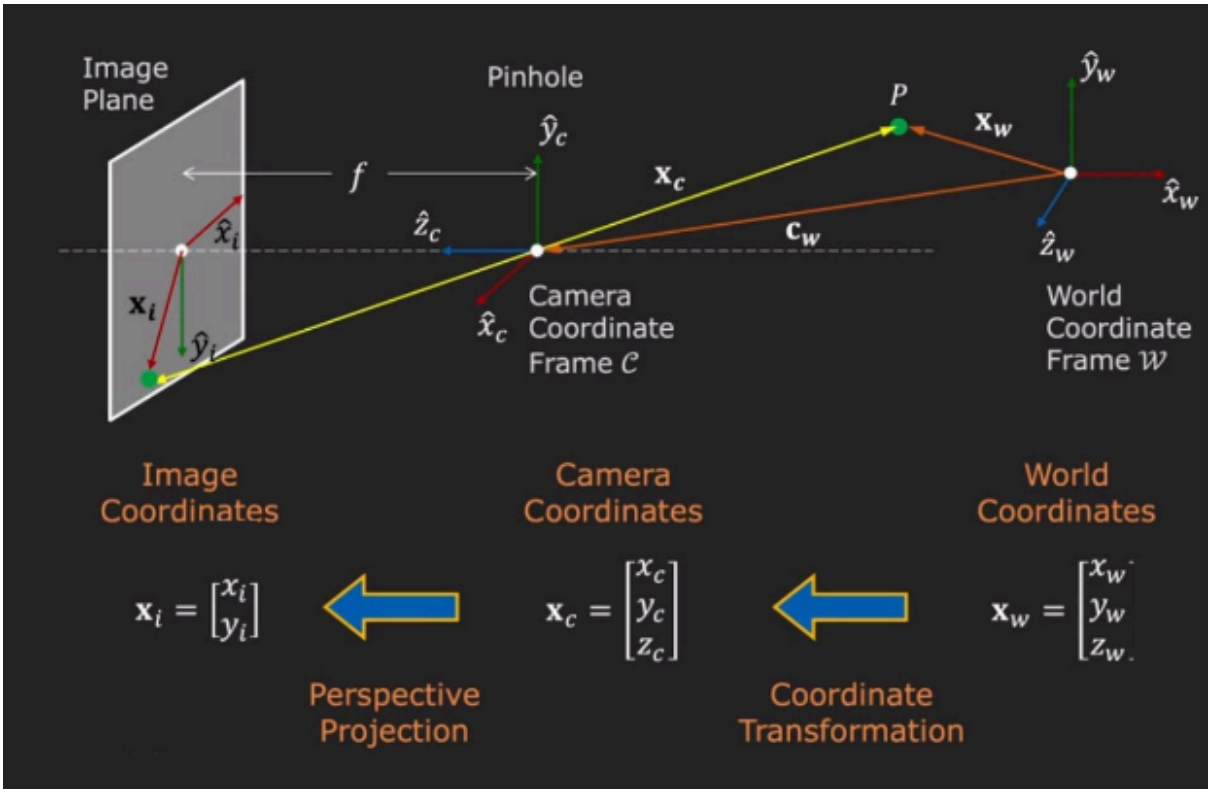
Optical Flow can be used for Video Interpolation and Frame Rate Adaptation. For instance, if we know the image motion we can compute images at intermediate frames.



Also autonomous driving is a field in which you can apply optical flow.

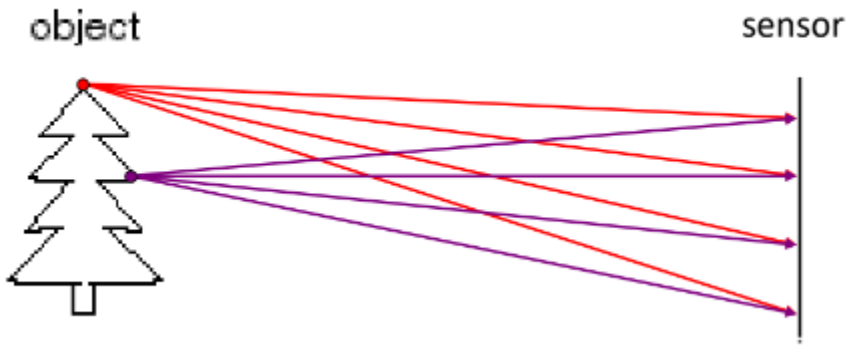
Cameras and Camera Calibration

Cameras are dimensionality reduction machines, as they take the 3D world and turn it into 2D. Basically, they are performing a projective transformation. Camera calibration is a way of finding the camera's internal and external parameters



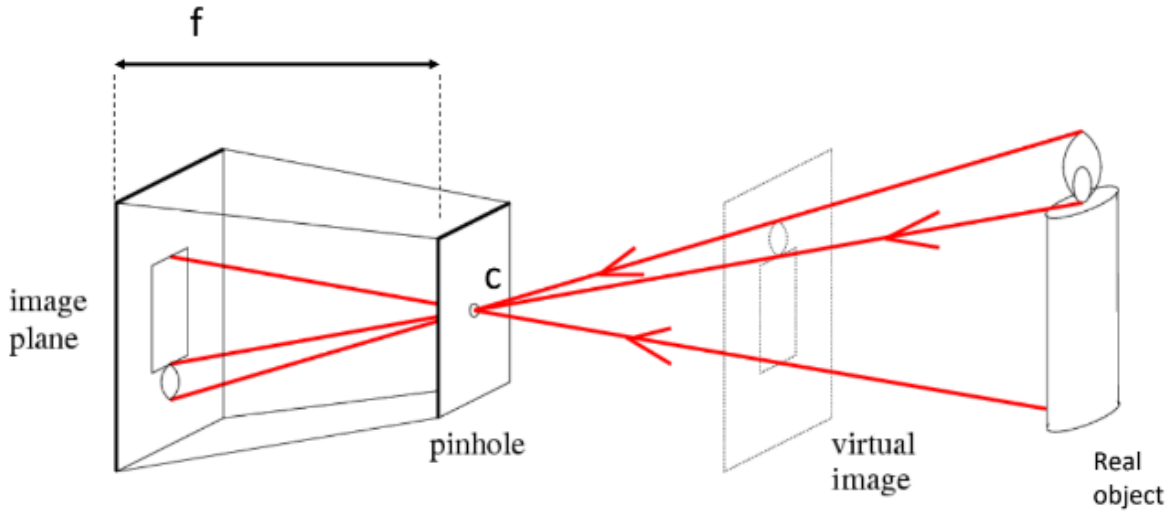
Design a Camera

Idea 1: naively put a sensor in front of an object. The problem is that light rays from every point in the 3D world hit every point on the sensor, which results in a blurry image



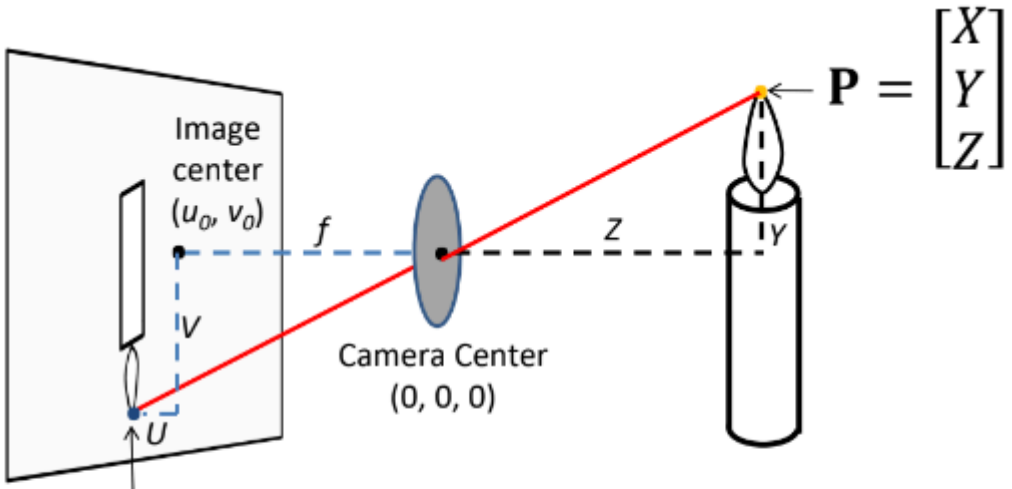
Idea 2: add a barrier to block most rays. This barrier has a pinhole, that is a small hole, from which the light can pass: in this way the sensor senses light from one direction, reducing the blurring. That leads us to the pinhole camera model.

Pinhole Camera Model



f = Focal length
c = Optical center of the camera

The camera is a box with a tiny hole, called a pinhole. Opposite the pinhole is the image plane where the image is formed. The light rays enter through the pinhole in the optical center point “c”. The distance between the pinhole and the image plane is the focal length “f”. The image formed on the image plane is inverted w.r.t. to the real object.



$$\mathbf{p} = \begin{bmatrix} U \\ V \end{bmatrix}$$

f = distance from image center

$$U = -X * \frac{f}{Z} \qquad V = -Y * \frac{f}{Z}$$

3D points in camera coordinates, that have its origin in c, are mapped to the image plane by dividing them by their Z component and multiplying with the focal length.

Ideal Pinhole Size

It must be tiny, but too tiny it will cause diffraction

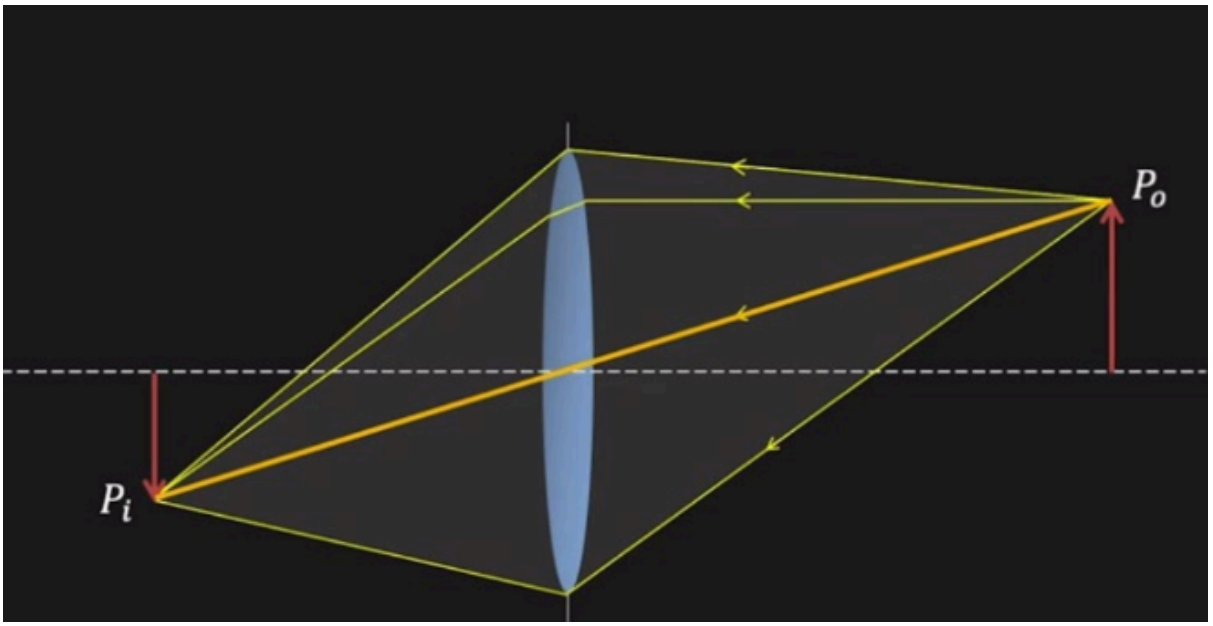
with f effective focal length and lambda wavelength

Exposure

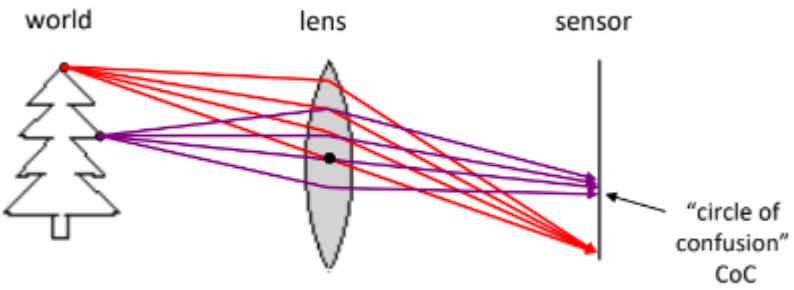
Pinhole passes less light so we need long exposure to capture bright images.

Lenses - Focus and Defocus

A lens does the same projection as a pinhole, but gathers more light.



It focuses light onto the sensor, but only from objects at a specific distance, that are indeed, “in focus”. In fact, as you can see from the red arrows, points in focus have light rays that converge in a single point on the sensor, creating a sharp image. On the other hand, points at a different distance are not in focus and their light rays are projected to a circle of confusion CoC, and so the light rays do not converge at a single point in the image plane.



Changing the shape of the lens changes this distance

Gaussian Lens (Thin Lens) Law

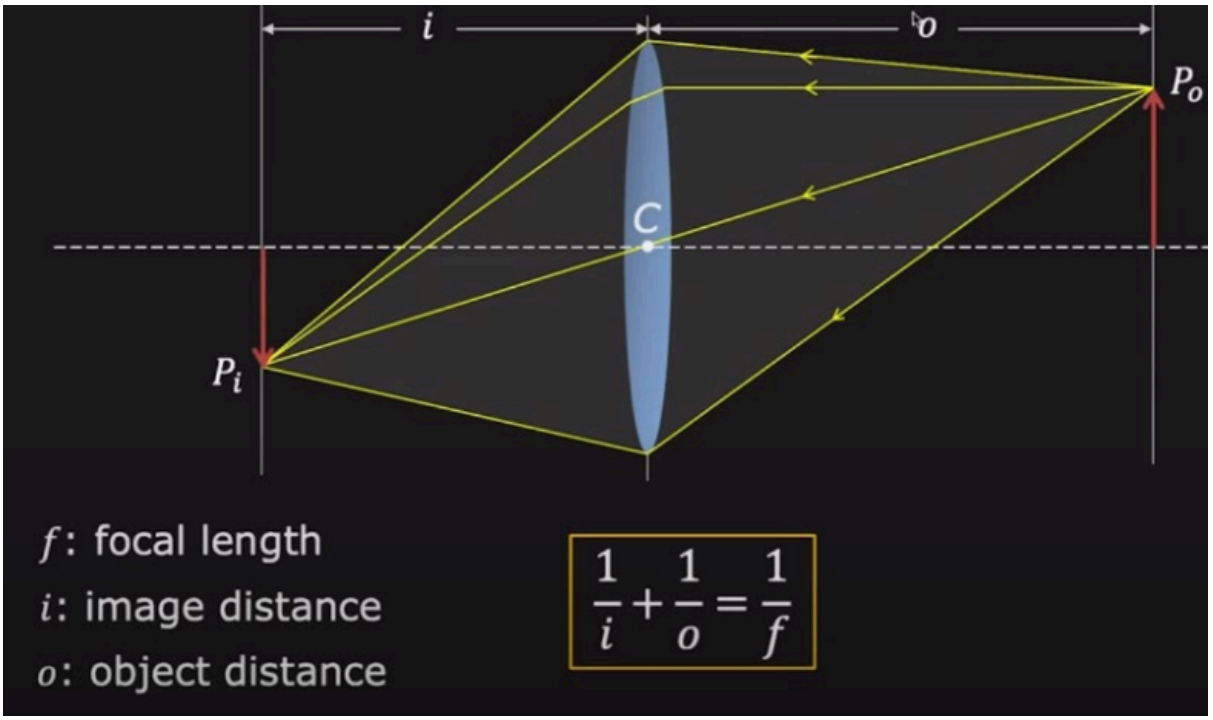
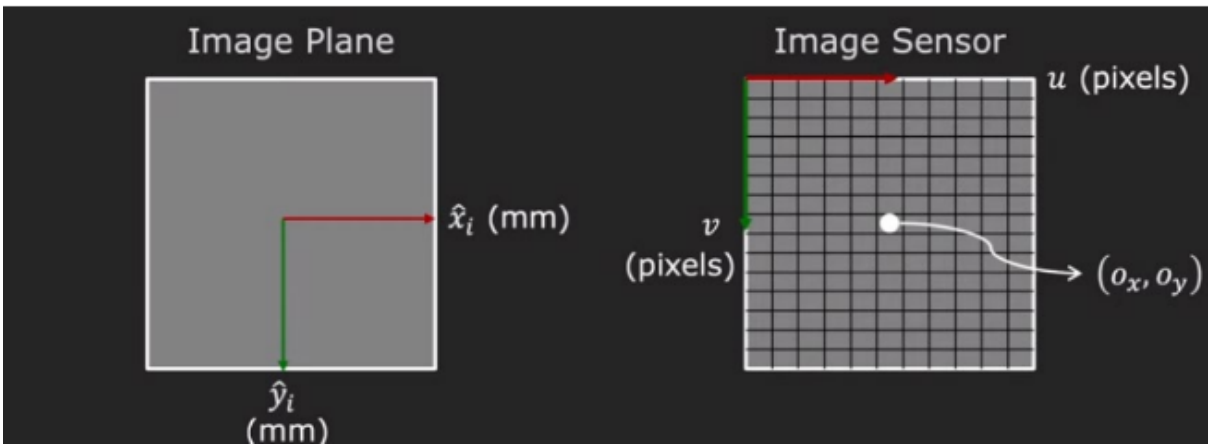


Image Plane to Image Sensor Mapping



Considering the top-left corner as the origin of the image sensor, and considering the pixel (o_x, o_y) as the principle point where the optical axis pierces the sensor, we have the following transformation:

$$u = m_x f \frac{x_c}{z_c} + o_x \qquad v = m_y f \frac{y_c}{z_c} + o_y$$

$$u = f_x \frac{x_c}{z_c} + o_x \qquad v = f_y \frac{y_c}{z_c} + o_y$$

Where (f_x, f_y) = (m_x* f, m_y* f) are the focal lengths in pixels. Now (f_x, f_y, o_x, o_y) are the intrinsic parameters of the camera, namely its internal geometry. The problem with these equations is that they are non-linear. We can express them as linear equations using homogeneous coordinates

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} \equiv \begin{bmatrix} z_c u \\ z_c v \\ z_c \end{bmatrix} = \begin{bmatrix} f_x x_c + z_c o_x \\ f_y y_c + z_c o_y \\ z_c \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

where: $(u, v) = (\tilde{u}/\tilde{w}, \tilde{v}/\tilde{w})$

Intrinsic Matrix

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

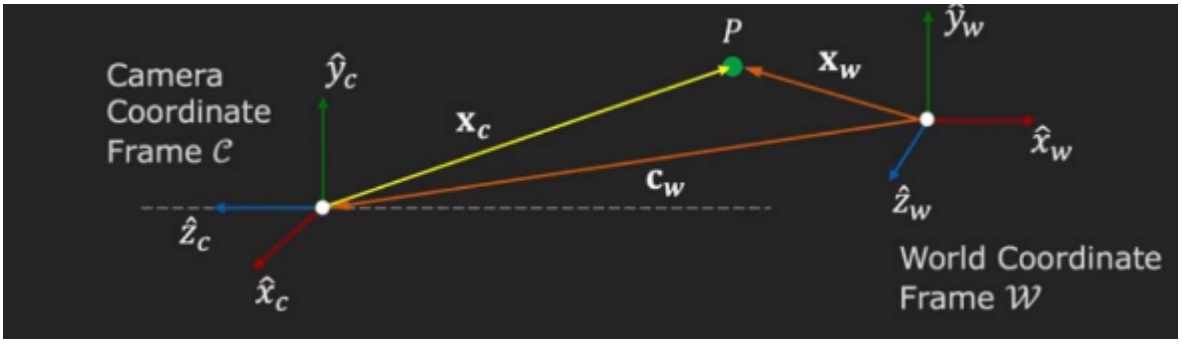
The matrix for going from the homogeneous camera coordinates to the homogeneous image plane is the intrinsic matrix

$$M_{int} = [K|0] = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

K is the calibration matrix, an upper right triangular matrix

$$\tilde{\mathbf{u}} = [K|0]\,\tilde{\mathbf{x}}_c = M_{int}\,\tilde{\mathbf{x}}_c$$

Extrinsic Matrix



We can express position C_w and orientation R of the camera w.r.t. the world coordinate frame W. These two are the camera's extrinsic parameters.

