Vision & Perception

Daniel Gigliotti

Images as functions

An image is a 3D tensor of numbers.

We can think of an image as a function

$$f : R^2 \to R$$

$$f : \underbrace{[a \ \times \ b][c \ \times \ d]}_{domain} \to \underbrace{[0, \ 255]}_{range}$$

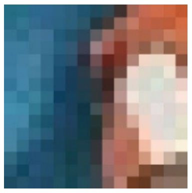
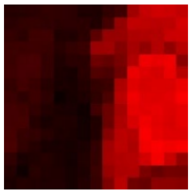
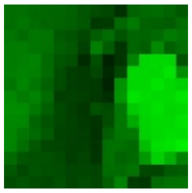
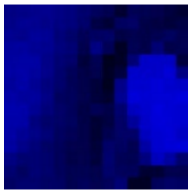
$f(x, y)$ gives the intensity at position (x, y). A digital image is a discrete (sampled, quantized) version of this function:

- **sampling** $\to$ sample a 2D space on a grid
- **quantization** $\to$ clip the value in a grid cell to a specific range (0, 255)

$$f(i, j) = quantize(i\Delta, j\Delta)$$

This formulation holds for single-channel images. A color image is composed of three channels and can be seen as three functions pasted together.

$$f(x, y) = \begin{bmatrix} r(x, y) \\ g(x, y) \\ b(x, y) \end{bmatrix}$$

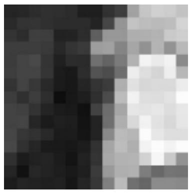
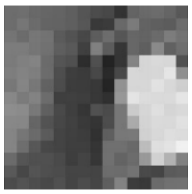
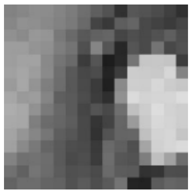





color image patch

R channel

G channel

B channel

actual intensity per channel:

Image processing

We have two types of image processing:

- **image warping** changes the domain (pixel positions) of the image; points are mapped to other points without changing their colors;
- **image filtering** changes the range (pixel values) of the image; the colors of the image are altered without changing the individual pixel position;

Image filtering

We have two types of image filtering:

- **point processing**: change the range of a pixel only accounting for its current value;
- **neighborhood processing**: change the range of a pixel taking into account the values of its neighbors;

Filtering allows us to make a new image whose pixels are a combination of the original pixels in the corresponding area. We modify the pixels in an image based on some function of a local neighborhood, in the source image, of each pixel. Filters can be both linear and non-linear.

$$\begin{tikzcd}
 & \textbf{Image warping} & \\
\textbf{Image processing} \arrow[ur] \arrow[dr] & & \textbf{Point processing} \\
 & \textbf{Image filtering} \arrow[ur] \arrow[dr] & \\
 & & \textbf{Neighborhood processing}
\end{tikzcd}$$

Point processing

Point processing in image filtering is a technique where a single pixel in an image is processed independently of its neighboring pixels. This is in contrast to neighborhood (or spatial) processing, where the value of a pixel is determined by the values of neighboring ones.

Examples of point processing include:

- **brightness and contrast adjustment**: this involves adjusting the overall brightness and contrast of an image by multiplying the value of each pixel by a constant factor;
- **gamma correction**: this involves adjusting the brightness of an image by applying a non-linear transformation to pixel values;
- **histogram equalization**: this involves adjusting the distribution of pixel values in an image to improve its overall contrast;
- **thresholding**: this involves converting every pixel within a certain range to have a fixed value;
- **color balance**: this involves adjusting the color balance of an image by altering the relative intensities of its color channels;
- **colorization**: this involves adding color information to a gray-scale image by mapping the intensity values to a color map;
- **negation**: this involves inverting the color of an image by subtracting the pixel values from the maximum value (255);

Convolution and correlation

Correlation and convolution are both mathematical operations that are used in signal processing and image processing.

They involve multiplying a signal/image by a filter kernel. The result of this operation is a new image where each pixel is a weighted sum of the pixels in the original image, but there are important differences between the two.

Correlation

Definition for 2D correlation for image filtering:

$$(f \otimes g)(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(i, j) I(x + i, y + j)$$

$$f(i, j) \text{ is the kernel}$$
$$I(x + i, j + j) \text{ is the input image}$$
$$(f \otimes g)(x, y) \text{ is the filtered image}$$

Notice the lack of a flip in $I(x + i, y + j)$.

Convolution

Definition for 2D convolution for image filtering:

$$(f * g)(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(i, j) I(x - i, y - j)$$

$$f(i, j) \text{ is the kernel}$$
$$I(x - i, j - j) \text{ is the input image}$$
$$(f * g)(x, y) \text{ is the filtered image}$$

The patch is flipped both horizontally and vertically.

Correlation and Convolution order

Right: correlation order in the linear combination
Left: convolution order in the linear combination

1	2	3
4	5	6
7	8	9

9	8	7
6	5	4
3	2	1

Padding

Just applying the convolution operator will reduce the size of the image and this is not always desirable. **Padding** refers to the technique of adding extra rows and columns of zeros around the edges of the image. Padding is commonly used to preserve spatial dimensions during convolutional operations and to prevent information loss at the edges of the feature map.

There are three common types of padding:

- **Same padding**: the input image is padded with zeros in such a way that the output has the same spatial dimensions as the input. This is achieved by adding $\lfloor((filter\_size - 1)/2)\rfloor$ rows and columns of zeros on both sides.
- **Valid padding**: the input image is not padded at all. This means that the filter is only applied to positions where it can completely overlap with the input. As a result, the output has smaller spatial dimensions.
- **Full padding**: the input is padded with zeros in such a way that the output has the maximum possible spatial dimensions. This is achieved by adding $\lfloor(filter\_size - 1)\rfloor$ rows and columns of zeros on both sides. The resulting output has larger spatial dimensions.

Filters

The operations applied with **filters** are typically convolutions and correlations. In both cases we replace a pixel in the new image with a combination (a weighted sum) of the respective pixel in the old image and its neighbors. The neighbors are multiplied by a set of coefficients, known as **kernel**.

Convolution between an image patch and the box filter:

$$\begin{array}{|c|c|c|}\hline 10 & 5 & 3 \\ \hline 4 & 6 & 1 \\ \hline 1 & 1 & 8 \\ \hline \end{array} * -1/9 \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|}\hline & & \\ \hline & 4.3 & \\ \hline & & \\ \hline \end{array}$$

Convolution between an image patch and a custom filter:

$$\begin{array}{|c|c|c|}\hline 10 & 5 & 3 \\ \hline 4 & 6 & 1 \\ \hline 1 & 1 & 8 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 0.5 & 0 \\ \hline 0 & 1 & 0.5 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|}\hline & & \\ \hline & 8 & \\ \hline & & \\ \hline \end{array}$$

If the kernel is constant while shifting it on all the pixel locations, the filter is called **shift invariant** (all the pixels use the same linear combination of their neighbors).

Linear filters can be implemented quickly and efficiently, but they can also be less effective than **non-linear filters** for certain types of image processing tasks. However, they are widely used due to their simplicity and effectiveness.

A 2D filter is **separable** if it can be written as the product of a column and a row vector:

$$\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|}\hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array} \times \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline \end{array}$$

2D convolution with a separable filter is equivalent to two 1D convolutions (with the column and row filters). If the image has $m \times n$ pixels and the kernel filter has size $l \times l$:

- the cost of a convolution with a non-separable filter is $l^2 \times (m \times n)$
- the cost of a convolution with a separable filter is $2 \times (l \times (m \times n))$

It is always convenient to use separable filters.

Box filter

The box filter is the simplest kind of filter we can imagine: it simply takes the average of all the pixels under the kernel area. It is linear (it fulfills both the superposition and the homogeneity principles) and separable (can be written as a product of of a column and a row vector).

The box filter takes the average of the pixels under the kernel area. By doing so, it smoothes out the image, reducing noise. It helps suppressing sharp transitions, however it also tends to blur edges and fine details along with the noise, which may not be desirable in certain applications.

Convolution between an image patch and the box filter:

$$\begin{array}{|c|c|c|}\hline 10 & 5 & 3 \\ \hline 4 & 6 & 1 \\ \hline 1 & 1 & 8 \\ \hline \end{array} \text{ * -1/9} \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|}\hline & & \\ \hline & 4.3 & \\ \hline & & \\ \hline \end{array}$$

Gaussian filter

Gaussian filter uses the Gaussian function to smooth images and reduce noise; The Gaussian function is a bell-shaped curve that gives more weight to pixels in the center of the filter and less weight to pixels at the edges (weight falls off with distance from center); the resulting smoothing effect preserves edges and details in the image better than box filter; a Gaussian effect can be obtained from a convolution between the image and a Gaussian function;

0.003	0.013	0.022	0.013	0.003
0.013	0.022	0.097	0.022	0.013
0.022	0.097	0.159	0.097	0.022
0.013	0.059	0.097	0.059	0.013
0.003	0.013	0.022	0.013	0.003

Above, weight contributions of neighboring pixels by nearness in a $5x5$ Gaussian filter with $\sigma = 1$. The kernel values are sampled from the 2D Gaussian function:

$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

Weights fall off with distance from the center pixel. The kernel can theoretically be infinite but in practice it is truncated to some maximum distance. Selecting where to truncate the distance of a Gaussian filter depends on the specific application and the desired level of smoothing. However, a common heuristic for selecting the truncation distance is to choose the distance at which the Gaussian function falls to a certain percentage of its maximum value.