The R matrix is the following:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{array}{l} \rightarrow \text{Row 1: Direction of } \hat{x}_c \text{ in world coordinate frame} \\ \rightarrow \text{Row 2: Direction of } \hat{y}_c \text{ in world coordinate frame} \\ \rightarrow \text{Row 3: Direction of } \hat{z}_c \text{ in world coordinate frame} \end{array}$$

Is an orthonormal matrix. Now that we have the extrinsic parameters, we can represent an arbitrary point in the camera frame using the extrinsic parameters

$$\mathbf{x}_c = R(\mathbf{x}_w - \mathbf{c}_w) = R\mathbf{x}_w - R\mathbf{c}_w = R\mathbf{x}_w + \mathbf{t}$$

$$\mathbf{x}_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

From which

$$\tilde{\mathbf{x}}_c = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Extrinsic Matrix: $M_{ext} = \begin{bmatrix} R_{3\times3} & \mathbf{t} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$$\tilde{\mathbf{x}}_c = M_{ext}\tilde{\mathbf{x}}_w$$

Projection Matrix P

Combining Camera to Pixel with Intrinsic Matrix and World to Camera with Extrinsic Matrix

$$\tilde{\mathbf{u}} = M_{int}\,\tilde{\mathbf{x}}_c \qquad \tilde{\mathbf{x}}_c = M_{ext}\tilde{\mathbf{x}}_w$$

We get:

$$\tilde{\mathbf{u}} = M_{int}M_{ext}\,\tilde{\mathbf{x}}_w = P\,\tilde{\mathbf{x}}_w$$

With P projection matrix

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Image Magnification

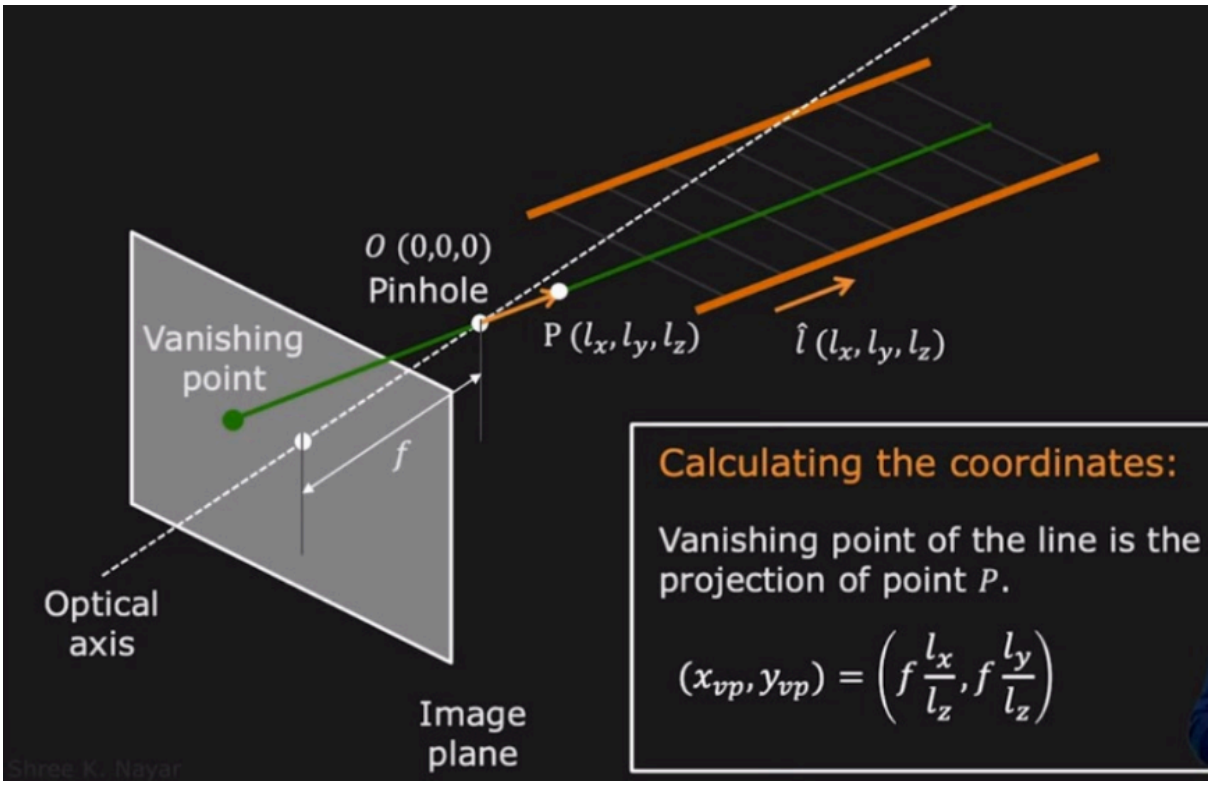
The image size is inversely proportional to depth, so $m = f / z_0$

Vanishing Point

A vanishing point is a point on the image plane of a perspective rendering where the two-dimensional perspective projections of mutually parallel lines in three-dimensional space appear to converge.

The lines of the image below are parallel in real life, while in the image plane appear to converge. The vanishing point is the point in which they appear to converge.

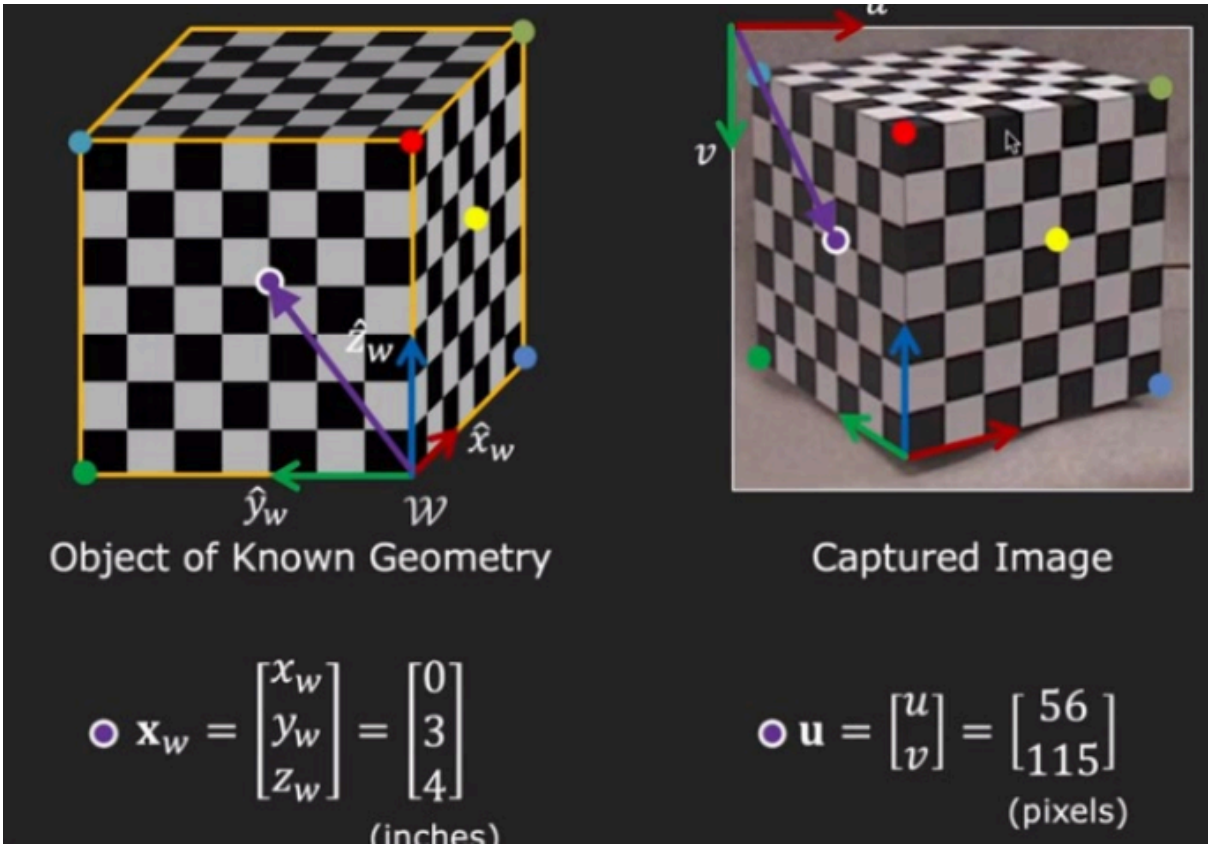
How to calculate the coordinates of this point?



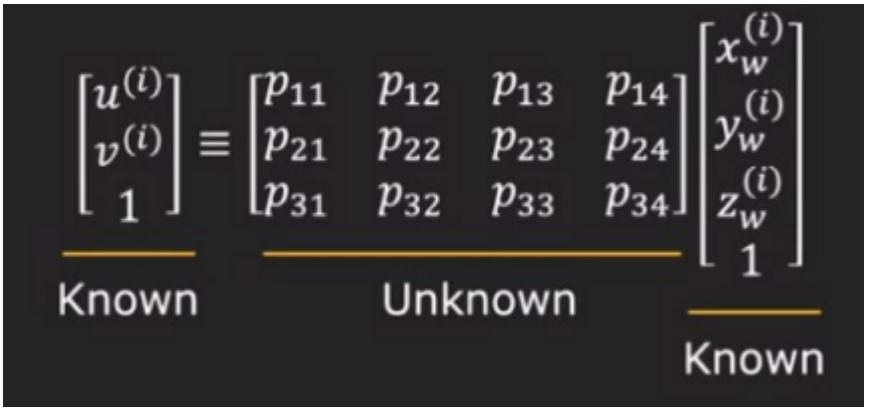
Camera Calibration Procedure

Step 1: Capture an image of an object of known geometry, so that you can find the correspondence between 3D points and image points, e.g. cube with checkered grid

Step 2: identify correspondence between 3D scene points and image points



Step 3: for each point i



expand the matrix as linear equation

$$u^{(i)} = \frac{p_{11}x_w^{(i)} + p_{12}y_w^{(i)} + p_{13}z_w^{(i)} + p_{14}}{p_{31}x_w^{(i)} + p_{32}y_w^{(i)} + p_{33}z_w^{(i)} + p_{34}}$$

$$v^{(i)} = \frac{p_{21}x_w^{(i)} + p_{22}y_w^{(i)} + p_{23}z_w^{(i)} + p_{24}}{p_{31}x_w^{(i)} + p_{32}y_w^{(i)} + p_{33}z_w^{(i)} + p_{34}}$$

Step 4: rearrange the terms in order to have Ap = 0

$$\underbrace{\begin{bmatrix} x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & 0 & 0 & 0 & 0 & -u_1x_w^{(1)} & -u_1y_w^{(1)} & -u_1z_w^{(1)} & -u_1 \\ 0 & 0 & 0 & 0 & x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & -v_1x_w^{(1)} & -v_1y_w^{(1)} & -v_1z_w^{(1)} & -v_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & 0 & 0 & 0 & 0 & -u_ix_w^{(i)} & -u_iy_w^{(i)} & -u_iz_w^{(i)} & -u_i \\ 0 & 0 & 0 & 0 & x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & -v_ix_w^{(i)} & -v_iy_w^{(i)} & -v_iz_w^{(i)} & -v_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & 0 & 0 & 0 & 0 & -u_nx_w^{(n)} & -u_ny_w^{(n)} & -u_nz_w^{(n)} & -u_n \\ 0 & 0 & 0 & 0 & x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & -v_nx_w^{(n)} & -v_ny_w^{(n)} & -v_nz_w^{(n)} & -v_n \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{bmatrix}}_{\mathbf{p}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Known

Unknowr

Step 5: solve Ap = 0 for p

Scale of Projection Matrix

Since the P matrix acts on homogeneous coordinates, and it's true that

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} \equiv k \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} \quad (k \neq 0 \text{ is any constant})$$

We have:

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \equiv k \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Therefore, P and kP produce the same homogeneous pixel coordinates. Projection matrix P is defined only up to a scale.

Least Square Solution for matrix P

We have two options:

1. Set scale so that $p_{34} = 1$
2. Set scale so that $||p||^2 = 1$

Let's consider the second option.

We want Ap as close to 0 as possible and $||p||^2 = 1$, so

$$\min_{\mathbf{p}} \|A\mathbf{p}\|^2 \text{ such that } \|\mathbf{p}\|^2 = 1$$

Rewriting norm(x) as sqrt(x^Tx)

$$\min_{\mathbf{p}} (\mathbf{p}^T A^T A \mathbf{p}) \text{ such that } \mathbf{p}^T\mathbf{p} = 1$$

From this expression above, we can write the loss function

$$L(\mathbf{p}, \lambda) = \mathbf{p}^T A^T A \mathbf{p} - \lambda(\mathbf{p}^T\mathbf{p} - 1)$$

Deriving it w.r.t. p and setting it to 0 we got:

$$2A^T A\mathbf{p} - 2\lambda\mathbf{p} = \mathbf{0}$$

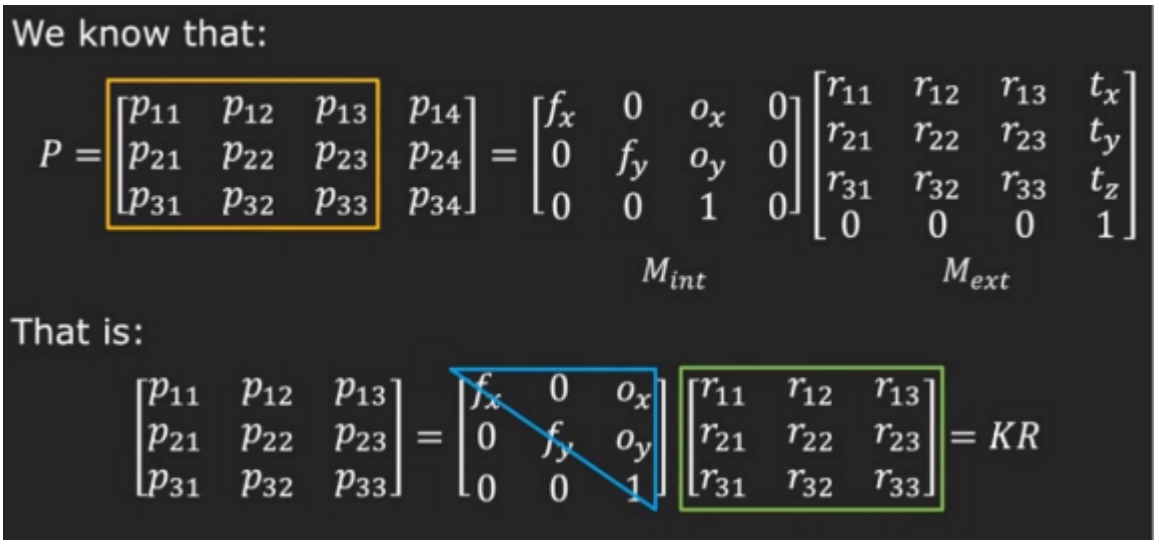
Now from here, we got:

$$A^T A\mathbf{p} = \lambda\mathbf{p}$$

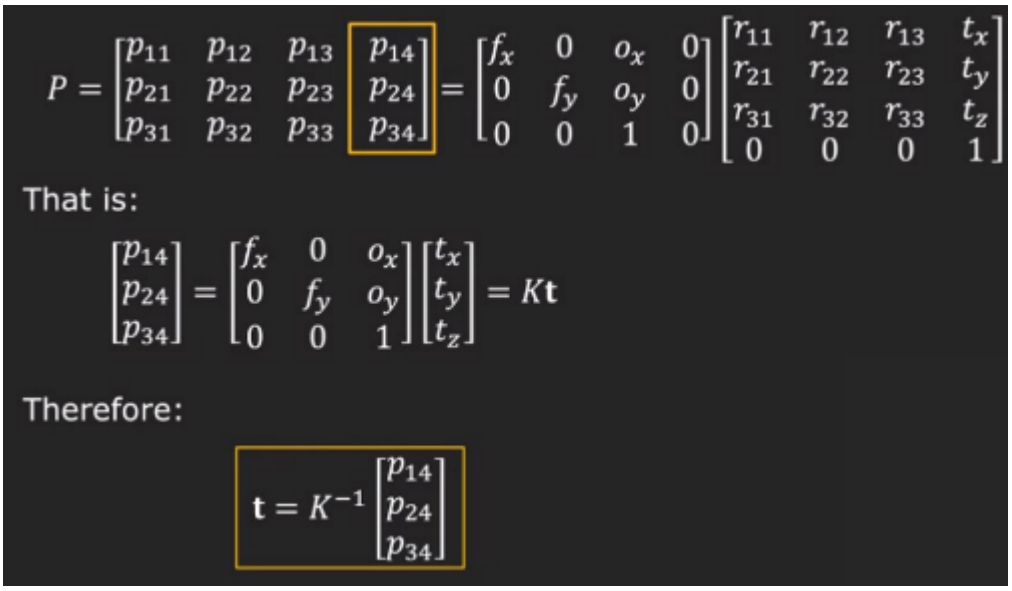
Eigenvector p with the smallest eigenvalue lambda of matrix A^TA[21] minimizes the loss function L(p). Now with vector p you can obtain the projection matrix P.

[21] Second Moment Matrix

How to get Intrinsic and Extrinsic parameters from P



Since K is the Upper Right Triangular matrix and R is Orthonormal, we can decouple K and R from their product P using QR factorization. Considering now the last column:



To find the camera center c, we do the following

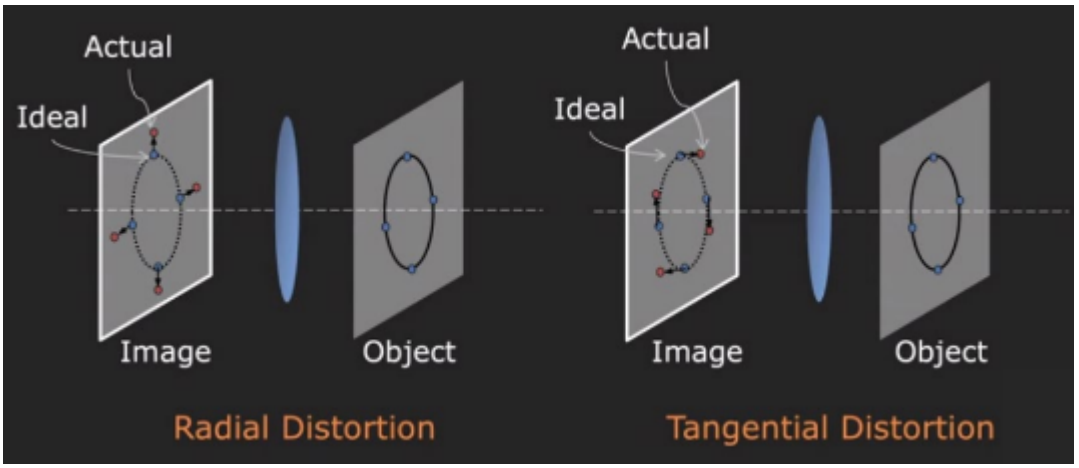
$$\mathbf{Pc} = \mathbf{0}$$

c is the singular vector corresponding to the smallest singular value.

Now that we have found these quantities, our cameras are calibrated.

Other Intrinsic Parameters: Distortion Coefficients

Pinholes do not exhibit image distortions, but lenses do



The intrinsic model of the camera will need to include these distortion coefficients, that we have ignored until now.

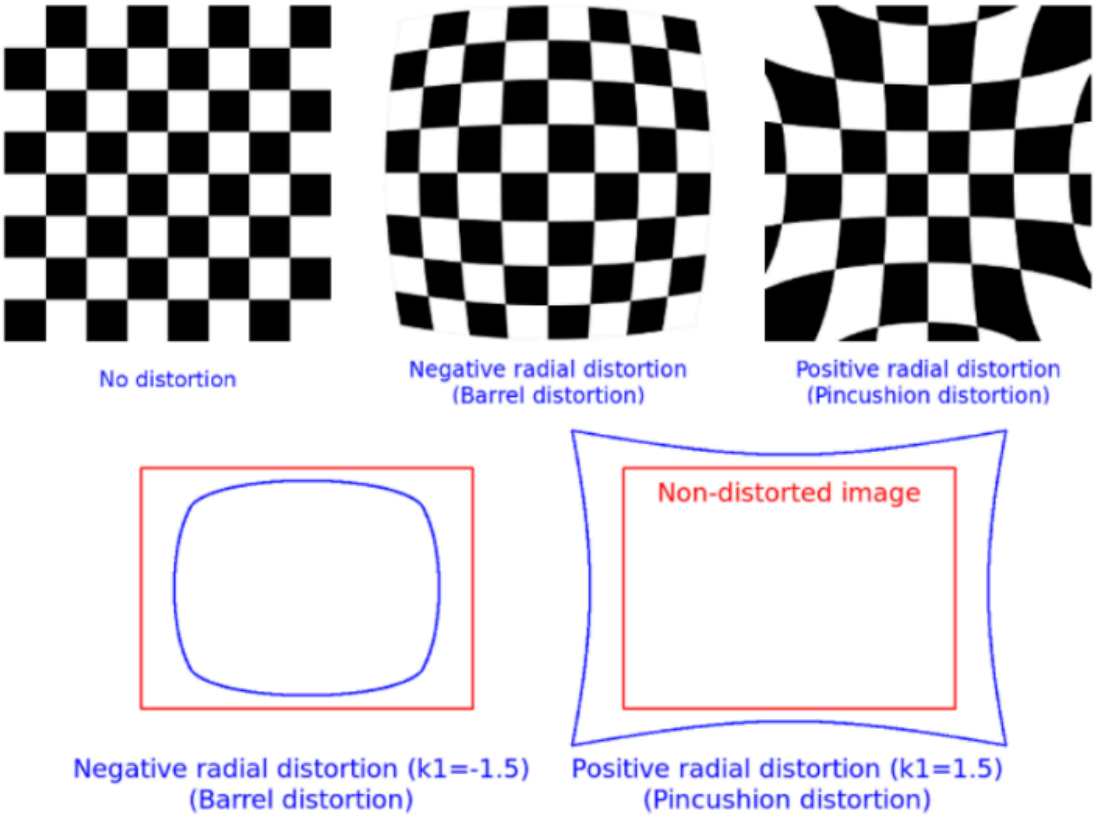
What does this distortion provide? That the assumption of linear projection (straight lines remain straight) is violated.

Let $x = x_c/z_c$, $y = y_c/z_c$ and $r^2 = x^2 + y^2$. The distorted point is obtained as:

$$\mathbf{x}' = \underbrace{(1 + \kappa_1 r^2 + \kappa_2 r^4)}_{\text{Radial Distortion}} \begin{pmatrix} x \\ y \end{pmatrix} + \underbrace{\begin{pmatrix} 2\kappa_3 x y + \kappa_4(r^2 + 2x^2) \\ 2\kappa_4 x y + \kappa_3(r^2 + 2y^2) \end{pmatrix}}_{\text{Tangential Distortion}}$$

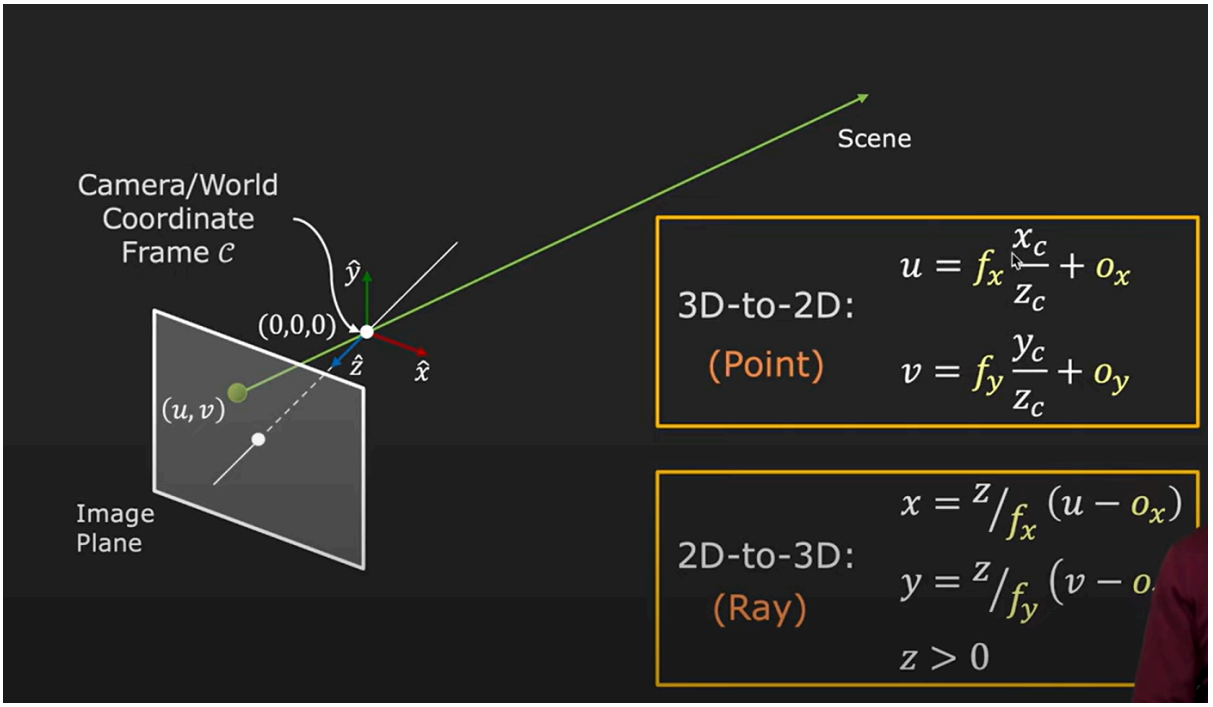
$$\mathbf{x}_s = \begin{pmatrix} f_x x' + c_x \\ f_y y' + c_y \end{pmatrix}$$

We undistort the image such that the perspective projection model applies. This above is a possible distortion model but others can be used for wide-angle lenses such as fisheye.

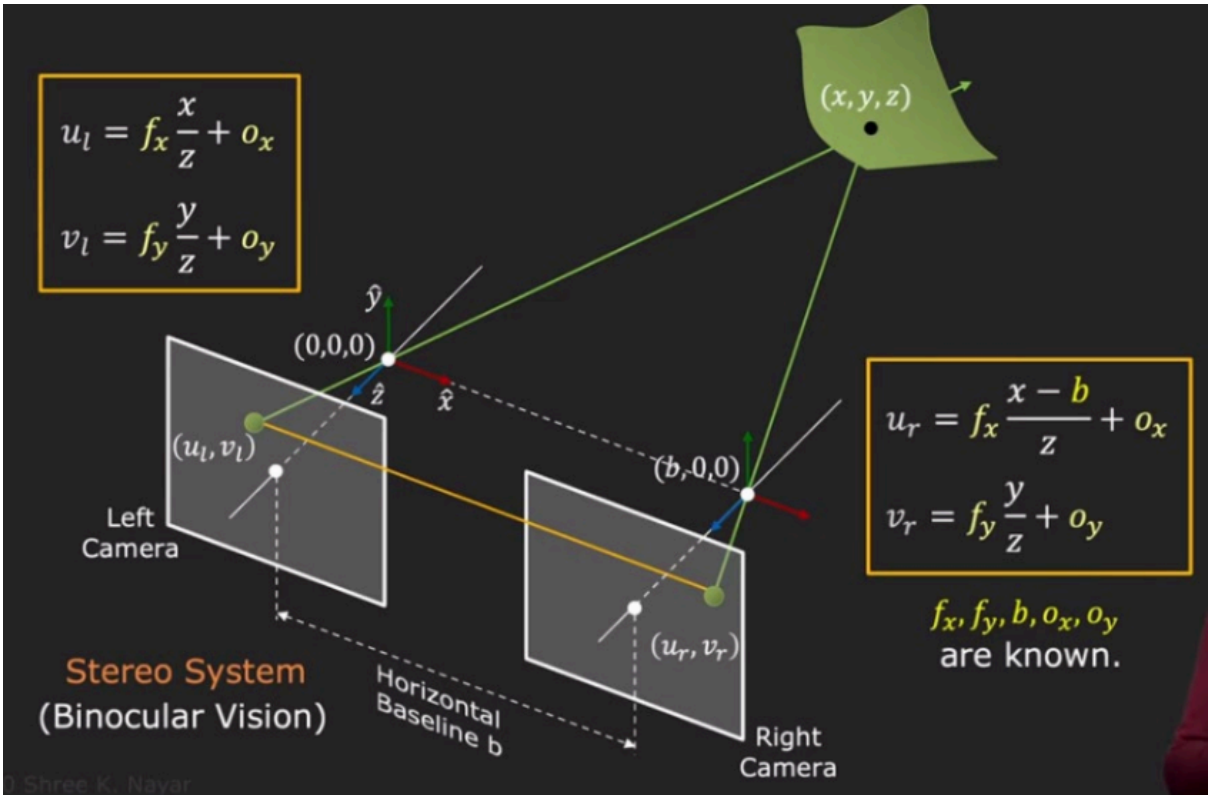


Application, Simple Stereo Vision (Triangulation using two camera)

Given a calibrated camera, and a point (u,v) in the image plane, can we find the 3D scene point from a single 2D image point?



With a single camera, we can only back project a pixel to obtain a ray on which the actual 3D point lies, but we can't solve the problem[22]. We need two cameras



This stereo system uses two cameras. We know the intrinsic parameters of the cameras. Now, we have:

[22] In fact, we should be able to invert the P matrix, that is a 3x4 matrix, in order to obtain the transformation from (u,v) to (x_w, y_w, z_w)

$$(u_l, v_l) = \left(f_x \frac{x}{z} + o_x, f_y \frac{y}{z} + o_y\right) \qquad (u_r, v_r) = \left(f_x \frac{x-b}{z} + o_x, f_y \frac{y}{z} + o_y\right)$$

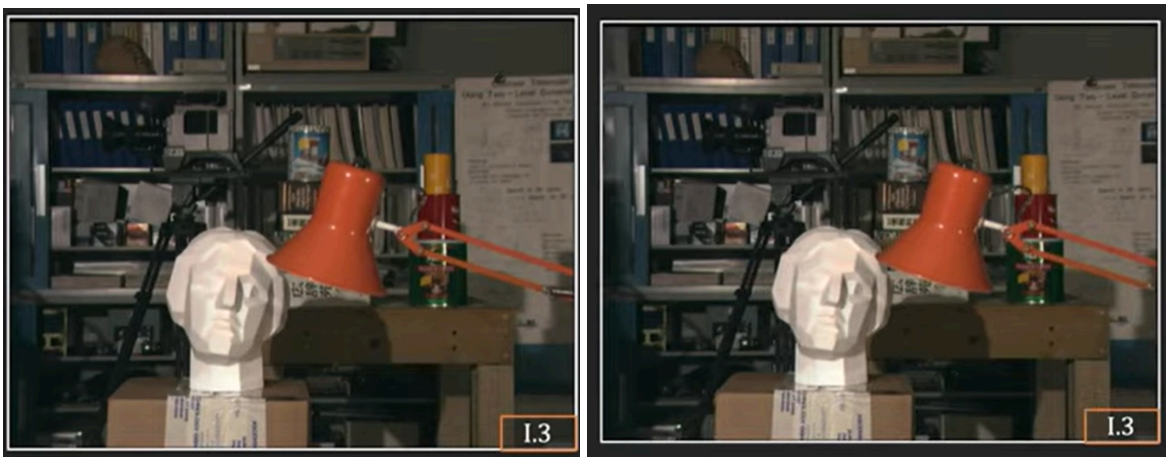
And solving for (x, y, z) we got

$$x = \frac{b(u_l - o_x)}{(u_l - u_r)} \qquad y = \frac{bf_x(v_l - o_y)}{f_y(u_l - u_r)} \qquad z = \frac{bf_x}{(u_l - u_r)}$$

Where (u_l - u_r) is the Disparity. As you can see the depth z of the object in the camera coordinates is inversely proportional to the Disparity. In fact if a point is really close to the two cameras, the disparity is going to be large, namely the two images will be very different. On the other end, at the infinity, the disparity is going to go to zero, namely the two images will be the same at infinity, regardless of the baseline b. In addition, the disparity is proportional to the Horizontal Baseline b[23], meaning that if the baseline is very small, the distances between the two images is going to be small, intuitively.