For example, one common choice is to truncate the Gaussian filter at a distance where the Gaussian function falls to 1% or less of its maximum value. This means that the filter will consider only the pixels within a certain radius from the center of the filter, and pixels farther away will not have a significant effect on the output image.

Another heuristic for selecting the truncation distance is to use the standard deviation of the Gaussian distribution. The standard deviation determines the width of the Gaussian function, and choosing the truncation distance based on the standard deviation ensures that the filter considers only the most relevant pixels within a certain distance from the center of the filter.

Difference between Gaussian filter and Box filter

Gaussian filter and box filter are two commonly used image smoothing filters in image processing. Although **both filters are used for the same purpose of reducing noise and smoothing an image**, they produce different results due to their different characteristics.

The main difference between a Gaussian filter and a box filter is that a Gaussian filter uses a Gaussian function to calculate the weights of the neighboring pixels, while a box filter uses a uniform weight for all neighboring pixels within a given window.

In terms of the output image, the main difference between the two filters is the level of smoothing and blurring that they produce. **A Gaussian filter produces a smoother output image with less blurring compared to a box filter**. This is because the Gaussian filter assigns higher weights to the pixels closer to the center of the filter window and lower weights to the pixels farther away, resulting in a smoother transition between neighboring pixels. In contrast, **a box filter assigns equal weights to all neighboring pixels, resulting in a more uniform smoothing and potential blurring of edges and details in the image**.

Therefore, if the goal is to reduce noise and smooth the image while preserving the edges and details, a Gaussian filter is generally preferred. On the other hand, if the goal is to simply reduce noise and blur the image uniformly, a box filter may be more suitable. However, it is important to note that the choice of filter depends on the specific application and the desired level of smoothing, and it may require some experimentation to find the most appropriate filter for a given image processing task.

In the frequency domain, the box filter and the Gaussian filter have different effects on the image:

- the **box filter** has a rectangular frequency response, which **leads to a sinc-like function in the spatial domain**. The sinc function has oscillations, which can lead to **ringing** artifacts in the image (ripple like structures);
- the **Gaussian filter** has a smooth, bell-shaped frequency response. This results in a smoother, more gradual transition between the filtered and unfiltered regions of the image. Gaussian filtering typically **produces fewer ringing artifacts** than box filtering;

Sharpening filters

A sharpening filter is a type of image filter that enhances the edges and details of an image, making it appear sharper. It works by amplifying the high-frequency components of the image, which correspond to the edges and fine details (edges and fine details are high frequency components because there is rapid change in pixel intensity or color).

To obtain a sharpening filter from an original image and a blurring filter, we can use a technique called **unsharp masking**. Here are the steps:

- Apply the blurring filter to the original image (low frequency components).
- Subtract the blurred image from the original image to obtain the high-frequency components.
- Add the result back to the original image. This will create a sharpened version of the image.

The resulting filter can be applied to other images to enhance their sharpness. However, it's important to note that this technique can also amplify noise in the image, so it should be used with caution.

Linear and non-linear filters

Linear Filters	**Non-Linear Filters**
Gaussian Filter	Median Filter
Mean Filter	Bilateral Filter
Box Filter	Non-local Means Filter
Sobel Filter	Maximum Filter
Laplacian Filter	Minimum Filter

Table 1: Examples of Linear and Non-Linear Image Filters

Why is the Gaussian filter linear?

The Gaussian filter is a linear filter because it satisfies the two fundamental properties of linearity: **superposition** and **homogeneity**.

- **Superposition** means that the response of the filter to a sum of inputs is equal to the sum of the responses to each input. In other words, if we apply the filter to a weighted sum of two images, the resulting output will be the weighted sum of the filtered versions of each image. The Gaussian filter satisfies this property because it convolves the input image with a linear, separable kernel that can be expressed as the sum of two 1D Gaussian functions.

- **Homogeneity** means that the response of the filter is proportional to the input signal. In other words, if we scale the input signal by a constant factor, the output of the filter will also be scaled by the same factor. The Gaussian filter satisfies this property because it convolves the input image with a kernel that is a function of the distance from the center of the kernel, and this distance is proportional to the magnitude of the input signal.

In layman's terms, a filter is linear if its response to a sum of two signals is the same as the sum of the individual responses. This is equivalent to saying that each output pixel is a weighted summation of some number of input pixels. This is not true for non-linear filters, like the Maximum filter, where the output pixel is the maximum among the pixels in the considered window.

Why is the Gaussian filter separable?

The Gaussian filter is separable because it can be expressed as the convolution of two 1D Gaussian filters, one applied horizontally and one applied vertically (the proof is omitted). We can apply these filters separately to achieve the same result as applying the full 2D Gaussian filter, but with less computational cost. This is the key reason why the Gaussian filter is commonly used in image processing applications.

Feature Detection and Feature Description

Feature detection is the process of finding key points or regions in an image that are relevant for a specific task, such as object recognition or image matching. For example, feature detectors like SIFT (Scale-Invariant Feature Transform) or SURF (Speeded Up Robust Features) can identify distinctive points in an image, such as corners or edges, which can be used to describe the image.

Commonly used feature detection algorithms are:

- Harris Corner Detector
- Canny Edge Detector
- Scale Invariant Feature Transform (SIFT) Detector
- Speeded-Up Robust Features (SURF) Detector

Feature description, on the other hand, is the process of representing the key points or regions identified by feature detection in a compact and meaningful way. The goal of feature description is to create a representation that is invariant to geometric transformations, such as rotation or scaling, so that the same feature can be recognized in different images.

Common methods for feature description include:

- Scale Invariant Feature Transform (SIFT) Descriptor
- Speeded-Up Robust Features (SURF) Descriptor
- Histograms of Oriented Gradients (HOG)

Once you have identified and represented the features, you can do whatever you want with them: feature matching, feature tracking, ...

Edges

In computer vision, edges refer to the boundaries between objects or regions of interest in an image. They are **sudden changes in the intensity or color of an image**, and often highlight important features.

Edges are important for two main reasons:

- Most semantic and shape information can be deduced from them, so we can perform object recognition and analyze perspectives and geometry of an image;
- They are a more compact representation than pixels;

How can we find discontinuities? Since the image can be seen as a function, we can take its derivative: derivatives are large at discontinuities. How can we differentiate a discrete image (or any other discrete signal)? We can use finite differences:

Definition of a derivative using forward difference:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Definition of a derivative using central difference:

$$f'(x) = \lim_{h \to 0} \frac{f(x+0.5h) - f(x-0.5h)}{h}$$

For discrete signals, remove the limit and set $h = 2$:

$$f'(x) = \frac{f(x+1) - f(x-1)}{2}$$

Same thing for 2D discrete functions: $f(x+1, y+1)$ represents an adjacent pixel, the lower right diagonal one, with respect to $f(x, y)$, $f(x-1, y)$ the left one and so on.

Image gradients

Image gradients are representations of the rate of change in the intensity or color in an image in a particular direction (e.g. x or y)

Gradients are useful in various image processing tasks such as edge detection, feature extraction, and image segmentation. They provide information about the structure and properties of an image, and are used to identify areas of high intensity or color change, which correspond to image features like edges and corners.

- An image gradient, **which is a generalization of the concept of derivative to more than one dimension**, points in the direction where intensity increases the most;
- After gradient images have been computed (we have two directions since the image is a 2D function, thus we need to compute two gradient images), pixels with large gradient values become possible edge pixels;
- The pixels with the largest gradient values in the direction of the gradient become edge pixels; edges can be traced in the direction perpendicular to the gradient direction.

Note that plotting the pixel intensities of the gradient often results in noise, making it almost impossible to identify where an edge is by only taking the first derivative of the function: we will later use the second derivative.

To attenuate this problem, **when using derivative filters, it is critical to blur first**.

Sobel filter

One of the earliest and most well-known approaches for edge detection involves the Sobel operator. The basic idea is to quantify the rate of change of the pixel intensities throughout an image. In other words, it seeks to calculate the spatial derivative of each pixel. At the boundary between objects or regions in an image, there is usually a rapid shift in pixel intensities. Consequently, the magnitude of the derivative at these boundaries tends to be relatively large. Finding the areas in an image with a large derivative can provide insight into the locations of edges and contours. Traditionally, the Sobel technique considers a 3×3 pixel neighborhood.

A great explanation on how to compute the Sobel filter can be found here

$$\begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 1 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline \end{array}$$

Vertical Sobel filter, finds changes (edges) in x direction, thus vertical edges

$$\begin{array}{|c|c|c|} \hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array} = \begin{array}{|c|} \hline -1 \\ \hline 0 \\ \hline 1 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline \end{array}$$

Horizontal Sobel filter, finds changes (edges) in y direction, thus horizontal edges

- **Horizontal Sobel filter** $\rightarrow$ when this filter is convolved with an image, it computes the horizontal gradient (it highlights the edges in the image that have a strong change in intensity in the horizontal direction); this means that it can be used to detect edges that are oriented horizontally in the image;

-1	-2	-1
0	0	0
1	2	1

- **Vertical Sobel filter** $\rightarrow$ when this filter is convolved with an image, it computes the vertical gradient (it highlights the edges in the image that have a strong change in intensity in the vertical direction); this means that it can be used to detect edges that are oriented vertically in the image;

-1	0	1
-2	0	2
-1	0	1

In practice, these two filters are often used together in a technique called **Sobel edge detection**. By computing the magnitude of the gradient at each pixel using the horizontal and vertical Sobel filters, we can identify the regions in the image where there is a strong change in intensity in any direction, which are likely to correspond to edges in the image. **This is a rather naive technique**.

Other derivative filters

Depending on how you compute the distance between two pixels, the neighborhood size and so on, you can have different derivative filters:

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Horizontal and vertical Sobel filters

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

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

Horizontal and vertical Scharr filters

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

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

Horizontal and vertical Prewitt filters

1	0
0	-1

0	1
-1	0

Horizontal and vertical Roberts filters

Computing image gradients

- Select the derivative filters. In this example we will use the Sobel filter:

$S_x =$

-1	-2	-1
0	0	0
1	2	1

$S_y =$

-1	0	1
-2	0	2
-1	0	1

- Convolve with the image to compute derivatives:

$$\frac{\partial f}{\partial x} = S_x \otimes f \qquad \frac{\partial f}{\partial y} = S_y \otimes f$$

- Form the image gradient and compute its direction and amplitude:

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

Gradient

The gradient points in the direction of most rapid increase in intensity. The gradient direction and magnitude can be computed this way:

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

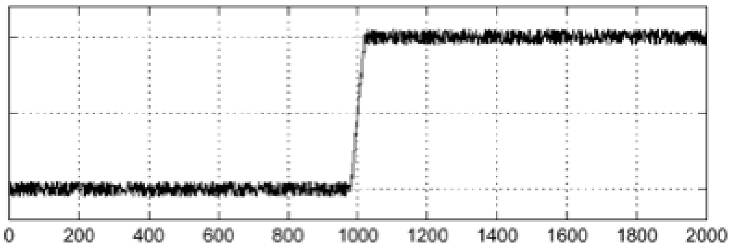
Direction

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

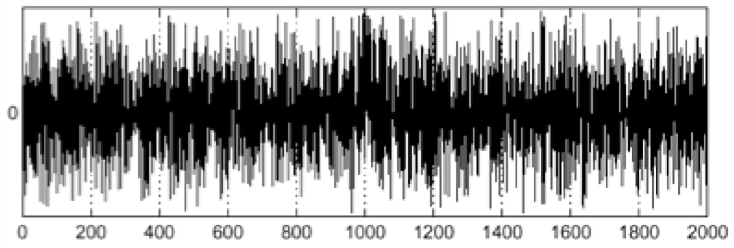
Amplitude

Using image gradients for edge detection

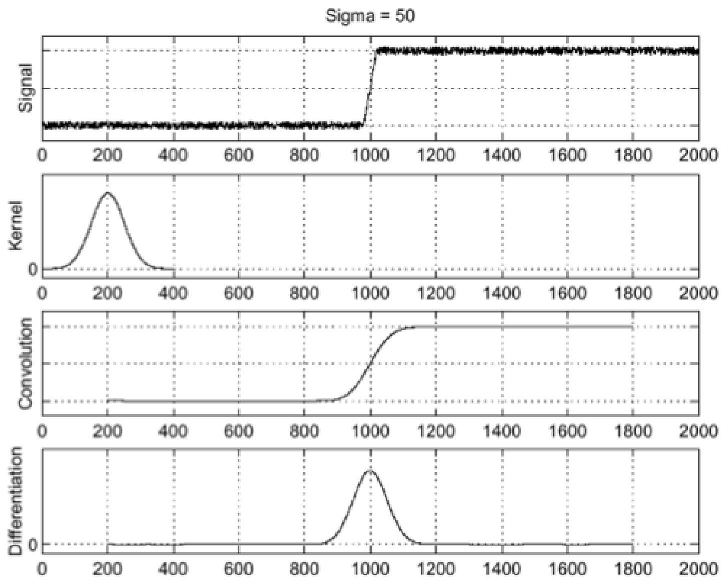
Given a signal



Directly taking its derivative results in this:



Blurring before taking the derivative:



We can focus on peaks in the convolution between the image and a blurring filter to find edges.

Canny Edge detector

The Canny Edge Detector is a technique that resorts on the gradient of the image to detect edges. Edges correspond to a sudden change in pixel intensity. The algorithm requires the image to be grayscale. It consists of 5 steps:

0. **Grayscale the image**.
1. **Noise reduction**: resorting on the gradient, the pipeline is very susceptible to noise in the image, so the first step is to apply a Gaussian filter to blur it. We use the Gaussian over a Box filter for various reasons, first two among them are the fact that Box filter leaves ringing artifacts in the image and the fact that Gaussian filter better preserves the edges.
2. **Gradient computation**: the smoothed image is then filtered with a Sobel kernel (first derivative) both in the horizontal direction (G_x) and in the vertical direction (G_y); once we have the image gradient we can compute the orientation and magnitude of the gradient at each pixel.
3. **Non-maximum suppression**: ideally, the final image should have thin edges, thus we must perform non-maximum suppression to thin out the edges; the algorithm goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions. The purpose of the algorithm is to check if the pixels on the same direction are more or less intense than the ones being processed. If the pixel (i, j) is being processed and the pixels on the same direction are (i, j-1) and (i, j+1), if one of those two pixels is more intense than the one being processed, then only the more intense one is kept.
4. **Double threshold**: the double thresholding step is then applied to the resulting image with thin edges; it aims at identifying 3 kinds of pixels:
   - **Strong pixels** are pixels that have an intensity so high that we are sure they contribute to the final edge;
   - **Weak pixels** are pixels that have an intensity value that is not enough to be considered as strong ones, but yet not small enough to be considered as nonrelevant for the edge detection;
   - Other pixels are considered as **non-relevant** for the edge.

   The threshold values are chosen empirically.
5. **Linking**: based on the threshold results, the linking consists in transforming weak pixels into strong ones if and only if at least one of the pixels around the one being processed is a strong one.

Threshold and linking steps are also referred to as hysteresis.

Laplace filter

Laplace filter is a **second derivative filter**.

The first-order finite difference

$$f'(x) = \lim_{h \to 0} \frac{f(x + 0.5h) - f(x - 0.5h)}{h}$$

results in a 1D derivative filter:

1	0	-1

The second-order finite difference

$$f''(x) = \lim_{h \to 0} \frac{f(x + h) - 2f(x) + f(x - h)}{h^2}$$

results in a 1D Laplace filter:

1	-2	1

-1	-1	-1
-1	8	-1
-1	-1	-1

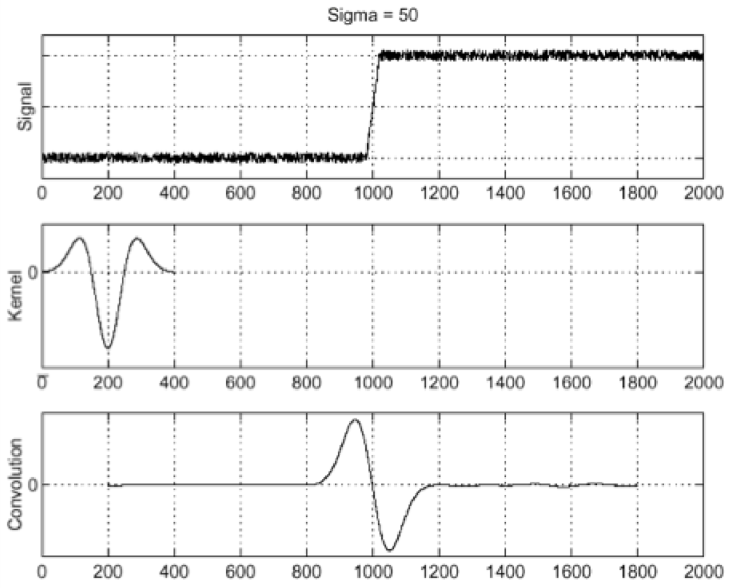
0	-1	0
-1	4	-1
0	-1	0

Laplacian operator with and without diagonals

Laplacian of Gaussian (LoG) filter

Instead of convolving the image with the kernel and **then** taking the derivative (expensive operation since the derivative is taken with respect to the whole image), we can first compute the derivative of the filter (small) and then convolve it with the image.

We obtain this:



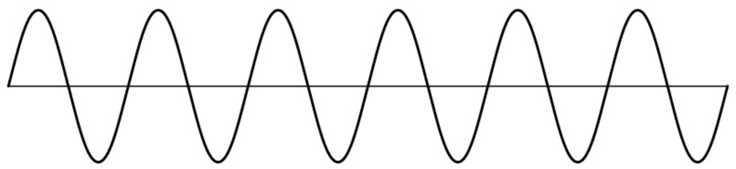
With the second derivative (laplacian) we have a **zero-crossing behaviour at edges**, whilst in with the first derivative we had a peak in the same spot. It is **easier to search for a zero** than to find a maximum.

Building off of this procedure we can design an edge detector.

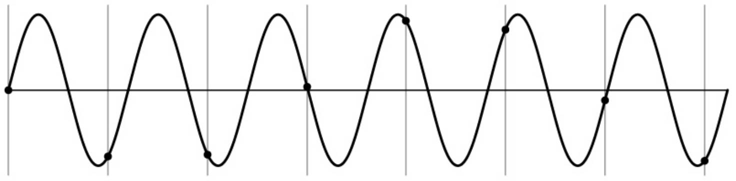
Naive Image Downsampling

How could we reduce the image size? A naive approach could be to throw away half the rows and/or columns. **Why half? Is it enough?** Images are a discrete, sampled, representation of a continuous world. If you undersample the image, like any other kind of signal, you won't be able to reconstruct it without information loss.