How to find disparities to compute depth

We have the two images



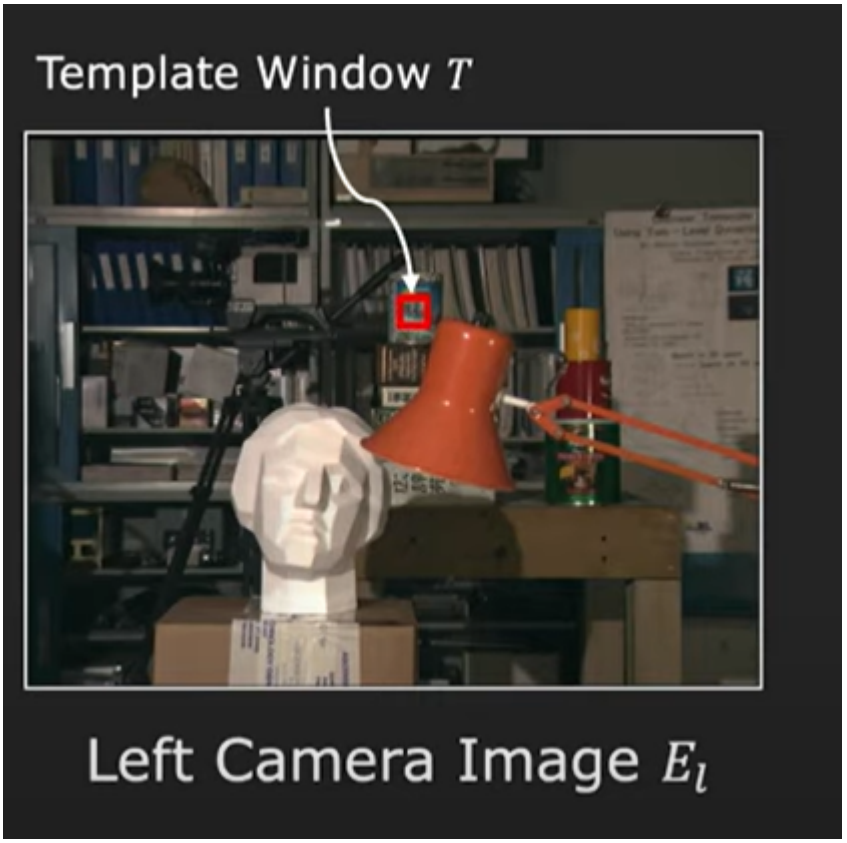
We want to find the disparity between left and right

[23] In fact $u_l - u_r = - f_x * b/z$

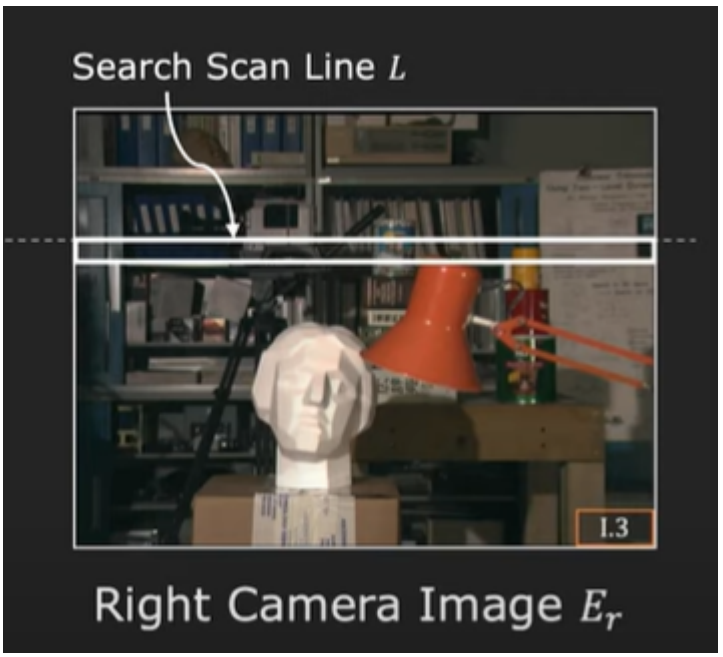
This above is the ground truth, let’s do some attempts. We know that there is no disparity in the vertical direction

$$v_l = v_r = f_y \frac{y}{z} + o_y$$

How to compute the horizontal disparity? A possible method is the Window Based Method: we take a window in the image to one image (the left)



we know the corresponding point must lie in the same horizontal scanline, since there is no vertical disparity, so we just need to scan horizontally



Matching the window in the scan line we will find the match, that is the corresponding point in the right camera image. Now that we have found the point u_r that matches with u_l, we can compute the disparity and so the depth of the object.

Similarity Metrics for Template Matching

To do template matching there are many different metrics we can use to get how similar two windows are. A possibility is to minimize the Sum of Absolute Differences

$$SAD(k,l) = \sum_{(i,j)\in T} |E_l(i,j) - E_r(i+k, j+l)|$$

Or we can minimize Sum of Squared Differences

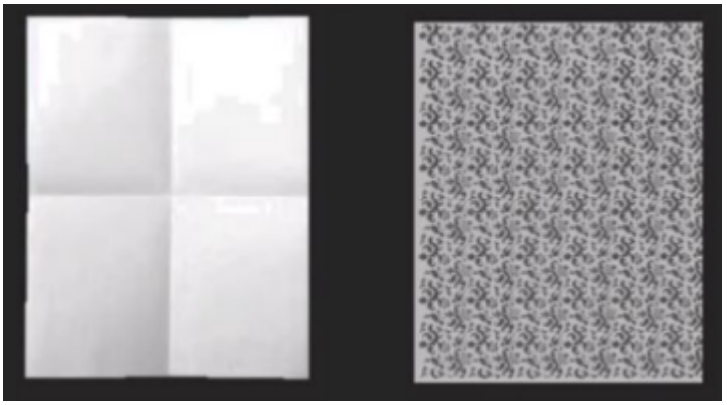
$$SSD(k,l) = \sum_{(i,j)\in T} |E_l(i,j) - E_r(i+k, j+l)|^2$$

Or maximize Normalized Cross-Correlation

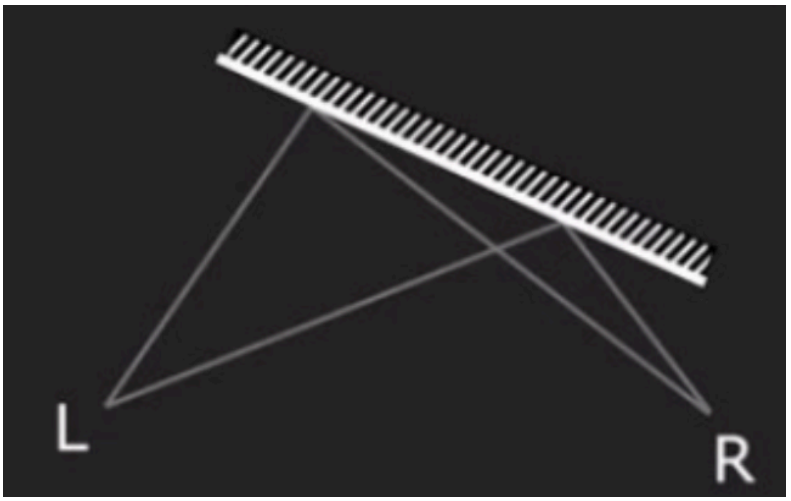
$$NCC(k,l) = \frac{\sum_{(i,j)\in T} E_l(i,j)\, E_r(i+k, j+l)}{\sqrt{\sum_{(i,j)\in T} E_l(i,j)^2 \sum_{(i,j)\in T} E_r(i+k, j+l)^2}}$$

Issues with Stereo Matching

1. The surface must have texture, so the one on the left is okay as long as you have the cross, otherwise having a flat region leads to a lot of matches and so the impossibility of finding a good match. Moreover, the surface must have non-repetitive texture, so the one on the right is not okay.



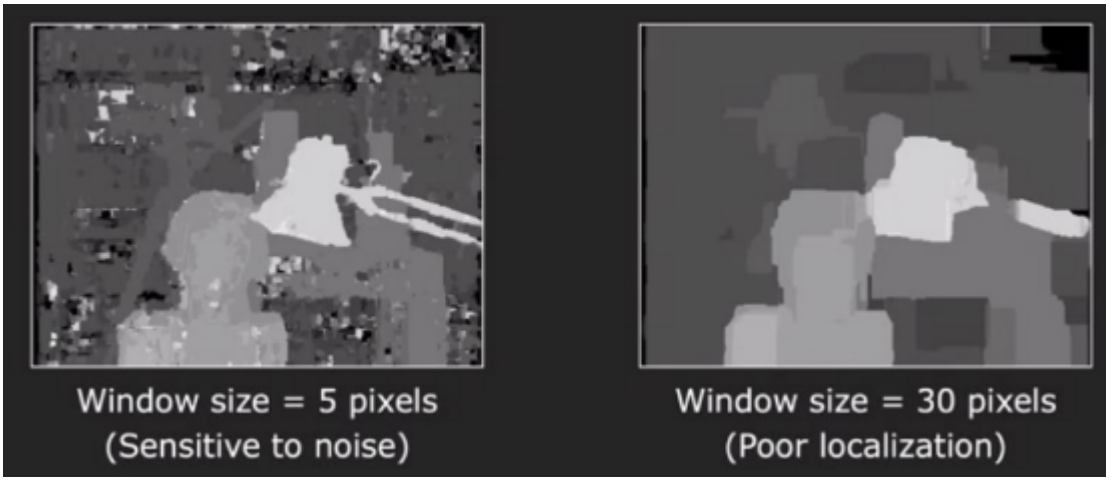
2. Foreshortening effects makes matching challenging



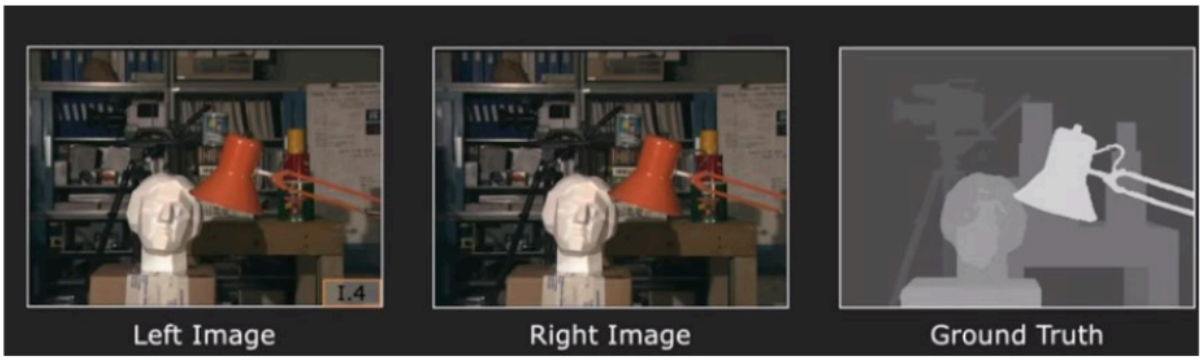
we need a window size, we can't use just 1 pixel to match. Once you take a window that implies that we are looking at a certain area in the scene. This area, projected onto the left image is different from the projected area onto the right image, and therefore we are not matching the same brightness patterns but a warped, distorted version of each other. So we can incorporate warping techniques that make the matching process more robust

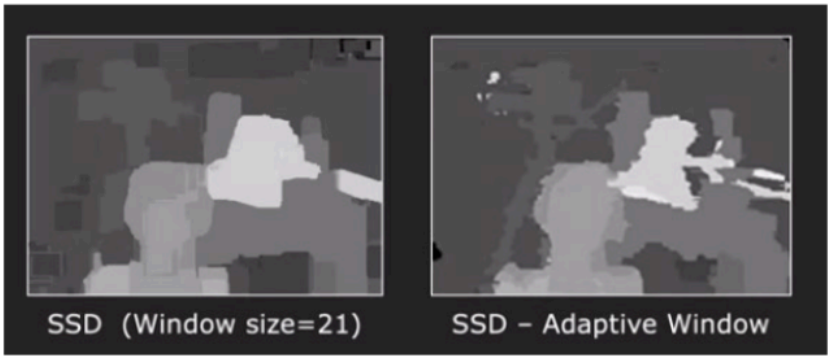
Size of the Window for Matching - Adaptive Window Method

When the window size is too small we are going to get good localization but high sensitivity to noise, as in edge detection: the smaller the window, the less descriptive the pattern is. On the other hand, when the window is large the matches will be more robust in terms of depth value but the disparity map will be more blurred especially at boundaries, i.e. poor localization, as in edge detectors, where with large windows you are not able to localize the edge quite well.

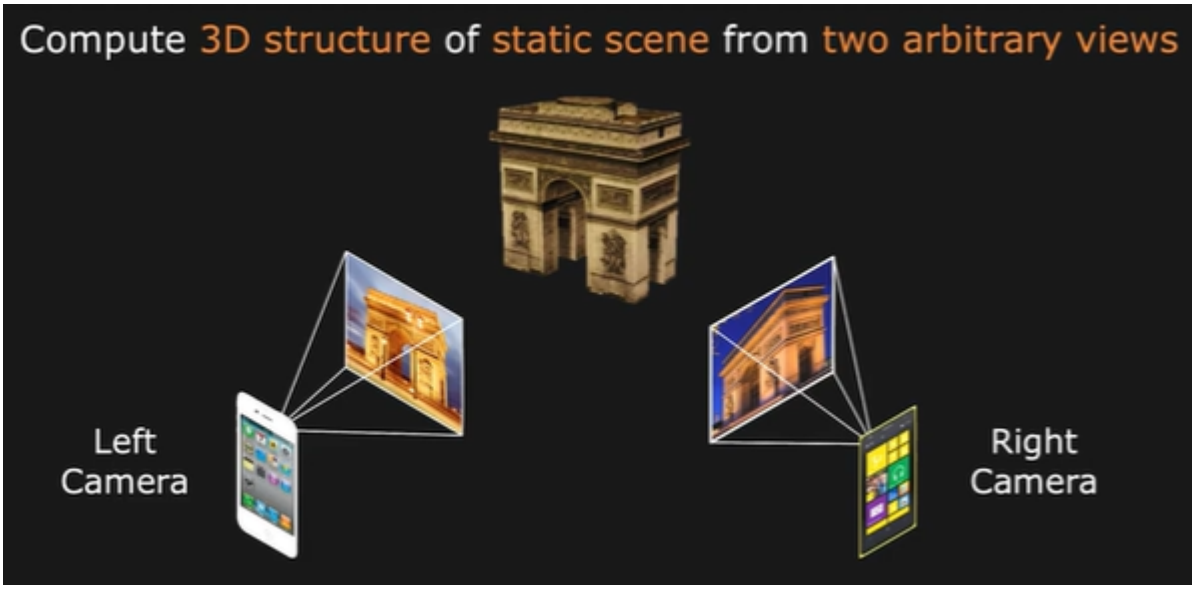


To find the tradeoff we use the Adaptive Window Method: for each point, match using windows of multiple sizes, choose then the window size that gives the best match in terms of similarity function chosen. Then uses this window size to compute the disparity. Note that the metric must be normalized using the number of pixels within the window.





Problem of Uncalibrated Stereo



We assume that we know the internal parameters of the two cameras[24]. We want to calibrate the uncalibrated stereo system, namely compute the extrinsic parameters, so the relative position and orientation of one camera w.r.t. the other.

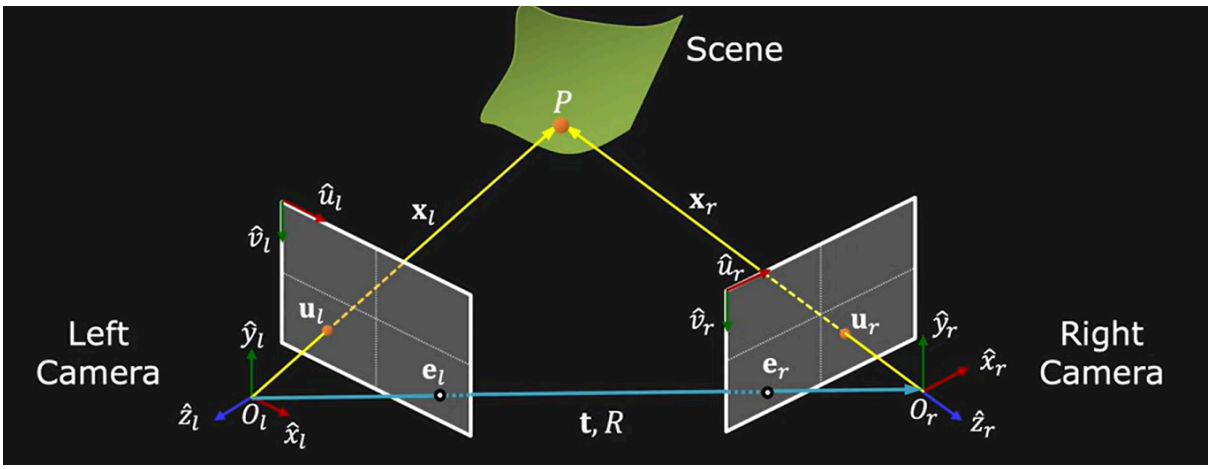
[24] This is not unrealistic: in the image itself there are metatag that includes this informations

How to Calibrate It

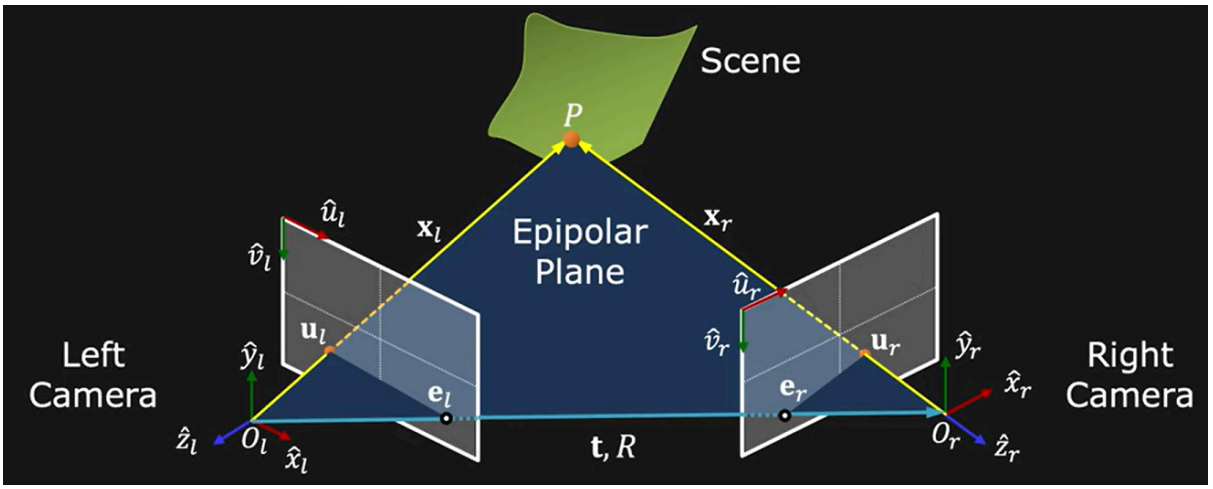
1. Assume K is known for each camera
2. First, we want to find a few reliable corresponding points. To find a set of initial correspondences (at least 8) we can use SIFT, finding robust matches.
3. Find relative camera position t and orientation R

Essential Matrix

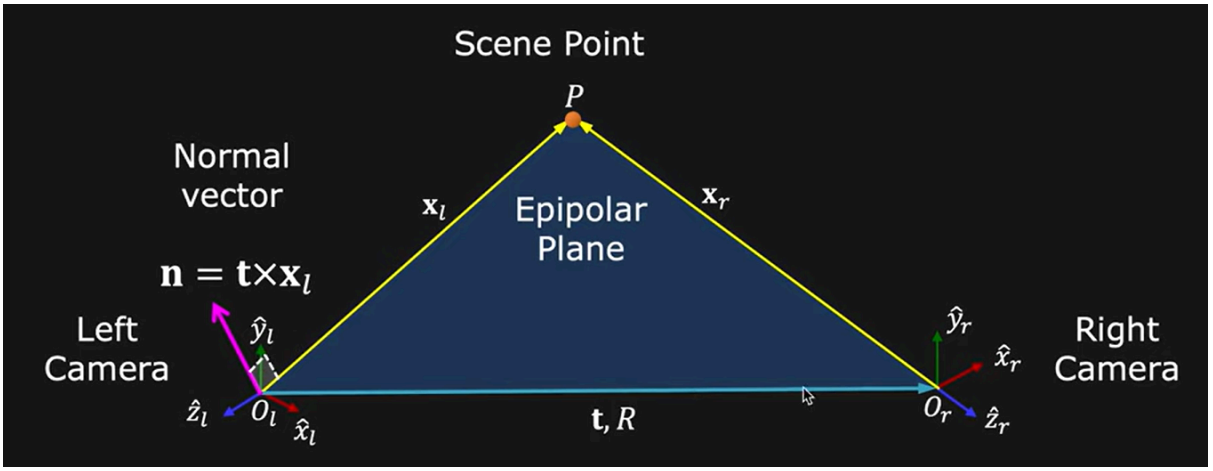
Our goal is to find the relative position and orientation of one camera w.r.t. to the other in the uncalibrated stereo system



We want to compute the translation t and rotation R. The projection of the center of the right camera onto the left camera images and vice versa are the epipoles of the stereo system. Any stereo system has a unique pair of epipoles. Each point in the scene has a unique epipolar plane



We want to set an epipolar constraint with this plane. There is a normal to the plane, so orthonormal to the backproject of the 2D point to a ray in 3D (x_l) and the translation t.