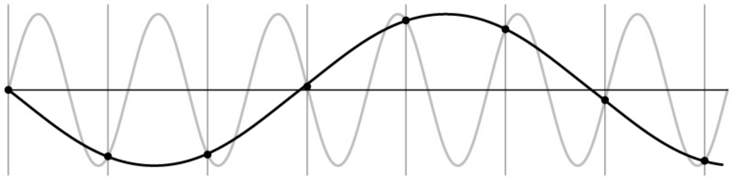
Consider a very simple example, a sine wave:



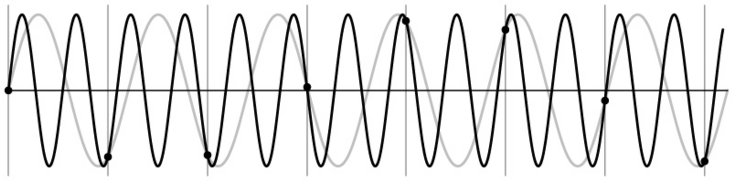
Suppose you sample the signal like so:



When you try to reconstruct the signal, you can confuse it with one of lower frequency:



Note that we could always confuse the signal with one of higher frequency even if we sample correctly:



We have undersampling when we can disguise a signal as one of lower frequency. The resulting effect is called **aliasing**.

The **Nyquist sampling theory** states that the sample rate needs to be at least 2 times the highest frequency component (cutoff frequency) of your input signal to prevent aliasing; that rate is called **Nyquist rate**. Aliasing is a phenomenon where higher frequency components fold back into your signal and cause interference.

How to deal with aliasing?

- **Oversample the signal**: you should sample with a rate over the Nyquist rate.
- **Smooth the signal**: remove some of the details that cause aliasing; this causes information loss but lets you have better aliasing artifacts.

Dealing with images, we can apply a smoothing filter (Gaussian, box, ...) first, then proceed to downsample (throw away half the rows and columns).

How much smoothing do we need to avoid aliasing? How many samples do we need to avoid aliasing? **Enough to reach the Nyquist limit of 2 times the highest frequency reached by the image**. How can we practically reduce aliasing? By smoothing the signal we remove high frequency components (features) and thus lower the Niquist rate ($\geq$ two times the maximum frequency).

Gaussian Image Pyramid

Image pyramids are a type of multi-scale representation of an image, where the original image is successively down-sampled to create a series of smaller versions, each representing a different level of detail. They are important because they allow us to perform operations such as object detection, feature extraction, and image segmentation at multiple scales, which can be useful in a variety of computer vision applications.

- **Multi-scale image analysis**: By using image pyramids, we can process an image at different scales, allowing us to detect objects or features that may appear at different sizes in the image. This can be useful in scenarios where the size of an object in the image is not known in advance, or when we need to detect objects or features that may have different sizes in different parts of the image.
- **Feature extraction**: Image pyramids can be used for feature extraction, where we extract features from the different levels of the pyramid and use them to represent the image at multiple scales. This can be useful in tasks such as image classification, where we want to use features that capture different levels of detail in the image.
- **Image blending**: Image pyramids can be used for image blending, where we combine two images of different sizes or resolutions. By constructing image pyramids for each image, we can blend them at different levels of the pyramid to create a seamless blend between the images.

Overall, image pyramids are an important tool in computer vision and image processing, allowing us to process images at multiple scales and extract features that capture different levels of detail.

Image pyramids allow us to perform operations on images at different resolutions, which can be **computationally efficient**. For example, instead of processing the original image at high resolution, we can first down-sample the image and then perform the operation on the smaller image, reducing the computational cost. All this at what cost? How much is the image pyramid bigger than the original image? **The whole image pyramid is just $4/3$ times the size of the original image**.

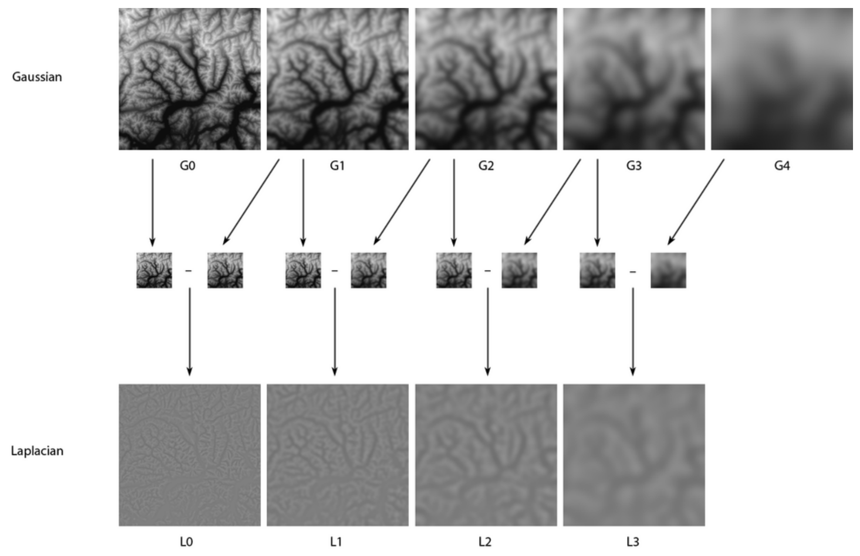
Laplacian Image Pyramid

Going up a level in a Gaussian Image Pyramid we loose features, high frequency components. Mostly large uniform regions are preserved. It is not possible to reconstruct the original image from the image at the upper level in the pyramid alone. This is why we introduce the **Laplacian Image Pyramid**.

The Laplacian pyramid is constructed by decomposing the original image into a series of bandpass-filtered images, where each image is obtained by subtracting the upsampled version of the next level's image from the current level's image. This difference image is also called **residual**, as it represents the high-frequency details or edges in the image that were removed by the bandpass filtering.

The Laplacian pyramid is typically constructed by first computing a Gaussian pyramid and then subtracting the upsampled version of each Gaussian pyramid level from the next level in the Gaussian pyramid.

Thus, we have both the residual image, which contains the high-frequency details or edges that were removed during the Gaussian filtering, and the downsampled images, which contain the low-frequency information. By storing both the residual and downsampled images we have a multi-scale representation of the image that can be used for tasks such as image compression, feature extraction, and image blending.



Constructing a Laplacian Pyramid

The k-th level of Laplacian pyramid can be obtained by the following formula:

$$L_k(I) = G_k(I) - u(G_{k+1}(I))$$

I is the input image
$L_k(I)$ is the k-th level of the Laplacian pyramid
$G_k(I)$ is the k-th level of the Gaussian pyramid
$u(\cdot)$ is a 2x scale-up operation

These are the steps to build a Laplacian pyramid:

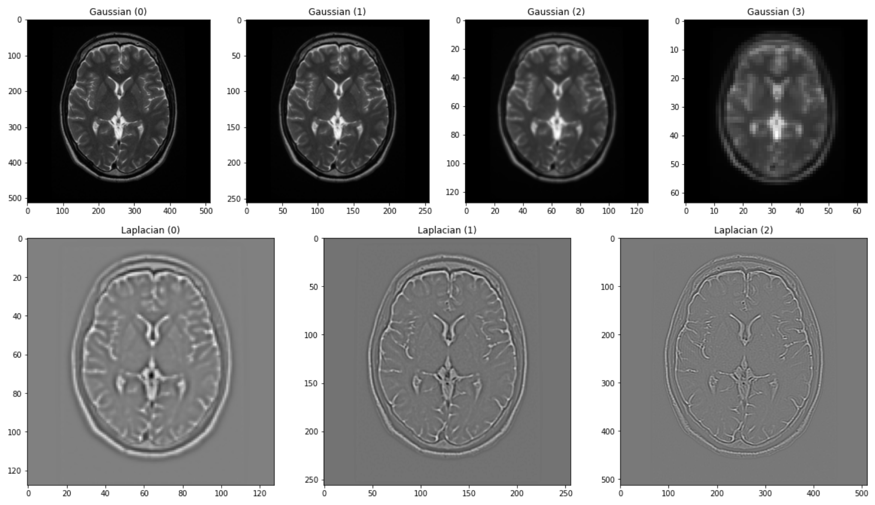
1. Create a Gaussian image pyramid. The Gaussian pyramid is computed by repeatedly applying a Gaussian filter to the image and then down-sampling it.

2. Create a Laplacian image pyramid by subtracting an up-sampled and blurred version of the image from the original image at each level of the pyramid. To do this, follow these steps:

    - Starting with the higher level of the Gaussian pyramid ($k + 1$ if we are on level k), up-sample it.
    - Apply a Gaussian filter to the up-sampled image.
    - Subtract the resulting blurred image from the corresponding level of the Gaussian pyramid to get the Laplacian image.

3. Repeat step 2 for all levels of the Gaussian pyramid, except the lowest level, which does not have a corresponding lower level to subtract from.

```python
levels = 3

# Create a Gaussian Pyramid
lower = img.copy()
gaussian_pyr = [lower]
for i in range(levels):
   lower = cv2.pyrDown(lower)
   gaussian_pyr.append(lower)

# Last level of Gaussian remains same in Laplacian
laplacian_top = gaussian_pyr[-1]

# Create a Laplacian Pyramid
laplacian_pyr = []
for i in range(levels,0,-1):
   size = (gaussian_pyr[i - 1].shape[1], gaussian_pyr[i - 1].shape[0])
   gaussian_expanded = cv2.pyrUp(gaussian_pyr[i], dstsize=size)
   laplacian = cv2.subtract(gaussian_pyr[i-1], gaussian_expanded)
   laplacian_pyr.append(laplacian)

plot_all(gaussian_pyr, ['Gaussian ({})'.format(i) for i in range(len(gaussian_pyr))])
plot_all(laplacian_pyr, ['Laplacian ({})'.format(i) for i in range(len(laplacian_pyr))])
```



Steerable pyramid

A steerable pyramid is a type of image decomposition algorithm that can be used for tasks such as image compression, feature extraction, and object recognition.