The dot product of x_l and the normal must be zero, since they are orthogonal

$$\mathbf{x}_l \cdot (\mathbf{t}\times\mathbf{x}_l) = 0$$

$$[x_l \quad y_l \quad z_l] \begin{bmatrix} t_y z_l - t_z y_l \\ t_z x_l - t_x z_l \\ t_x y_l - t_y x_l \end{bmatrix} = 0$$

Cross-product definition

$$[x_l \quad y_l \quad z_l] \underbrace{\begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}}_{T_\times} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = 0$$

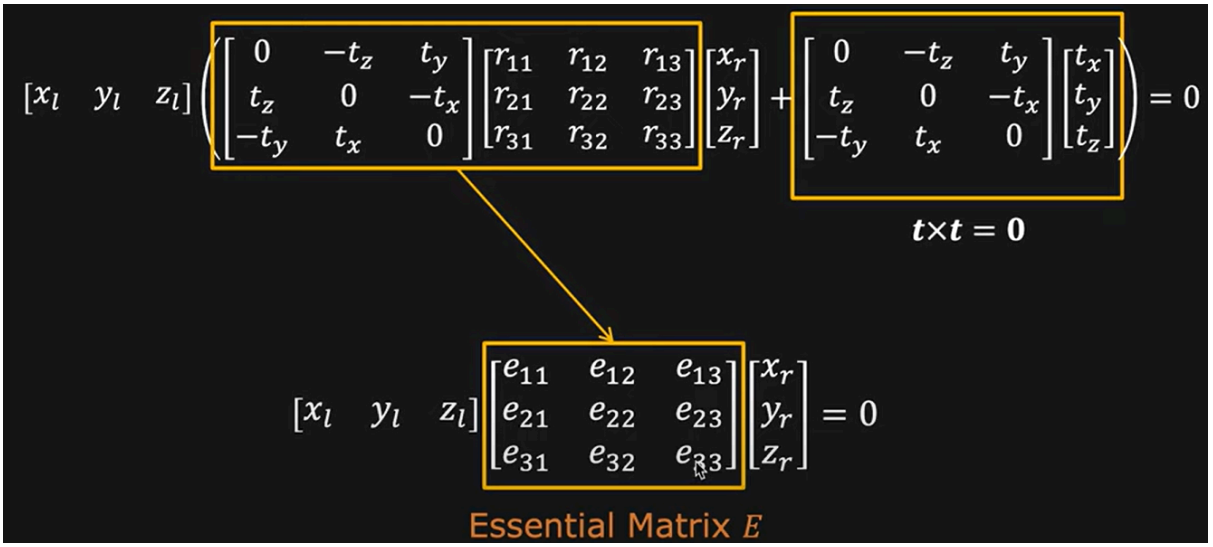
Matrix-vector form

With skew-symmetric translation matrix T_x. Now we can write the transformation from the right camera to the left:

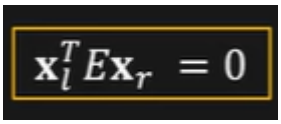
$$\mathbf{x}_l = R\mathbf{x}_r + \mathbf{t}$$

$$\begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

substituting $[x_l, y_l, z_l]^T$ in the epipolar constraint in Matrix-vector form we get :

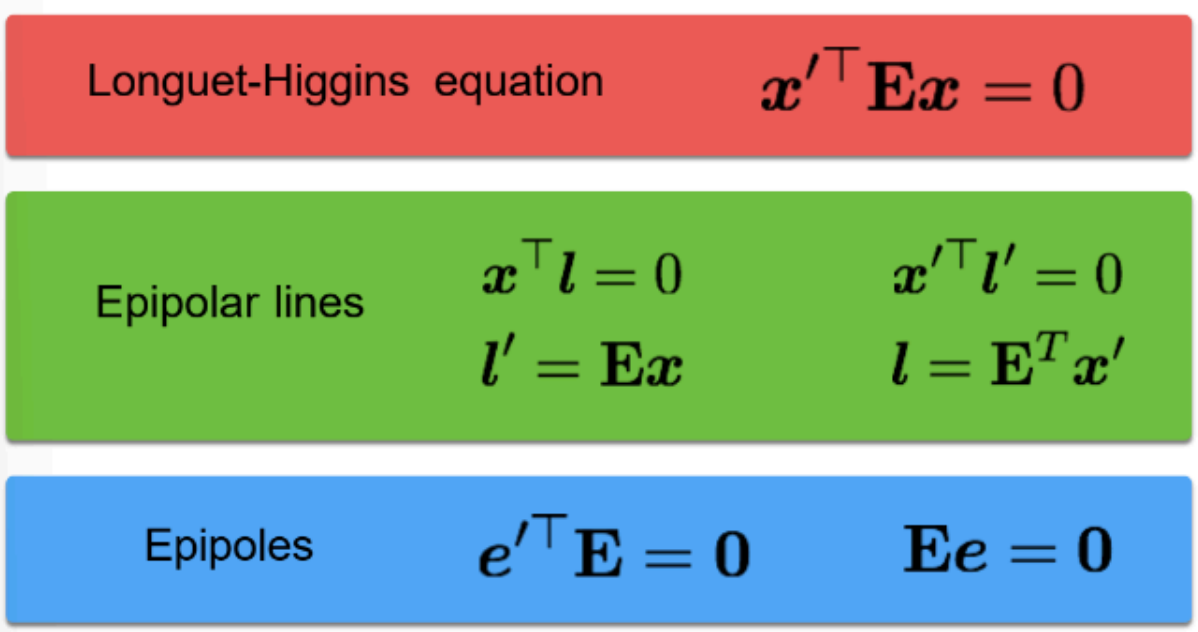


Deriving the epipolar constraint in the form



The essential matrix[25] is the product of the translation matrix and the rotation matrix. E can be decoupled in the translation and rotation matrix: in fact, T_x is skew-symmetric ($a_{ij} = -a_{ji}$), while R is orthonormal, so we can decouple E using Singular Value Decomposition.

Properties of the Essential Matrix

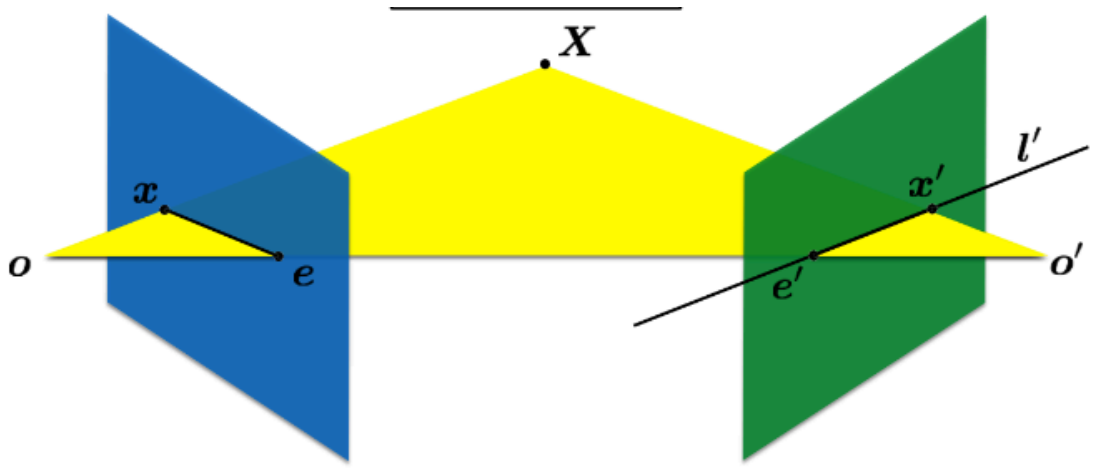


As you can see. the essential matrix tells us, given a point in one image, the epipolar line in the second view.

[25] The difference between the essential matrix and the homography is that the essential matrix maps a point to a line, while the homography maps a point to a point

Other properties:

- E has 5 DoF, since R and T have 3 DoF but one can apply an arbitrary scale to D since is a projective transformation
- rank(E) = 2
- E has 2 singular value both of which are equal



Fundamental Matrix

How can we compute the essential matrix? We have a problem with the new form of the epipolar constraint: x_l and x_r are unknown, since we don't know the location of the point in 3D, that's our goal. What we do have is u and v for both left and right camera

$$u_l = f_x^{(l)} \frac{x_l}{z_l} + o_x^{(l)} \qquad v_l = f_y^{(l)} \frac{y_l}{z_l} + o_y^{(l)}$$

With a bit of manipulation

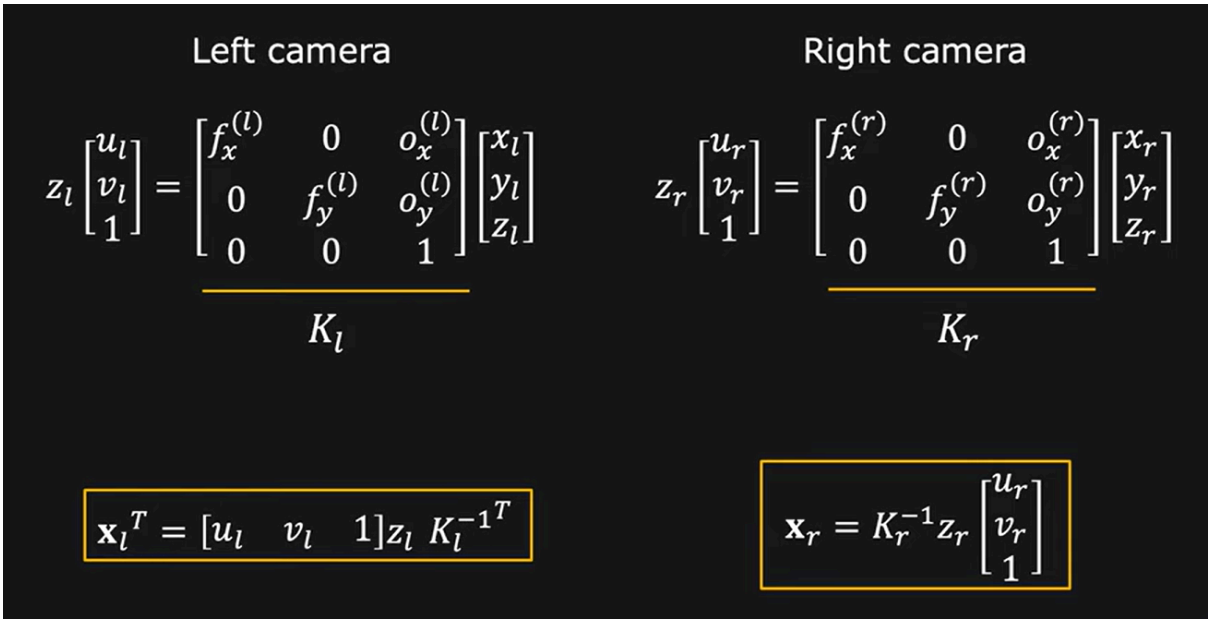
$$z_l u_l = f_x^{(l)} x_l + z_l o_x^{(l)} \qquad z_l v_l = f_y^{(l)} y_l + z_l o_y^{(l)}$$

Representing in matrix form:

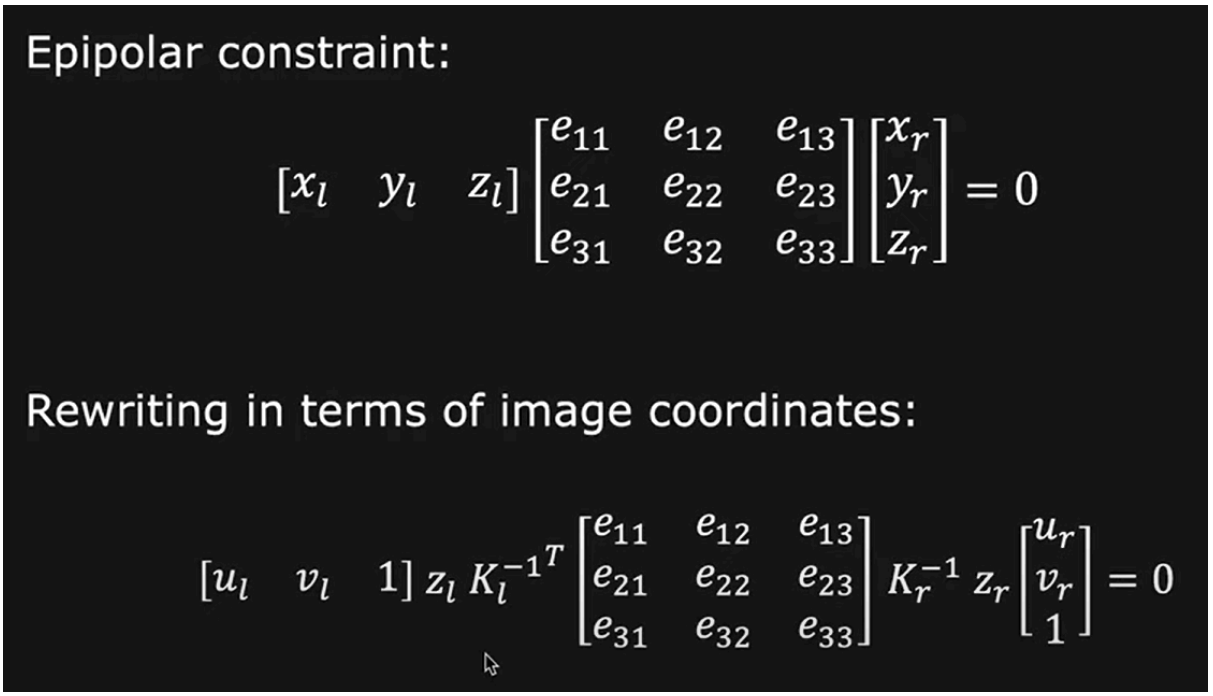
$$z_l \begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} z_l u_l \\ z_l v_l \\ z_l \end{bmatrix} = \begin{bmatrix} f_x^{(l)} x_l + z_l o_x^{(l)} \\ f_y^{(l)} y_l + z_l o_y^{(l)} \\ z_l \end{bmatrix} = \begin{bmatrix} f_x^{(l)} & 0 & o_x^{(l)} \\ 0 & f_y^{(l)} & o_y^{(l)} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix}$$

Known Camera Matrix K_l

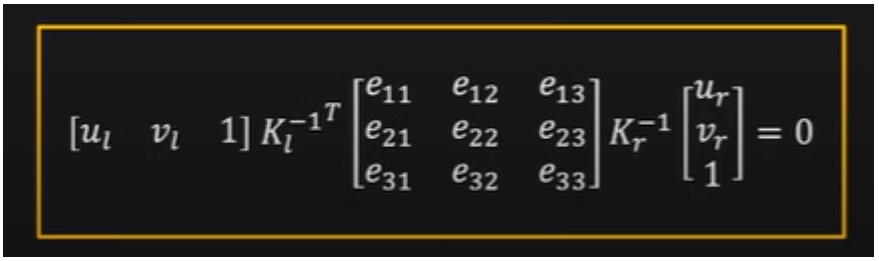
we can do that for both the cameras



having isolated x_l^T and x_r. Then we can rewrite the epipolar constraint with this new knowledge



We still have z_l and z_r that are unknown. We can say that z_l and z_r != 0, since the object can't be in the center of the projection. Since we have that the entire equation is equal to zero, and both z_l and z_r are not zero, we can remove them



Multiplying the three internal matrices, we get the Fundamental Matrix F

$$\begin{bmatrix} u_l & v_l & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0$$

Fundamental Matrix F

The fundamental matrix is a generalization of the essential matrix, where the assumption of identity intrinsic matrices is removed. The unknowns of the expression are in F. Solving this equation, we can find E, since from the inverse formula we get

$$E = K_l^T F K_r$$

And once we have the essential matrix E, using singular value decomposition, we can find the translation and the rotation for going from one camera to the other.