The basic idea behind a steerable pyramid is to construct a series of filters that can be applied to the image at different scales and orientations. The filters are designed in such a way that they can be steered in any direction, which means that they can be rotated to match the orientation of the features in the image. This allows the steerable pyramid to capture information in a directionally selective way, which is useful for tasks such as edge detection or texture analysis.

The steerable pyramid can be constructed using a set of bandpass filters that are applied to the image at different scales and orientations. The filters are typically implemented using a wavelet transform, which decomposes the image into a series of frequency bands. Each frequency band is then convolved with a set of filters that are steerable in different directions.

Once the steerable pyramid is constructed, it can be used for a variety of image processing tasks. For example, the pyramid can be used for image compression by discarding the high-frequency subbands, which contain fine details that are not essential for reconstructing the image. The pyramid can also be used for feature extraction by analyzing the responses of the different subbands to detect edges, corners, or other features in the image.

Image frequency

We can easily think about audio signals as sound waves varying at certain frequencies. High-frequency is a high pitched noise, like a bird chirp, while low frequency sounds are low pitch, like a deep voice or a bass drum. For sound, frequency actually refers to how fast a sound wave is oscillating. As for sound, frequency in images is a rate of change. While sound varies across time, images varies across space. An high frequency image is one where the intensity of pixels changes a lot, a low frequency image may be one that is relatively uniform in brightness or changes very slowly.

The Fourier Transform (FT) is an important image processing tool which is used to decompose an image into its frequency components. The output of an FT represents the image in the frequency domain, while the input image is the spatial domain (x, y) equivalent. In the frequency domain image, each point represents a particular frequency contained in the spatial domain image. So, for images with a lot of high-frequency components (edges, corners, and stripes), there will be a number of points in the frequency domain at high frequency values.

Fourier Series

The Fourier theorem state that any univariate function (signal) can be expressed as a weighted sum of sines and cosines of various frequencies.

$$A \sin(\omega x + \phi)$$

Basically, if we add enough of these blocks we can build any periodic signal we want. This thing also works for non-periodic signals, as long as the area under the curve is finite.

Fourier Transform

The mathematical tool used to do this is known as the Fourier Transform:

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{j2\pi kx} dk$$

This is used for continuous signals. What about images? Well, as for sound, frequency in images is a **rate of change**. So, what does it means for an image to change? Differently from sound, which varies across time, **images change in space**.

- a **high frequency image** is one where the intensity of pixels changes a lot;

- a **low frequency image** may be one that is relatively uniform in brightness or changes very slowly;

For discrete signals we will use the discrete Fourier Transform:

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

Being images 2D, we use a 2D Fourier Transform:

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{-j2\pi(ux/M+vy/N)}$$

The **Fourier Transform (FT)** is an important image processing tool which is used to decompose an image into its frequency components. The output of a Fourier Transform represents the image in the frequency domain, while the input image is the spatial domain (x, y) equivalent. In the frequency domain image, each point represents a particular frequency contained in the spatial domain image. So, for images with a lot of high-frequency components (edges, corners, and stripes), there will be a number of points in the frequency domain at high frequency values. **Zero frequency component (DC component) will be at top left corner**. DC components are shifted on the center for ease of visualization, obtaining the **frequency transform image**.

In practice, this is implemented by using the **fast Fourier Transform (FFT)** algorithm.

Convolution theorem

The Fourier Transform of the convolution of two functions is the product of their Fourier transforms:

$$F(g * h) = F(g)F(h)$$

The inverse Fourier Transform of the product of two Fourier Transforms is the convolution of the two Fourier transforms:

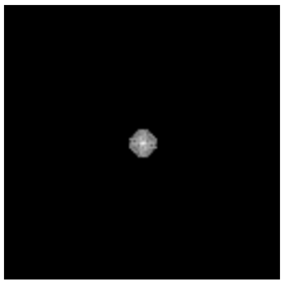
$$F^{-1}(gh) = F^{-1}(g) * F^{-1}(h)$$

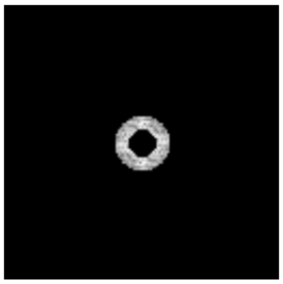
Convolution in spatial domain is equivalent to multiplication in frequency domain. Using the convolution theorem, we can implement all types of linear shift-invariant filtering as multiplication in the frequency domain (basically multiply the image with a mask).

Blurring in frequency domain

- Low frequencies are in the middle of the frequency transform image.
- High frequencies are far from the center of the frequency transform image.

Blurring an image involves attenuating or filtering out high-frequency components. The goal is to reduce image noise and/or to reduce the level of detail in the image. In the frequency domain, high-frequency components are far from the center of the image. To apply blurring in the frequency domain, a low-pass filter is needed. The low-pass filter attenuates high-frequency components and preserves low-frequency components. A simple example of a low-pass filter is the Gaussian filter, which has a bell-shaped response that attenuates high-frequency components. Another example is the square filter.

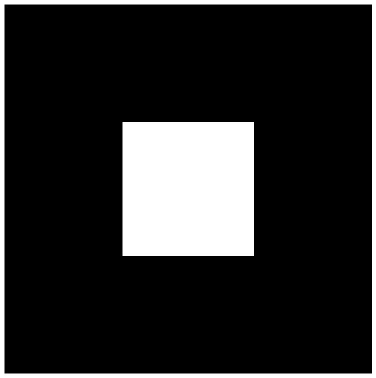

Low-pass


Band-pass

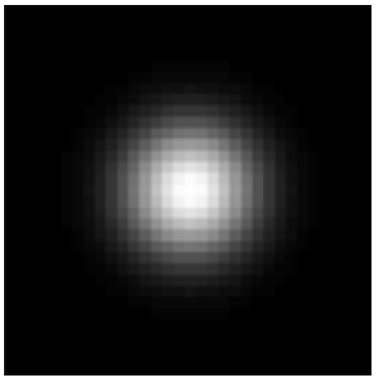
High-pass

Gaussian vs Box filter in frequency domain

Both the Gaussian and the Box filter are low-pass filters.



Box



Gaussian

In the frequency domain, the Fourier Transform of a Box filter is a sinc function, which has a main lobe with significant energy and high sidelobes that decay slowly. The sinc function has a narrow bandwidth, which means that it preserves high-frequency content.

On the other hand, the Fourier Transform of a Gaussian filter is also a Gaussian function that has a wider bandwidth than the sinc function of the Box filter. This means that the Gaussian filter attenuates high-frequency components more than the Box filter.

Compared to the Box filter, the **Gaussian filter** is more effective at **reducing high-frequency noise while preserving edges and details** in the image. This is because the Gaussian filter has a smooth, bell-shaped response that preserves more spatial information than the sharp, rectangular response of the Box filter. Furthermore the **box filter introduces ringing**. The ringing effect appears as oscillations or ripples around edges and sharp transitions in the image, which can make the image appear distorted or blurry. The sidelobes of the sinc function (Fourier Transform of the box filter) are responsible for the ringing effect because they spread the energy of the signal into neighboring frequencies, causing oscillations around sharp transitions.

Ideal Low Pass Filter (ILPF)

An **ideal low-pass filter (ILPF)** is defined by:

$$H(u,v) = \begin{cases} 1 if D(u,v) \leq D_0 \\ 0 if D(u,v) > D_0 \end{cases}$$

The point of transition between $H(u,v) = 1$ and $H(u,v) = 0$ is called **cutoff frequency**.

An ideal low-pass filter is a theoretical filter that completely eliminates all frequencies above a certain cutoff frequency while allowing all frequencies below that cutoff to pass through unaffected.

In an ideal low-pass filter, there is no attenuation of the desired frequencies below the cutoff frequency, and no distortion or delay of the signals passing through the filter. Additionally, the filter has an infinite roll-off, meaning that it provides complete attenuation of all frequencies above the cutoff frequency with no ripple in the passband.

It is impossible to build a real-world low-pass filter for various reasons:

- The components used in the construction of a filter, such as resistors, capacitors, and inductors, have inherent limitations in their performance characteristics. For example, capacitors have a limited frequency response, while inductors have resistance, which can limit their effectiveness at high frequencies.
- The signal processing techniques used to implement a low-pass filter, such as digital signal processing or analog signal processing, are subject to limitations in their precision and accuracy. These limitations can result in errors in the filtering process, which can affect the quality of the output signal.

But while this thing applies to analog components, why can't we simulate it with an algorithm?

- An ideal low pass filter has an **Infinite Impulse Response (IIR)**, which means that it requires an infinite number of computations to be implemented. In practice, only **Finite Impulse Response (FIR)** filters can be implemented, which have a limited number of coefficients and are therefore unable to achieve the ideal low pass filter response.

While **an ideal low-pass filter cannot be achieved in practice**, various types of real-world low-pass filters, such as Butterworth, Chebyshev, and Bessel filters, are designed to approximate the ideal response as closely as possible within certain design constraints.

Invariant local features

In image processing, **local features** refer to distinctive patterns or structures in an image that can be used to identify and describe specific regions of the image. Local features are typically extracted from an image by **analyzing small regions of the image, rather than the image as a whole**. This approach is known as **local analysis**.

We obviously want to find features that are invariant to transformations:

- **geometric invariance**: translation, rotation, scale, ...
- **photometric invariance**: brightness, exposure, ...

Examples of local features include:

- **Corner features** are points where the image intensity changes in multiple directions;
- **Blob features** are regions of the image with a distinct shape or texture;
- **Edge features** are areas where the intensity changes abruptly in one direction;
- **Texture features** describe repeating patterns or structures in the image;

Once local features have been identified and extracted (**feature extraction**) from an image, they can be used to create a feature vector that describes the image (**feature description**). This feature vector can then be used for tasks such as object recognition or image matching, by comparing the features of different images and identifying those that are most similar.

Why extract features?

- **Panorama stitching**: combine two or more images;
- **Visual SLAM**: constructing and updating a map of an unknown environment while simultaneously keeping track of an agent's location in it;
- **Image matching**;

Corner Detection

Corners are points in an image where the intensity or color changes significantly in multiple directions, and they are often associated with the intersection of edges or contours in an image. By detecting these corners, we can identify the locations of objects and their boundaries in an image, as well as extract important information.

Harris Corner Detector - Mathematical Formulation

A corner is a point whose local neighborhood is characterized by large intensity variation in multiple directions, and they are considered as important features of an image since **they are scale invariant**. They are often used in various computer vision and image processing tasks such as object recognition, image registration, and image matching. **Harris Corner Detector is a frequently used technique** in computer vision and one of the first takes to tackle the corner detection problem. For more information on the Harris Corner Detector (emphasis on the relation between eigenvalues and the ellipsoid containing the points) you can check here.

Consider a two-dimensional image I and a patch W of size $m \times m$ centered in (x_0, y_0). We want to evaluate the intensity variation occurred if the window W is shifted by a small amount (u, v). Such variation can be estimated by computing the **Sum of Squared Differences (SSD)**:

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

Where $g(x, y)$ is a window function that only enables the pixels in the window. We need to maximize the function $SSD(u, v)$ for corner detection, thus finding the windows (and their centers) in which the intensity variation is maximum. Since u and v are small, the shifted intensity $I(x + u, y + v)$ can be approximated by the first-order Taylor expansion:

$$I(x+u, y+v) \approx I(x,y) + uI_x(x,y) + vI_y(x,y)$$

Where I_x and I_y are partial derivatives of I in x and y direction respectively. We obtain:

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

$$\begin{aligned} SSD(u,v) &\approx \sum_{(x,y)\in W} [u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2] \\ &\approx Au^2 + 2Buv + Cv^2 \end{aligned}$$

With $A = \sum_{(x,y)\in W} I_x^2$, $B = \sum_{(x,y)\in W} I_xI_y$ and $C = \sum_{(x,y)\in W} I_y^2$.

This equation can be expressed in the following matrix form:

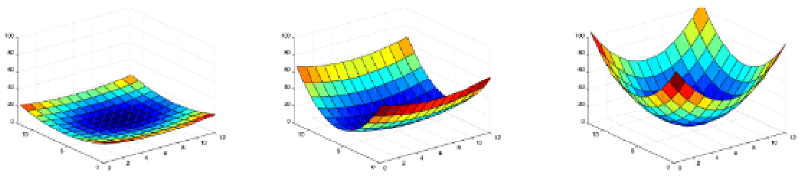
$$SSD(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

Where M is a 2×2 matrix computed from image derivatives:

$$M = \sum_{(x,y)} g(x,y) \begin{bmatrix} I_xI_x & I_xI_y \\ I_xI_y & I_yI_y \end{bmatrix}$$

This matrix is called **structure tensor** or **second moment matrix**.

The surface $E(u,v)$ is locally approximated by a quadratic form



From left to right, the error surface indicates a flat area, an edge and a corner.

The Harris detector uses the following response function (**Harris response function**) that scores the presence of a corner within the patch:

$$R = det(M) - k\ tr(M)^2$$

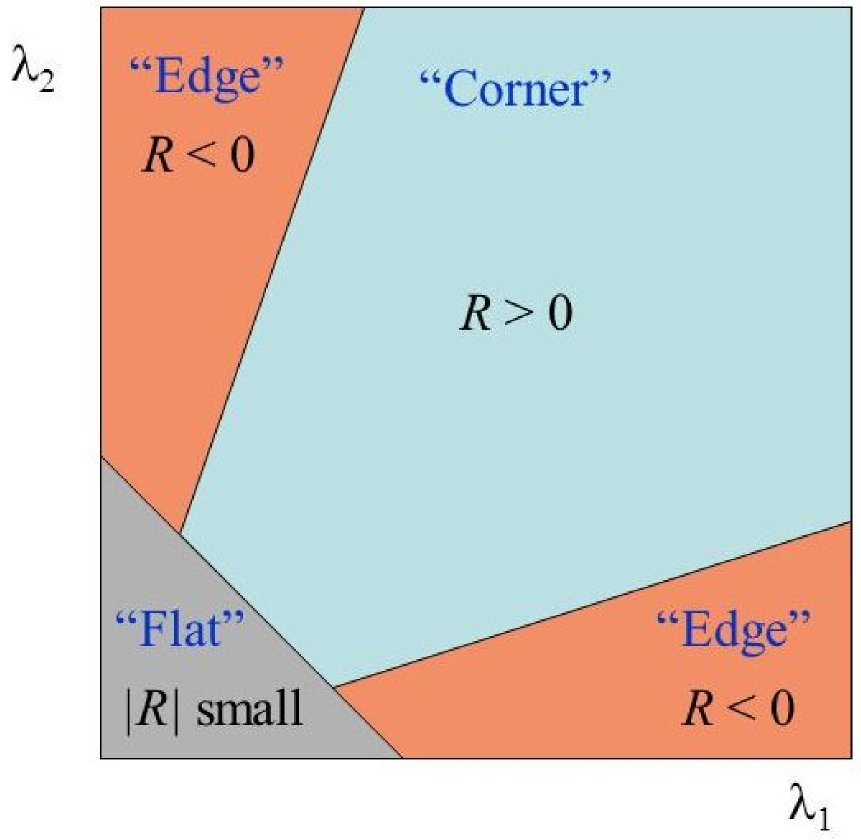
k is an empirical constant between 0.04 and 0.06 that let the eigenvalues be bigger than one. Since M is a symmetric matrix, $det(M) = \lambda_1\lambda_2$ and $tr(M) = \lambda_1 + \lambda_2$ where λ_1 and λ_2 are the eigenvalues of M. Hence, we can express the corner response as a function of the eigenvalues of the structure tensor:

$$R = \lambda_1\lambda_2 - k\ (\lambda_1 + \lambda_2)^2$$

So the eigenvalues determine whether a region is an edge, a corner or a flat:

- if λ_1 and λ_2 are small, then $|R|$ is small and the region is flat;
- if $\lambda_1 >> \lambda_2$ or $\lambda_1 << \lambda_2$, then $|R| < 0$ and the region is an edge;
- if $\lambda_1 \approx \lambda_2$ and both eigenvalues are large, then R is large and the region is a corner;

The classification of the points using the eigenvalues of the structure tensor is represented in the following figure:



Harris Corner Detector - Algorithm

The Harris Corner Detector algorithm consists of the following steps:

0. **Preprocessing**: convert the original image into a grayscale image I. The pixel values of I are computed as a weighted sum of the corresponding R, G, B values:
$$I = 0.299R + 0.587G + 0.114B$$

1. **Compute the derivatives I_x and I_y** by convolving the image I with a first derivative kernel, like the Sobel operator:
$$I_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * I$$
$$I_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I$$

   Another way is to convolve with the derivative of the Gaussian to have better results.

2. **[Optional] Subtract the mean from each image gradient** to eliminate the influence of the global intensity changes in the image.

3. **Compute the products of the derivatives** I_xI_x, I_xI_y, I_yI_y;

4. **Compute the sums of the products of derivatives** at each pixel: the next step is to sum the previously computed products over a small window around each pixel. This is done to obtain a measure of the local structure of the image. We obtain $S_{x^2}(x, y)$, $S_{xy}(x, y) = S_{xy}(x, y)$ and $S_{y^2}(x, y)$.

5. **Define the structure tensor (second moment matrix)** in that pixel:
$$M(x, y) = \begin{bmatrix} S_{x^2}(x, y) & S_{xy}(x, y) \\ S_{xy}(x, y) & S_{y^2}(x, y) \end{bmatrix}$$

6. Compute the Harris response in that pixel:
$$R = det(M) - k\ tr(M)^2 = \lambda_1\lambda_2 - k\ (\lambda_1 + \lambda_2)^2$$

7. Compute eigenvectors and eigenvalues;

8. Use threshold on eigenvalues to detect corners;

Eigenvalues and Eigenvectors computation

The eigenvectors of a matrix M are the vectors e that satisfy:

$$M\ e = \lambda\ e$$

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

- $e \rightarrow$ eigenvector
- $\lambda \rightarrow$ eigenvalue corresponding to eigenvector e

The steps to compute eigenvectors and eigenvalues of a matrix M are the following:

- Compute the determinant of $M - \lambda I$. This returns the **characteristic polynomial** of the matrix;
- Find the roots of the characteristic polynomial: $det(M - \lambda I) = 0$. This returns the eigenvalues;
- For each eigenvalue, solve: $(M - \lambda I)e = 0$. This returns the eigenvectors.

Once you have the eigenvectors and the eigenvalues of a matrix M, you can rewrite M in the following way:

$$M = \begin{bmatrix} u_1 & u_2 & ... & u_d \end{bmatrix} M \begin{bmatrix} u_1 \\ u_2 \\ ... \\ u_d \end{bmatrix}$$

being M of size $d \times d$.