Properties of the Fundamental Matrix

The same as E but in the image coordinates, and not in the camera coordinates

Longuet-Higgins equation	$x'^{\top} \mathbb{E} x = 0$	
Epipolar lines	$x^{\top} l = 0$	$x'^{\top} l' = 0$
	$l' = \mathbb{E} x$	$l = \mathbb{E}^T x'$
Epipoles	$e'^{\top} \mathbb{E} = 0$	$\mathbb{E} e = 0$

So, while the E matrix maps lines to epipolar lines, F maps pixels to epipolar lines.

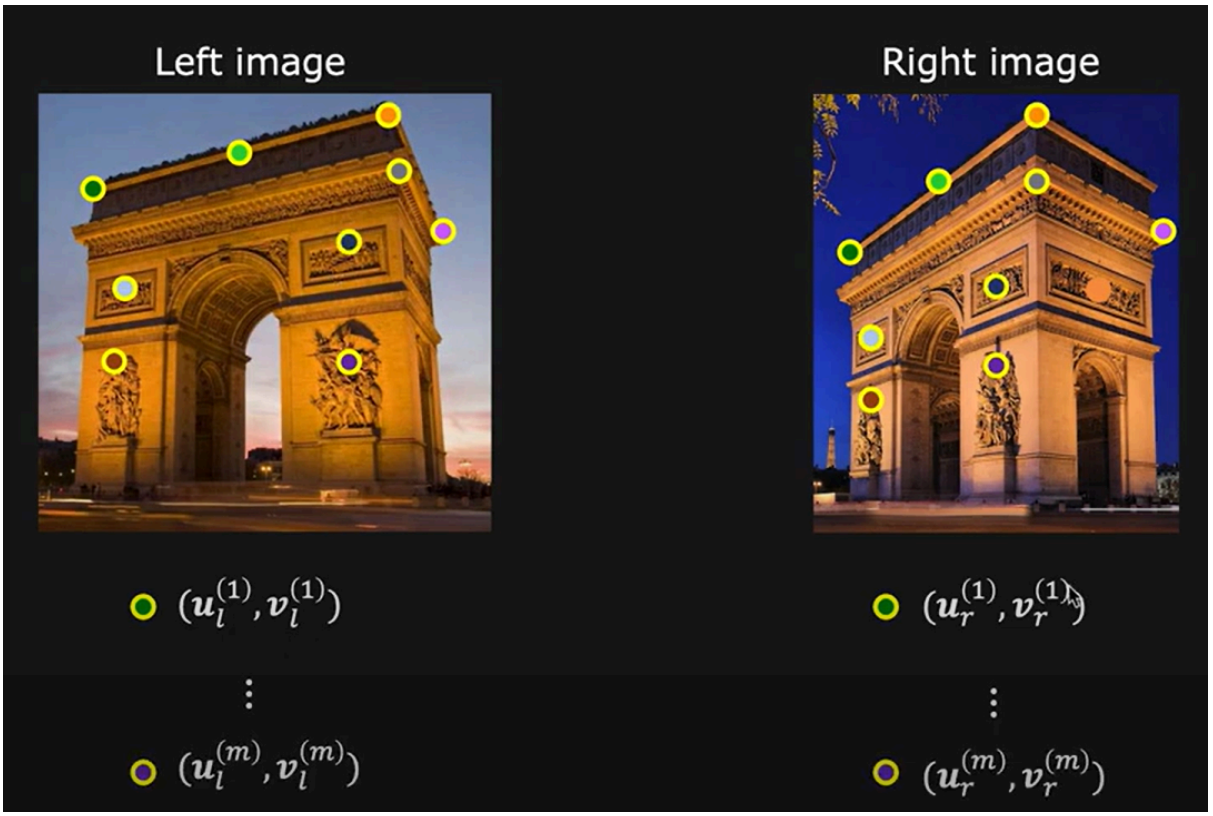
Other properties:

- F has 7 DoF
- rank(F) = 2

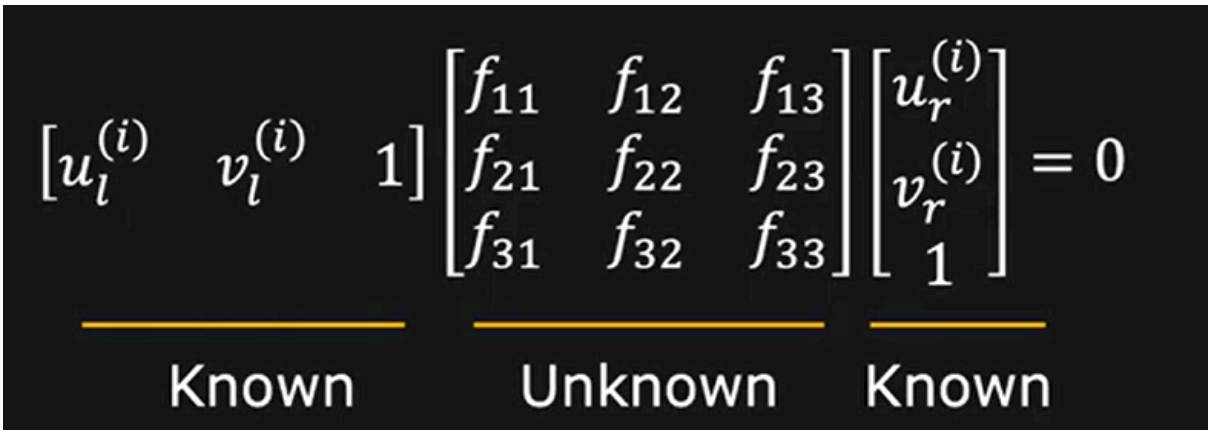
Estimating the Fundamental Matrix - 8 point algorithm

SIFT for initial correspondence

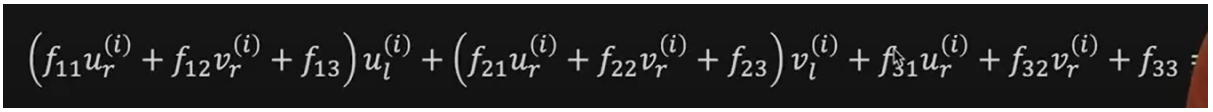
We need to find F to calibrate our system. First thing we need to find a set of corresponding features with SIFT, at least 8.



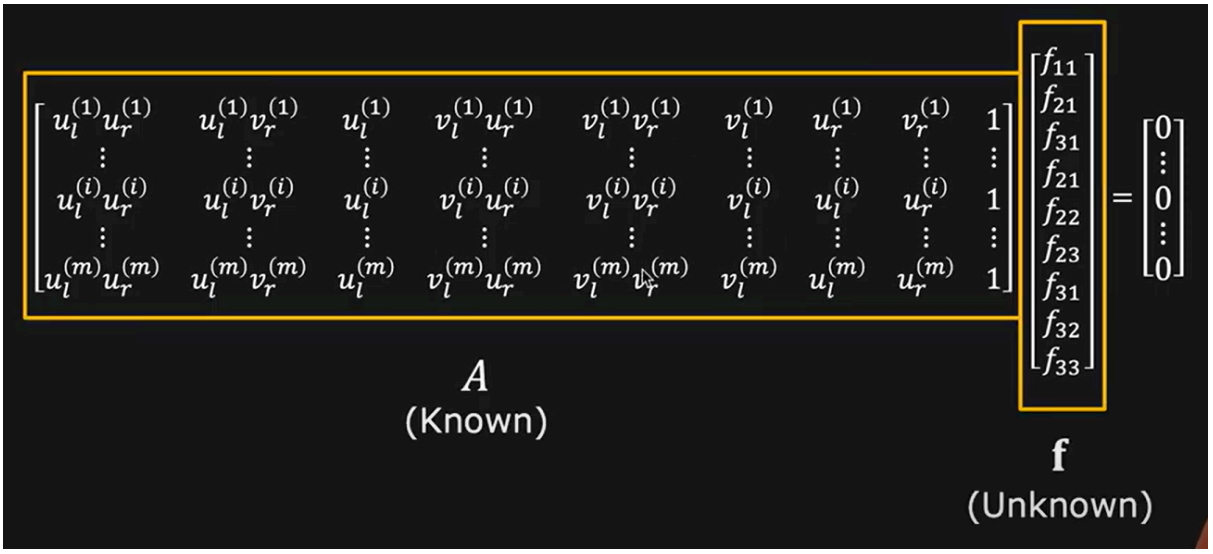
We are going to plug those correspondences in our epipolar constraint.



Expanding the matrix, to get al linear equation



we get one of these linear equation for each point found by SIFT, so we can stack them up to get the following linear system



The Tale of Missing Scale - Least Square Solution

The fundamental matrix acts on homogeneous coordinates, meaning that the following holds:

$$[u_l \quad v_l \quad 1]\begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}\begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0 = [u_l \quad v_l \quad 1]\begin{bmatrix} kf_{11} & kf_{12} & kf_{13} \\ kf_{21} & kf_{22} & kf_{23} \\ kf_{31} & kf_{32} & kf_{33} \end{bmatrix}\begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix}$$

So F and kF describe the same epipolar geometry, namely if you take a stereo pair and you take two images, and you scale up the entire system, both the world and the stereo system, we will get the same images. So we can set the scale of F, and we choose to set the scale of F such that

$$\|\mathbf{f}\|^2 = 1$$

Now we can solve Af = 0, finding a least square solution

We want $A\mathbf{f}$ as close to 0 as possible and $\|\mathbf{f}\|^2 = 1$:

$$\min_{\mathbf{f}} \|A\mathbf{f}\|^2 \quad \text{such that } \|\mathbf{f}\|^2 = 1$$

Constrained linear least squares problem

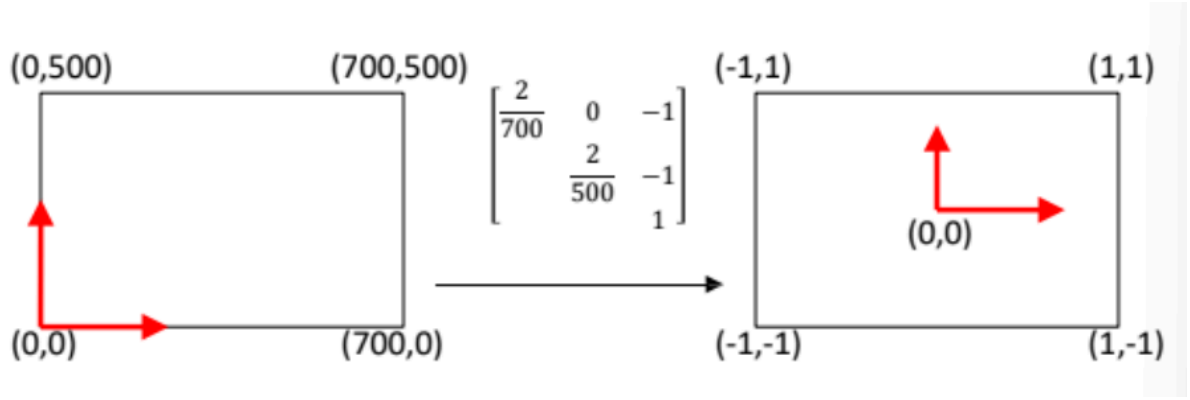
Normalization

$$\begin{bmatrix} u_1u_1' & v_1u_1' & u_1' & u_1v_1' & v_1v_1' & v_1' & u_1 & v_1 & 1 \\ u_2u_2' & v_2u_2' & u_2' & u_2v_2' & v_2v_2' & v_2' & u_2 & v_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_nu_n' & v_nu_n' & u_n' & u_nv_n' & v_nv_n' & v_n' & u_n & v_n & 1 \end{bmatrix}$$

~10000 ~10000 ~100 ~10000 ~10000 ~100 ~100 ~100 1

Orders of magnitude difference
between column of data matrix
–→ least-squares yields poor results

We can normalize the least squares transforming image in the coordinates [-1, 1]



The normalized 8-point algorithm is:

- Transform the input in the normalized space [-1, 1]
- Call 8-point on the normalized input in order to obtain $F_{normalized}$
- Transform $F_{normalized}$ to obtain F in the original space

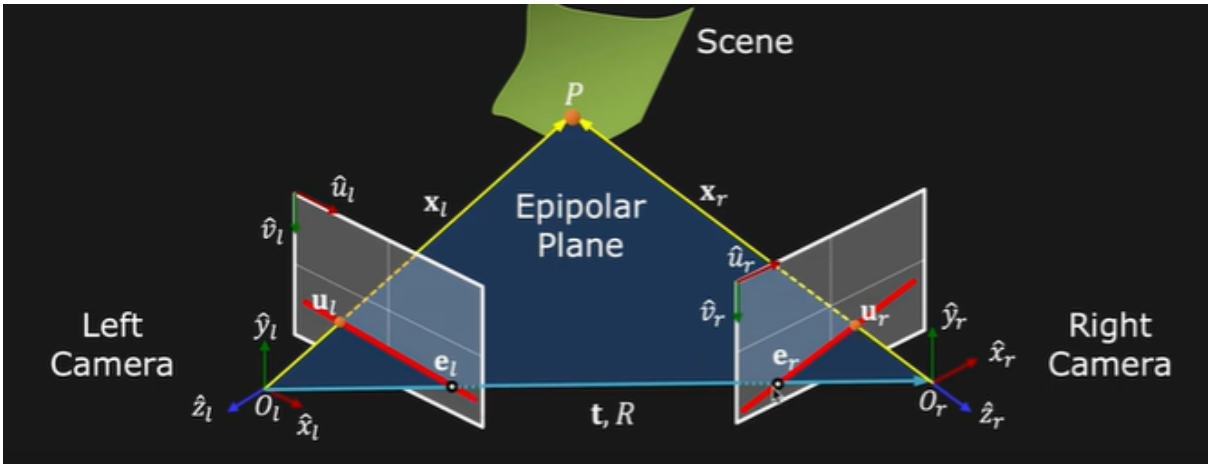
Final Algorithm Steps

1. Normalize input points
2. Construct $A_{M \times 9}$ with M = 8 at least
3. Find the SVD of A in order to find a solution for F
4. Entries of F are the elements of column of V corresponding to the least singular value
5. Enforce rank-2 constraint on F: the F obtained with the SVD of A might not be rank-2 due to noise and inaccuracies. For enforcing the rank-2 constraint we perform another SVD on F.
6. Un-normalize F

Find Dense Correspondence

Now that we have F, so E and so T and R. We can say that our system is now calibrated. The next step in the algorithm is to find dense correspondences, so that at the end we have a dense map of the 3D scene.

As in the simple stereo system, we have that the stereo matching problem is still a 1D search, but along which line? The epipolar plane of a point intersects with the two image planes, creating two lines



So for each point we have two corresponding epipolar lines. All the epipolar lines of all the points go through the epipoles. A point in one of the two images, let's say the left camera images, turns out that the corresponding point in the right image must lie in the epipolar line in the right image. So we just need to search along this line, as in the simple stereo system.

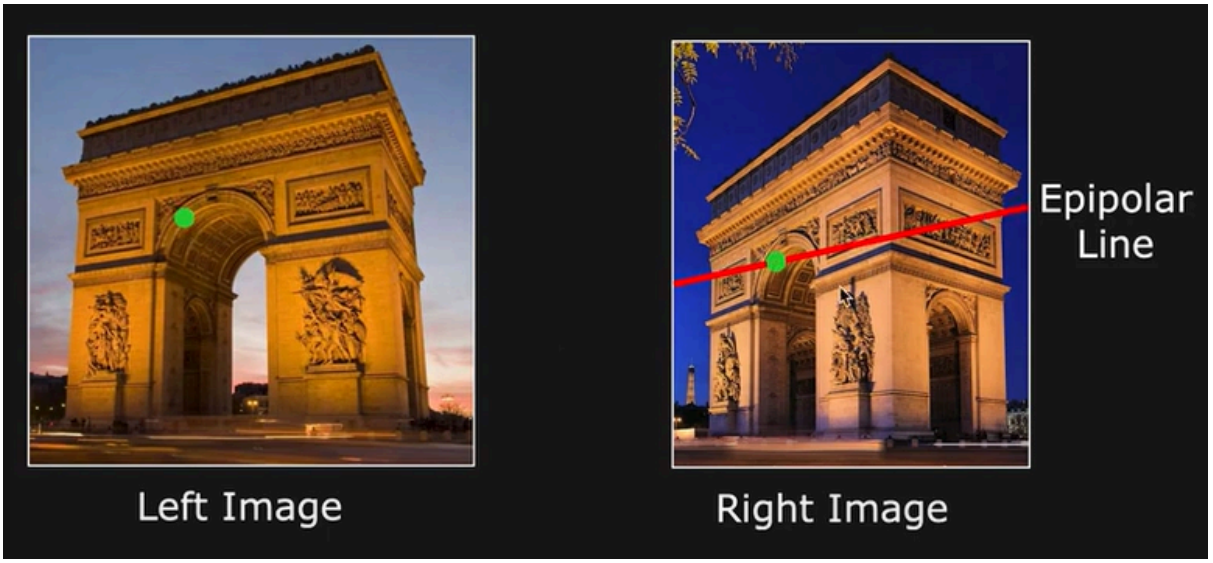
$$\begin{bmatrix} u_l & v_l & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0$$

We know u_l, v_l and all F, we want to find u and v in the right image

$$(f_{11}u_l + f_{21}v_l + f_{31})u_r + (f_{12}u_l + f_{22}v_l + f_{32})v_r + (f_{13}u_l + f_{23}v_l + f_{33}) = 0$$

Equation for right epipolar line: $a_l u_r + b_l v_r + c_l = 0$

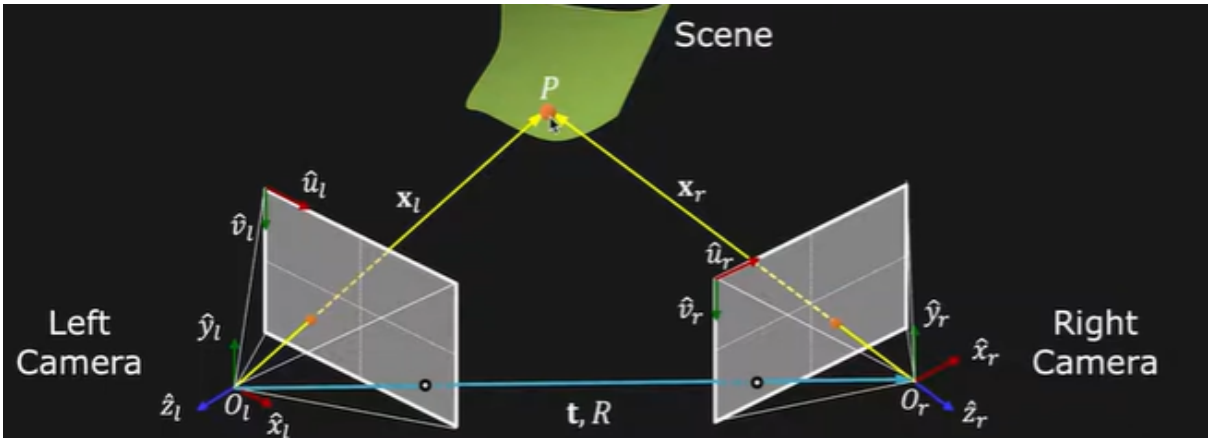
The same holds for finding the left (u,v) from the right one. Let's make this example:



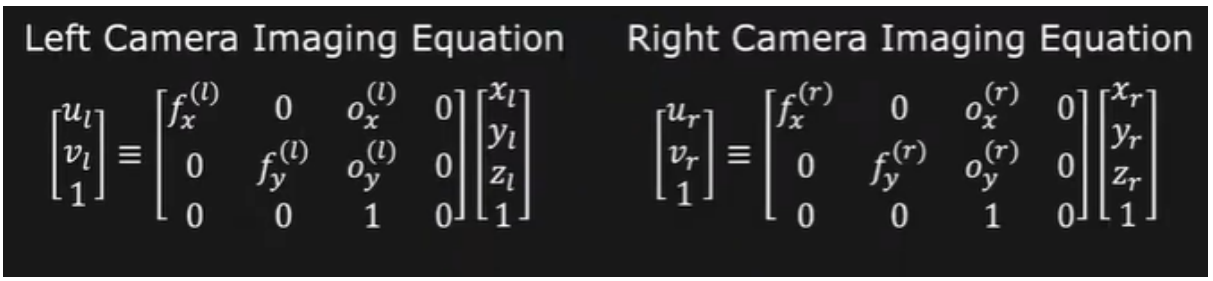
Now that we have a dense map, we can compute depth using triangulation

Compute Depth

We have correspondences between right and left images.



We know these two:



Moreover, we now know that

$$\begin{bmatrix} x_l \\ y_l \\ z_l \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}$$

We substitute in the imaging equations in order to have only the right coordinates

$$\begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} \equiv \begin{bmatrix} f_x^{(l)} & 0 & o_x^{(l)} & 0 \\ 0 & f_y^{(l)} & o_y^{(l)} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}$$

$$\tilde{\mathbf{u}}_l = P_l \, \tilde{\mathbf{x}}_r$$

Having multiplied M_{int} and M_{ext} to get the Projection Matrix.

$$\begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} \equiv \begin{bmatrix} f_x^{(r)} & 0 & o_x^{(r)} & 0 \\ 0 & f_y^{(r)} & o_y^{(r)} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}$$

$$\tilde{\mathbf{u}}_r = M_{int_r} \, \tilde{\mathbf{x}}_r$$

Now the only unknown is x_r, rearranging the terms we got

$$\begin{bmatrix} u_r m_{31} - m_{11} & u_r m_{32} - m_{12} & u_r m_{33} - m_{13} \\ v_r m_{31} - m_{21} & v_r m_{32} - m_{22} & v_r m_{33} - m_{23} \\ u_l p_{31} - p_{11} & u_l p_{32} - p_{12} & u_l p_{33} - p_{13} \\ v_l p_{31} - p_{21} & v_l p_{32} - p_{22} & v_l p_{33} - p_{23} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = \begin{bmatrix} m_{14} - m_{34} \\ m_{24} - m_{34} \\ p_{14} - p_{34} \\ p_{24} - p_{34} \end{bmatrix}$$

Using the pseudo-inverse, we can compute the least squares solution

$$A\mathbf{x}_r = \mathbf{b}$$

$$A^T A \mathbf{x}_r = A^T \mathbf{b}$$

$$\mathbf{x}_r = (A^T A)^{-1} A^T \mathbf{b}$$

Structure from Motion

From an uncontrolled video (motion of the camera not known) we can extract the three dimensional structure of the scene and figure out how the camera moved.

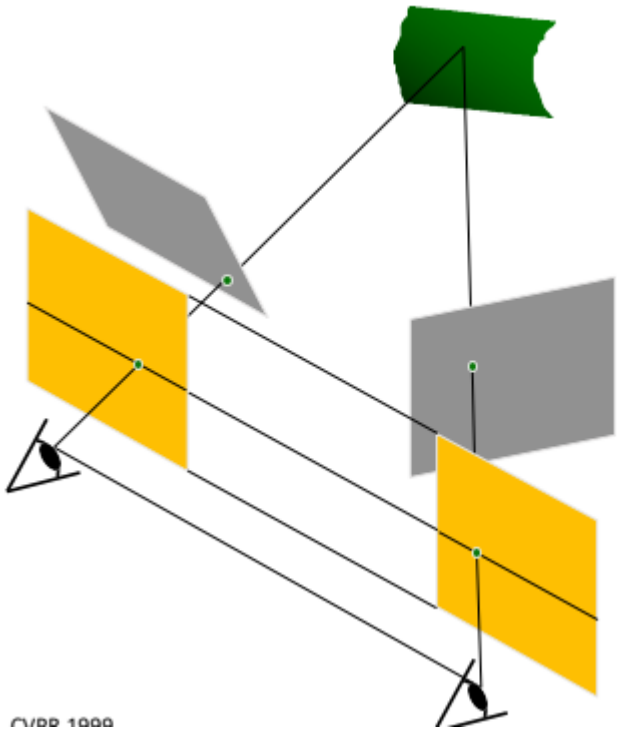
Stereo 3D reconstruction pipeline:

1. Detect feature points with SIFT
2. Calibrate cameras finding E or F
3. Rectify images with homographies in order to have epipolar lines horizontal
4. Find matching points along the epipolar lines: Stereo Matching. In this way we will have tracked the points in each image plane.
5. Perform Triangulation: estimate depth from disparity

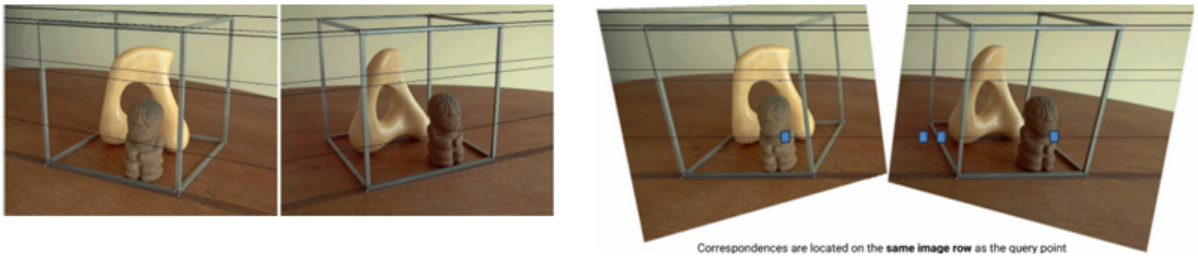
SIFT detection and calibration finding F, as well as triangulation, has been treated previously. Let's see rectification and stereo matching.

Rectification

If the image planes are not parallel we can find homographies to project each view onto a common plane parallel to the baseline.



After rectification, the image planes are parallel and the epipolar lines horizontal. The rectification algorithm includes calculating a rectifying rotation R_{rect} with which we warp pixels in both images. After rectification, correspondences are located on the same image row as the query point



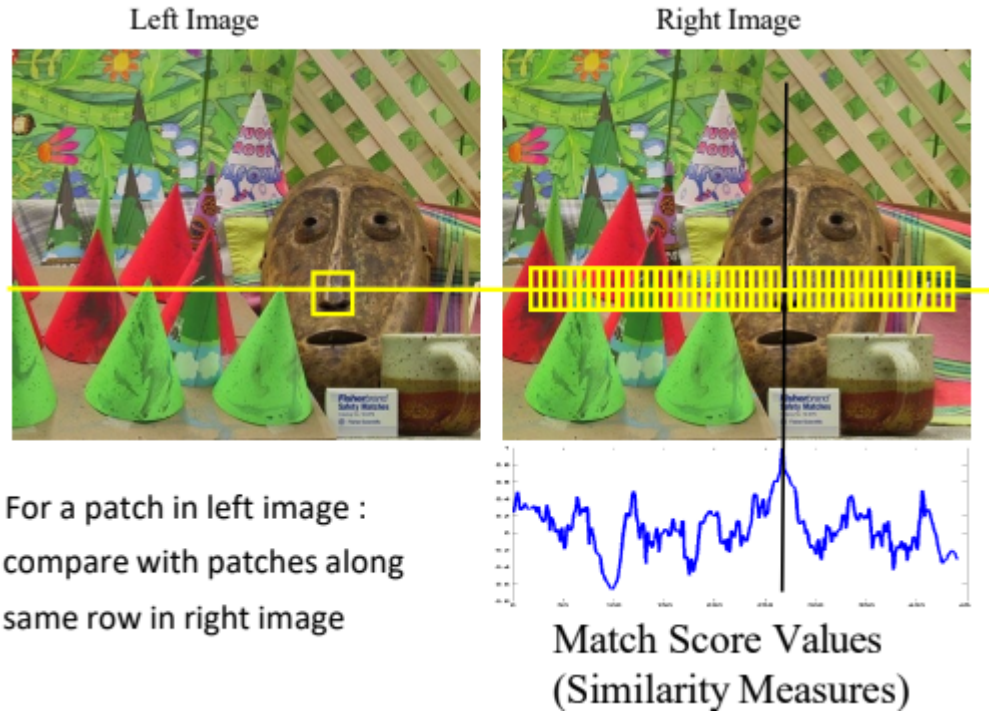
Stereo Matching

Finding correspondences between pixels in two or more images of the same scene taken from different viewpoints.

Local Stereo Matching Algorithm

Local Stereo Matching Algorithm: for a patch in the left image we compare it with patches along the same row in the right image and we compute the match score value (similarity measure). After that we select the patch with the highest match score. For similarity

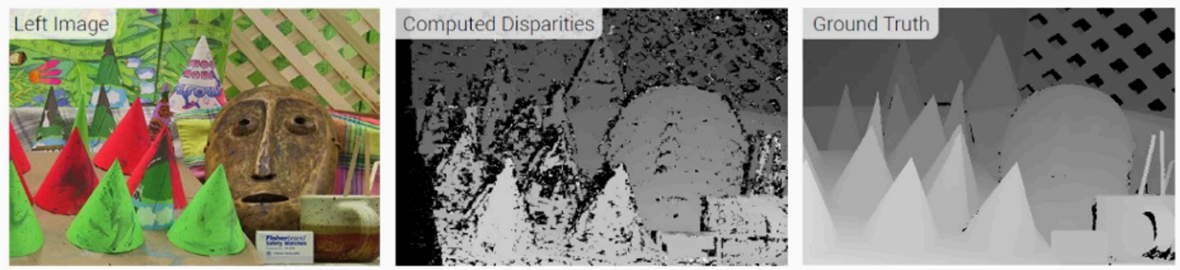
measures we can use NCC, SAD, SDD etc.



How large should the patch be? We can use the Adaptive window method to find the best size. Remember that smaller the more noise and details, the larger the poorer localization, less details but smoother.

Block Matching:

1. Start Block Matching: initiate the process
2. Partition Reference Image into Blocks: divide the reference image (e.g. the left one) into smaller blocks of a fixed size
3. Define Search windows in the other image: e.g. in the right image define corresponding search windows
4. Compute similarity
5. Select best match: highest similarity
6. Calculate disparity: determine the horizontal displacement
7. Repeat for all blocks
8. Generate disparity map: combine the disparity values to create a disparity map representing depth information for each block
9. Post-Processing

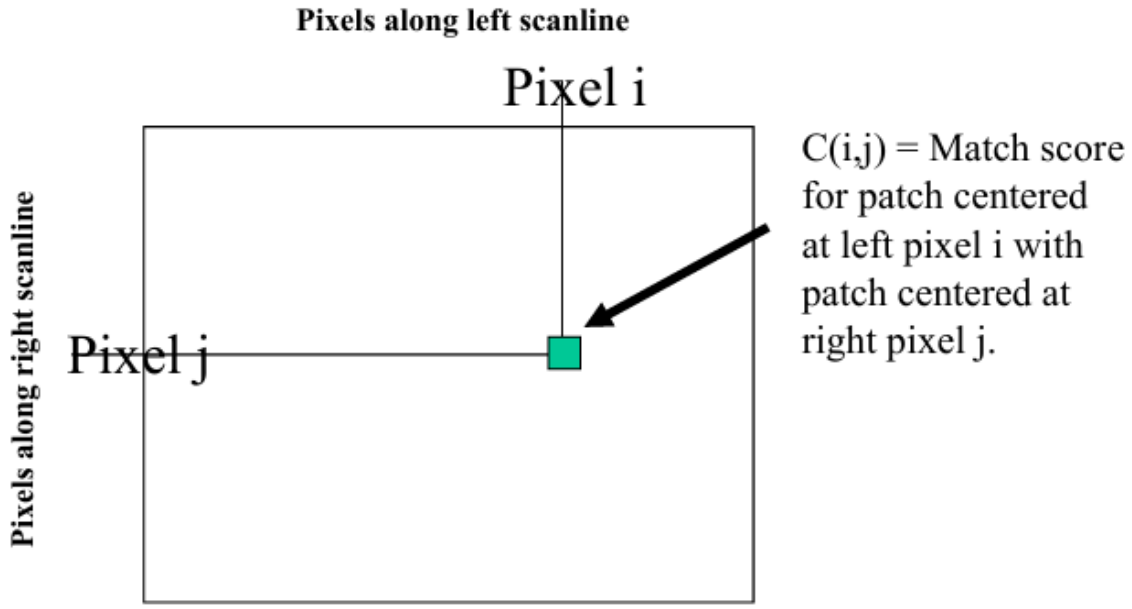


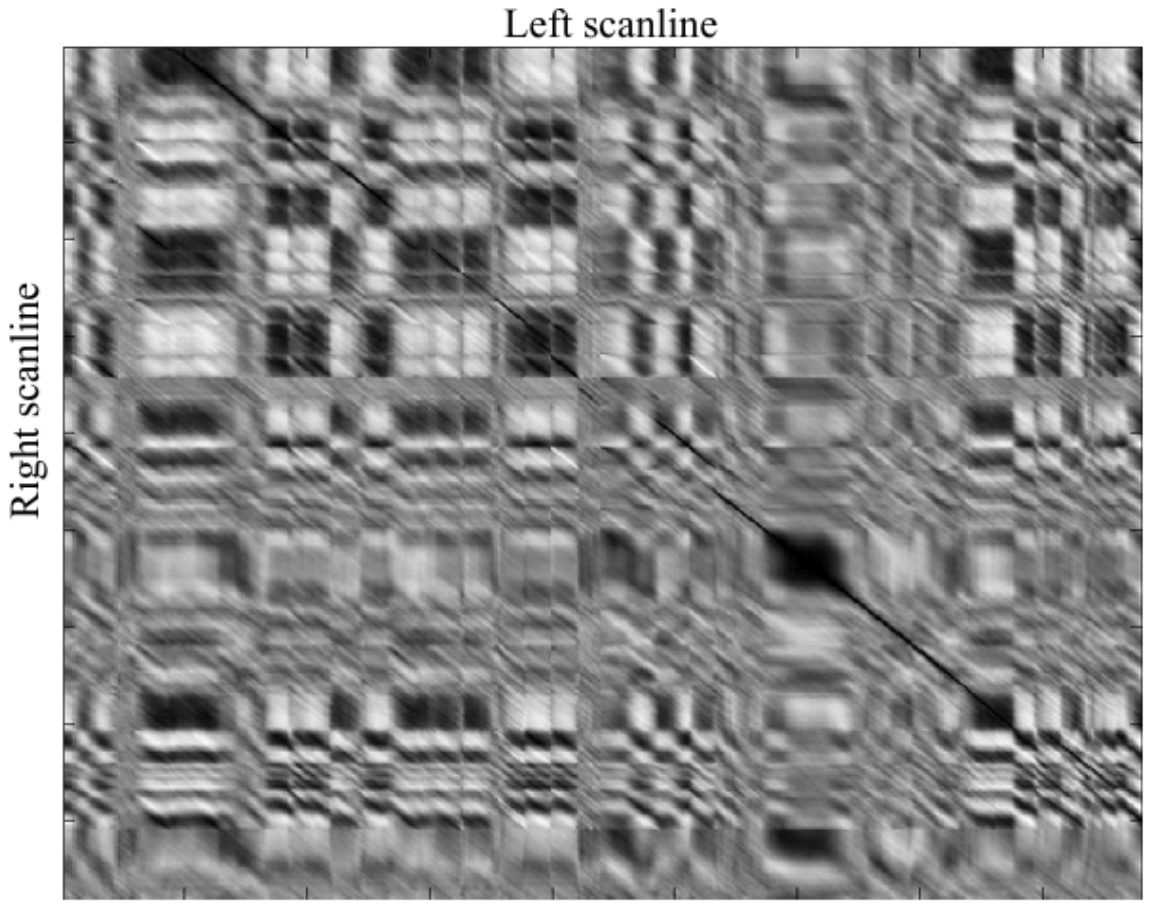
Black pixels are bad disparity values or no matching patches in the right image.