Trace of a matrix

The trace of a $d \times d$ matrix M is the sum of all the elements in its diagonal:

$$tr(M) = \sum_i^d a_{ii} = a_{1,1} + a_{2,2} + ... + a_{d,d}$$

Other Response Functions

We can use other response functions:

Harris & Stephens (1988):

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

Kanade & Tomasi (1994):

$$R = min(\lambda_1, \lambda_2)$$

Nobel (1998):

$$R = \frac{det(M)}{tr(M) + \epsilon}$$

Harris Corner Detector - Properties

We want to always find the same point on an object, regardless of changes in the image. To do so, the Corner detector should be insensitive to:

- scale;
- rotation;
- lighting;
- perspective imaging;
- partial occlusion;

The Harris Corner Detector is:

- **invariant to rotation**: the ellipse rotates but its shape given by the eigenvalues remains the same; corner response R is invariant to image rotation;
- **invariant to intensity changes** in the image:
  - invariance to intensity shift: shifting the intensity values of the image uniformly does not affect the gradients of the image. The Harris Corner Detector operates on the gradients of the image, so any constant shift in the intensity values will be cancelled out when computing the gradients. Therefore, the Harris Corner Detector will produce the same results regardless of any uniform shift in the image intensity values;
  - not fully invariant to intensity scale: because of the fixed intensity threshold we have only partial invariance on multiplicative intensity changes: it can detect false positives;

Harris Corner Detector is not invariant to scale.

In edge detection, the goal is to identify boundaries or transitions between regions of the image that have different pixel values or intensities. This is typically achieved by computing the gradient of the image, which represents the rate of change in pixel values or intensities in different directions. However, the gradient operator used in edge detection is typically based on a fixed kernel size or filter, such as the Sobel or Canny operators. This means that **the operator is tuned to detect edges at a specific scale or size of features in the image**. If the feature in the image are larger or smaller than the scale the operator is tuned to, the resulting edge or corner detection may not be accentuate or may miss important features.

Blobs

A blob can be defined as am amorphous group of connected pixels that share common properties, such as color or intensity. Blob detection algorithms aim to isolate and extract these regions from an image. Blob detection has several advantages:

- **Scale-invariant detection**: Blobs are detected at multiple scales. This allows us to identify blobs that may have different sizes or shapes in the image. In contrast, corner detection typically only works at a single scale and may miss features that are smaller or larger than the scale it is tuned to.
- **Robustness to noise**: Blob detection is often more robust to noise and variations in lighting conditions than corner detection. This is because blobs are typically larger regions of the image and can be more easily distinguished from the background noise. In contrast, corner detection is more sensitive to noise and may produce false positives or miss something in noisy images.

Scale Invariant Detection (Blob Detection)

We want to build a detection method that will identify features that are invariant to changes in orientation, illumination and **scale**, such as **blobs**. We talk about **blob detection**.

Methods such as blob detection are scale invariant because they detect features at multiple scales.

Intuitively, we can define a function f on a region that is scale invariant, i.e. that results in the same output if applied on a certain region even at different scales (for example, average intensity function is the same for corresponding regions, even at different sizes). Then we could simply take a local maximum of this function on different scales of the image. A good function will have one stable sharp peak in the key point.

Instead of computing f for larger and larger windows, we can implement an algorithm that will use a fixed size window over a **scale space**. The scale space refers to a series of images generated by applying Gaussian blurring to the original image at different scales. It is used to analyze the image at multiple levels of detail, allowing the detection of blobs at various sizes.

Blob detection algorithms often analyze the scale space by computing the difference of Gaussians (DoG) or Laplacian of Gaussian (LoG) images. The DoG image is obtained by subtracting two adjacent scales in the scale space, while the LoG image is generated by convolving the scale space with the Laplacian operator. These operations help identify regions with significant intensity changes across scales, which are indicative of potential blobs.

Scale space

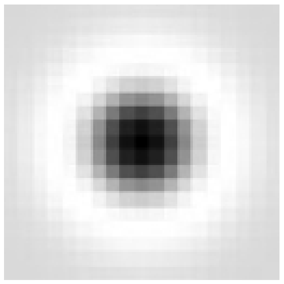
The first step is to build the scale space. The scale space refers to a series of images generated by applying Gaussian blurring to the original image at different scales. It is used to analyze the image at multiple levels of detail, allowing the detection of blobs at various sizes.

- **Define a range of scales**: determine the range of scales that identifies the desired range of blob sizes we want to detect;
- **Generate a Gaussian kernel for each scale**: for each scale compute the Gaussian function with the sigma value of the scale and discretize it into a matrix of appropriate dimensions. The size of the matrix (kernel) can be established using sigma.
- **Convolve the image with each Gaussian kernel**: convolve the original image with each Gaussian kernel to obtain a series of blurred images; store each of them in the scale space.

Once the scale space is constructed, it can be further analyzed using techniques like the difference of Gaussians (DoG) or Laplacian of Gaussian (LoG) to detect and localize blobs at different scales.

Laplacian of Gaussian (LoG)

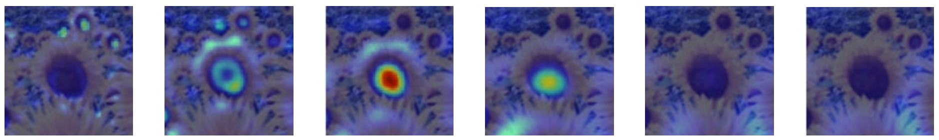
A common definition for f is the **Laplacian of Gaussian (LoG)**, that is the second derivative of the Gaussian filter.



$$\nabla^2 g = \frac{\delta^2 g}{\delta x^2} + \frac{\delta^2 g}{\delta y^2}$$

When applying a scale-invariant filter like the Laplacian of Gaussian (LoG) to an image, **the maximum response is typically obtained at the scale that best matches the size of the feature being detected** (the zero of the Laplacian has to be aligned with the blob).

For example, if the feature being detected is a small point, the maximum response will be obtained at a small scale (i.e., a small value of the filter scale parameter). If the feature being detected is a larger object, the maximum response will be obtained at a larger scale (i.e., a larger value of the filter scale parameter).



$\sigma = 2.1$ $\sigma = 4.2$ $\sigma = 6.0$ $\sigma = 9.8$ $\sigma = 15.5$ $\sigma = 17.0$

Here's how the LoG-based blob detection process typically proceeds within the scale space:

For each scale in the scale space:

- Determine the standard deviation (sigma) value corresponding to the current scale;
- Compute the kernel size based on the relationship between kernel size and sigma. The size of the kernel is determined to capture the desired blurring effect at the current scale;

- Generate the LoG kernel using the calculated kernel size.
- Convolve the current scale image with the LoG kernel to obtain the LoG response image, which represents the Laplacian of the blurred image at that scale. This image will have some local maxima based on some threshold.

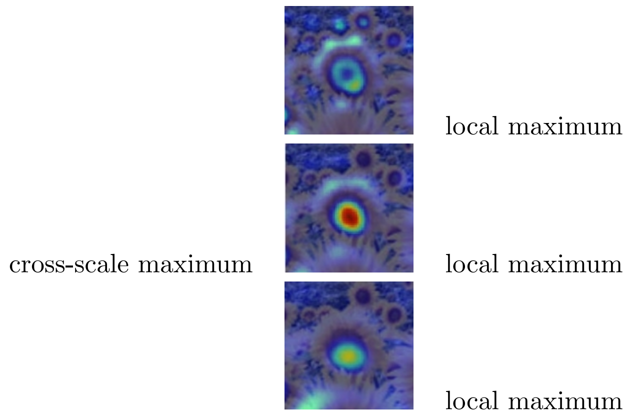
When using the Laplacian of Gaussian (LoG) for blob detection with the scale space, the kernel size does not remain the same for each image in the scale space. The kernel size varies across different scales to correspond to the appropriate blurring level.

Once we have all the LoG response images, each with some local maxima, for each feature compute the cross-scale maxima. Save its coordinates along with the scale (x, y, σ) .

Implementation

Pseudocode:

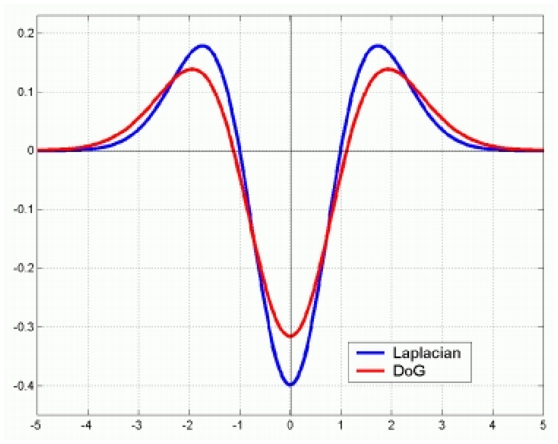
```
for each level of the scale space:
    compute the feature response (e.g. Laplacian)

# now, for each level (scale) we have a set of features

for each level of the scale space:
    if one feature is the local maximum (filter highlights blob in that feature)
        and cross-scale maximum (maximum filter response across levels):
            save scale and location of the feature (x,y,s)
```



For more information about the algorithm and a python implementation, you can read here.