When Stereo Block Matching Fails:

Disparity Space Image (DSI)

The DSI is a matrix C that has in position C(i,j) the match score for patch centered at left pixel i with patch centered at right pixel j. That means that in a row it has all the match scores between right pixel j and pixels in the left corresponding line. So the horizontal direction of the DSI corresponds to the left scanline. The same holds for the vertical direction, for which we have the right scanline.



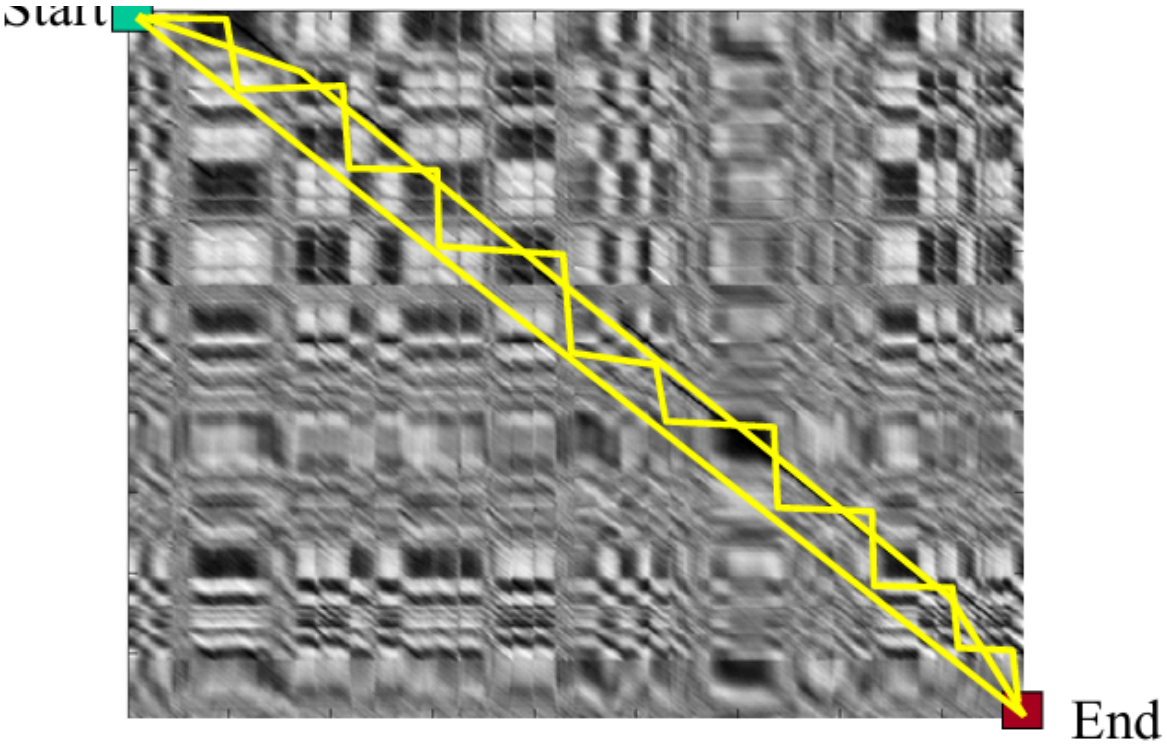


Valid disparities forma diagonal band in the DSI, in fact:

- Low value: disparities cannot be negative, thus any values below the main diagonal are invalid
- High value: disparities cannot be too much large, so the upper triangular region above the diagonal band is invalid

By focusing on the diagonal band we reduce computation complexity. We rearrange the diagonal in a rectangular array.

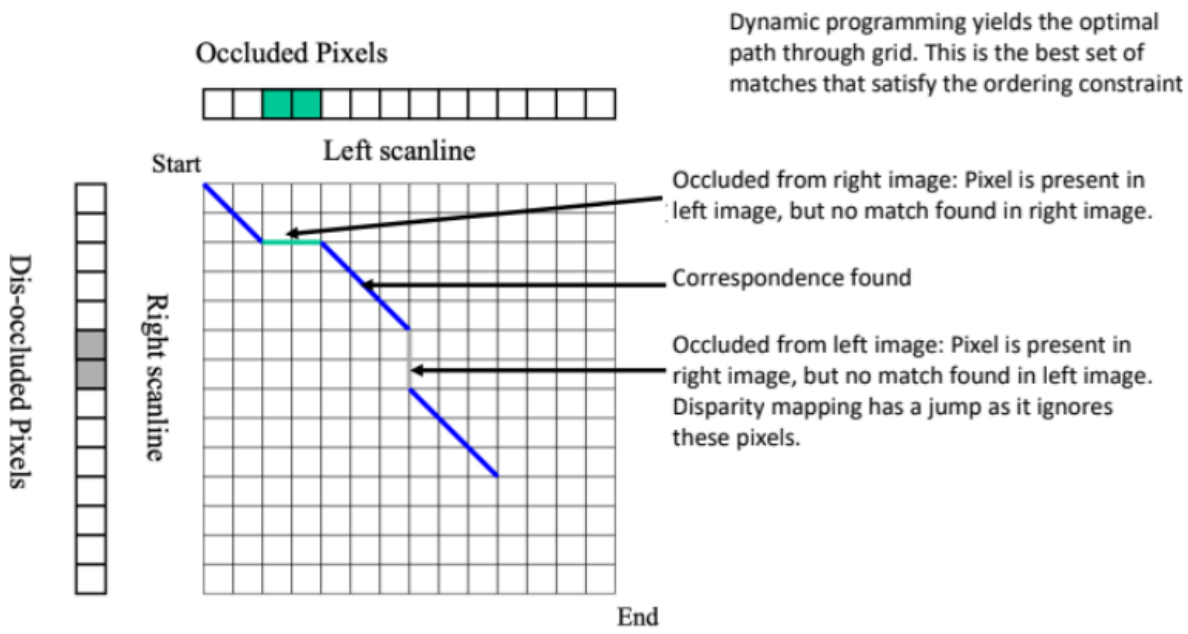
Now our goal is to find a connected path through the DSI

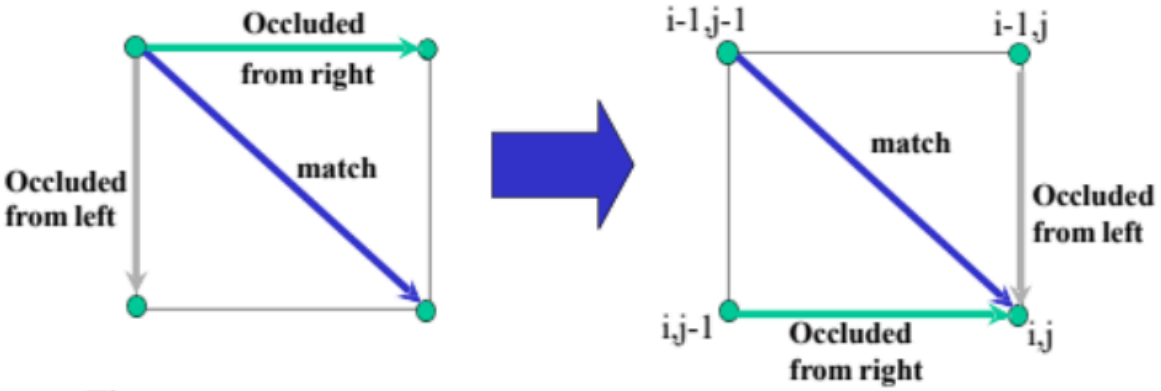


The one with lowest cost, namely the lowest sum of dissimilarity scores along the path, is our goal.

Dynamic Programming to find the path

Greedy Per-pixel path matching often does not satisfy order constraint and produces non-smooth disparity maps, so we can use Dynamic Programming.

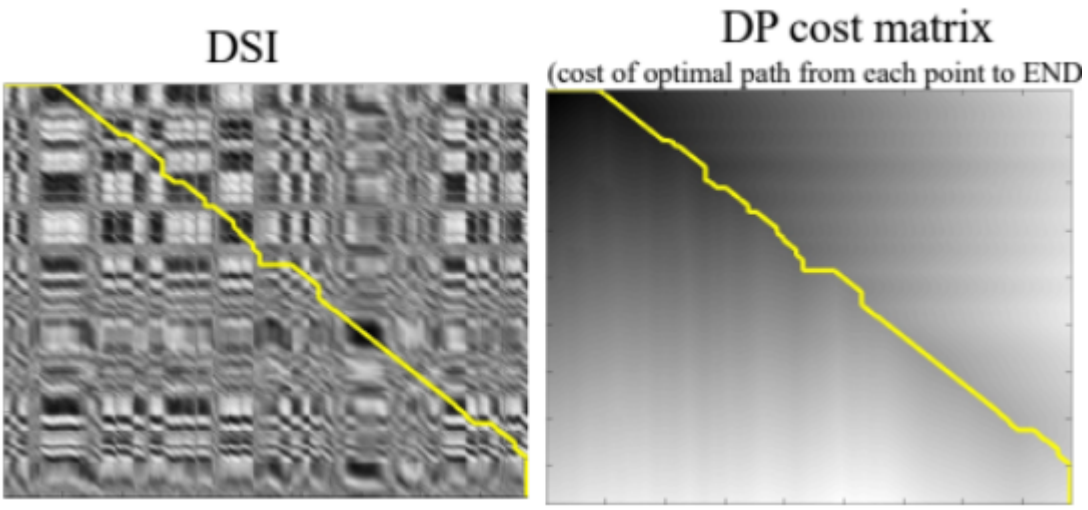




Three cases:

- Matching patches. Cost = dissimilarity score
- Occluded from right. Cost is some constant value.
- Occluded from left. Cost is some constant value.

```
C(i,j)= min([ C(i-1,j-1) + dissimilarity(i,j)
              C(i-1,j) + occlusionConstant,
              C(i,j-1) + occlusionConstant]);
```



Every pixel has a disparity value or an occlusion label

Problems:

- uniqueness constraint: each point in one image should match at most one point in the other image
- smoothness
- disparity should change slowly
- occlusion: occluded pixel is present in the source image but there is no corresponding pixel in the target. Consistency verification: calculating disparities for corresponding points in both images. The disparities should be consistent, i.e. agree within a certain tolerance
- ordering constraint: if pixel (a,b,c) are ordered in left image, it should have the same order in right image, but it's not always true, depends on the depth of the object

Stereo Matching with Graph Cut Algorithm

To remove discontinuities we expect disparity values to change slowly. We do the assumption that depth should change smoothly. That means that is two pixels are adjacent, they should move about the same amount.

We define an energy function that we want to minimize

$$E(d) = \underbrace{E_d(d)}_{\text{data term}} + \lambda \underbrace{E_s(d)}_{\text{smoothness term}}$$

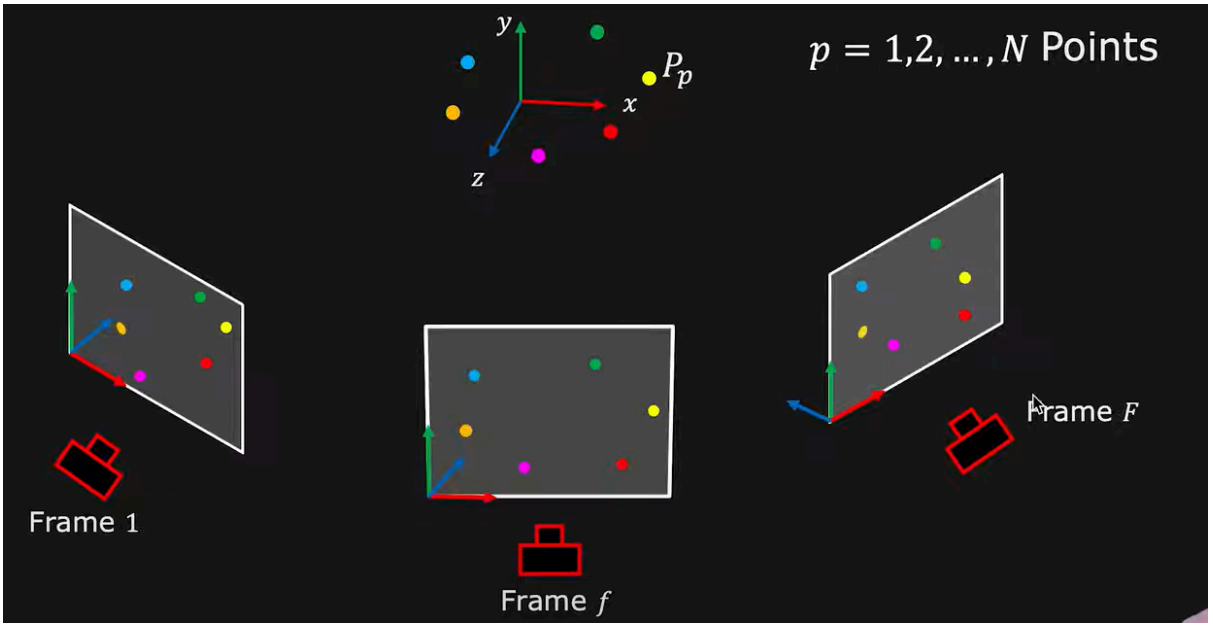
Want each pixel to find a good match in the other image (block matching result)

Adjacent pixels should (usually) move about the same amount (smoothness function)

We can minimize it using Dynamic Programming

Find the 3D points from the 2D Correspondences

Now we have a series of tracks, so the features represented in all the frames of the video



The problem is: given set of corresponding image points (2D): $u_{f,p}$, $v_{f,p}$ where f denotes the frame number, we want to find scene point (3D): p_p

Orthographic Camera

For solving this problem, we use Tomasi-Kanade factorization, that assumes the usage of orthographic cameras[26]: all the projection lines (rays) are orthonormal to the image plane and so parallel to the optical axis.

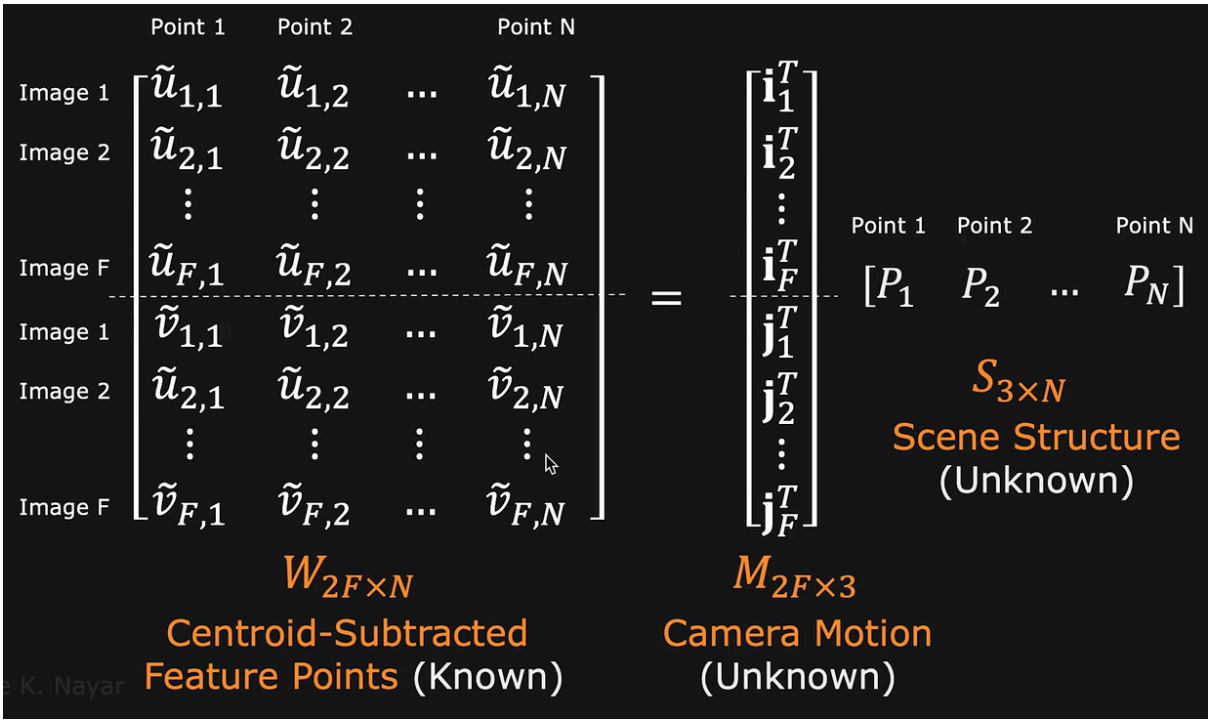
A perspective camera can be approximated as an orthographic camera when the distance of the scene is larger compared to the variation of depth within the scene itself. That means that the magnification $m = f / z_0$ of the camera remains constant for all points in the scene and through the entire sequence of images.

Observation Matrix W

Our goal is to find the 3D point P_P, but the camera position C_f and orientations (i_j, j_f) are unknown[27]. We can remove the dependence from C_f with the centering trick.

Now we can write the Observation Matrix W:

[26] With the advance in the technique, this assumption has been remove
[27] Note that the camera center is not in the pinhole but in a corner of the image plane.



Having rewritten the problem in this form, we can consider W the input of our problem.

Tomasi Kanade Factorization - Rank of Observation Matrix

Very low rank that allows to set a constraint used to decompose the matrix in the structure matrix and the motion matrix, the latter telling how the camera moved.

Rank(W) = Rank(MS) <= min(3, N, 2F)

Now, since N and F are generally big, since we have generally a large number of points that we want to consider, and than 3 frames , we can write

Rank(W) <= 3

This is exploited in the Tomasi-Kanade Factorization algorithm, to factorize W into M and S, solving the SFM problem.