Difference of Gaussians (DoG)

The LoG is very similar to a Difference of Gaussians (DoG), that is a Gaussian minus a slightly smaller Gaussian:



Gaussian:

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

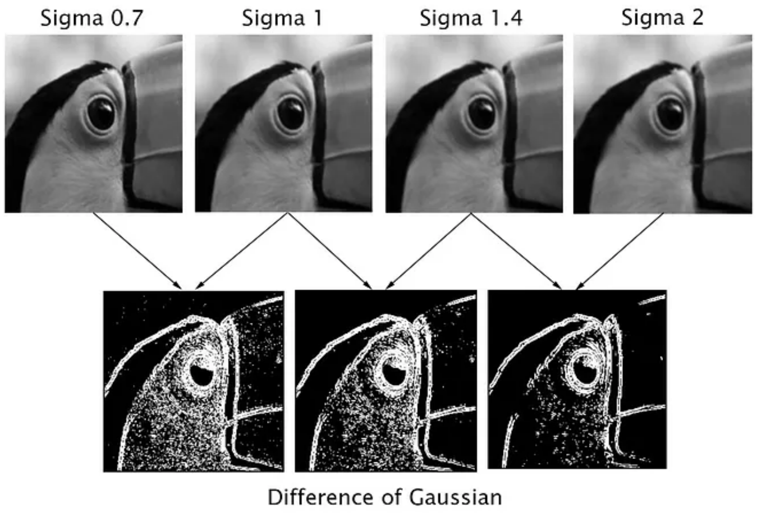
Laplacian of Gaussian:

$$LoG = \nabla^2 g = \frac{\delta^2 g}{\delta x^2} + \frac{\delta^2 g}{\delta y^2}$$

Difference of Gaussians:

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

Since we have to use a Gaussian scale space it is very convenient to use the DoG instead of the LoG: the response function can simply be the difference from a window on one layer of the pyramid and the same window on the upper layer. More information about the DoG algorithm can be found here



The Difference of Gaussian technique evolved from the Laplacian of Gaussian (LoG) technique. It turns out that subtracting Gaussian images from each other is a speedy estimation of the Laplacian of Gaussian. The Difference of Gaussian technique is a less compute intensive approximation of a Laplacian of Gaussian calculation.

Feature Descriptor

After an object or a region of interest is detected in an image, a feature descriptor is used to represent the detected object or region in a compact and distinctive way. This is necessary because raw pixel values of an image can be noisy, redundant, and not invariant to scale, orientation, and illumination changes, which can make object recognition and matching difficult.

A feature descriptor is a representation of an object or a region of interest in an image that captures its unique characteristics. Feature descriptors are designed to be robust to common image transformations, such as rotation, scale, and illumination changes. They are also designed to be efficient to compute and store.

Patches with similar content should have similar descriptors.

Naive Feature Descriptors

- **image patch**: just use the plain image patch (flattened to array) to describe the feature; this is perfectly fine if geometry and appearance remain unchanged (i.e. in **template matching**, the process of moving the template over the entire image and calculating the similarity between the template and the covered window on the image); The problem with this approach is that **small changes in the intensity values of the pixels within the patch can lead to significant changes in the feature descriptor**;
- **image gradients**: use the intensity variation instead of the intensity values; this way the feature is invariant to absolute intensity values; **How can it be less sensitive to deformations?**
- **color histogram**: count the colors in the image using a histogram; this is invariant to changes in scale and rotation; the problem with this approach is that **we loose spatial information**: two completely different patches with the same colors will have the same descriptor.
- **spatial histograms**: compute histograms over spatial cells; the descriptor divides the image into a grid of cells and then counts the number of occurrences of each visual feature in each cell. The resulting counts are organized into a histogram, where each bin corresponds to a range of values for the visual feature. This technique is partially invariant to rotation; **how can it be completely invariant to rotation?**
- **orientation normalization**: use the dominant image gradient direction to normalize the orientation of the patch; this is typically achieved by computing the dominant orientation of the image feature and then rotating the image patch; the problem with this approach is that orientation of

an image feature can be sensitive to changes in scale, which can lead to inconsistencies in the descriptor when comparing features at different scales.

Histogram of Oriented Gradients (HOG) Descriptor

For detailed steps and a python implementation, you can check this out. Here is a great video explaination of the HOG Descriptor.

0. **preprocessing**: preprocess the image and bring down the width to height ratio to 1:2. The image size should preferably be 64×128. This is because we will be dividing the image into 8×8 and 16×16 patches to extract the features. Having the specified size 64×128 will make the calculations pretty simple;

1. **image gradients**: the next step is to compute the gradients;

2. **magnitude and orientation**: once we have the gradients, we can compute magnitude and orientation for each pixel in the 8×8 patch;

3. **generate the histogram**: generate an histogram with magnitude and orientation for each 8×8 cell;

4. **normalize the gradients**: the gradients of the image are sensitive to the overall lighting. This means that for a particular picture, some portion of the image would be very bright as compared to the other portions. We cannot completely eliminate this from the image, but we can reduce this lighting variation by normalizing the gradients by taking 16×16 blocks. A 16×16 block has 4 histograms which can be concatenated to form a 36×1 element vector. This vector is normalized with **L2 norm**.

   Given a vector v:

   $$v = \begin{bmatrix} v_1 & v_2 & ... & v_n \end{bmatrix}$$

   Compute the L2 norm:

   $$k = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$$

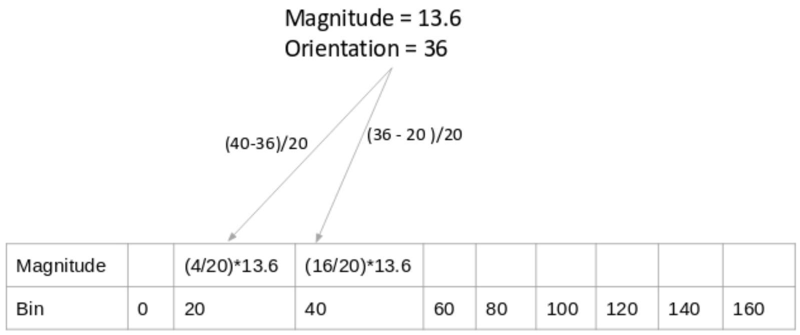
   Divide each component in the vector v by k:

   $$nv = \begin{bmatrix} \frac{v_1}{k} & \frac{v_2}{k} & ... \frac{v_n}{k} \end{bmatrix}$$

5. **shift the window**: the window is then moved by 8 pixels and a normalized 36×1 vector is calculated, the process is repeated. The feature is the centroid of the 16×16 block.

Histogram generation

The naive way to generate the histogram is to divide the $0 - 180$ spectrum (unsigned) into bins (we will use 9 bins, each spanning 20 degrees) and to count the occurrences of the orientation. **This does not take into account the magnitude**. Instead of using the frequency, we can use the gradient magnitude to fill the values in the matrix:



Magnitude		(4/20)*13.6	(16/20)*13.6						
Bin	0	20	40	60	80	100	120	140	160

Scale invariance

The HOG descriptor is partially invariant to scale. The HOG descriptor is based on the distribution of gradients in an image, and the gradients are computed over small image patches.

When the image is scaled up or down, the size of the patches also changes accordingly. However, the HOG descriptor compensates for this scale change by normalizing the histogram bins for each patch based on the overall gradient energy within that patch. This normalization helps to ensure that the descriptor is partially invariant to changes in scale.

However, **if the scale change is too significant, the HOG descriptor may not perform well**. For example, **if an object is very small in the image, its HOG descriptor may not be distinctive enough for reliable object recognition**. Similarly, **if an object is very large in the image, its HOG descriptor may become too coarse, resulting in loss of fine-grained details**. Therefore, it is important to carefully choose the scale of the image and the size of the patches when using the HOG descriptor for object recognition.

Scale Invariant Feature Transform (SIFT)

Scale Invariant Feature Transform (SIFT) **consist of both a detector and a descriptor**.

Feature detector:

- **Multi-scale extrema (keypoint) detection**;
- **Keypoint localization**;
- **Orientation assignment**;

Feature descriptor:

- **Keypoint descriptor**;

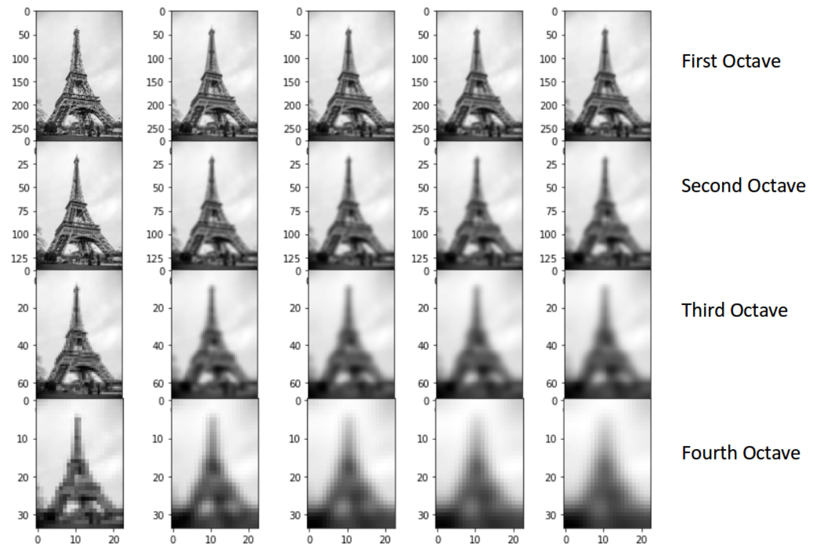
The idea is that proper scaling of objects in new image is unknown; exploring features in different scales is helpful to recognize different objects.

For more information about the SIFT algorithm you can read here.

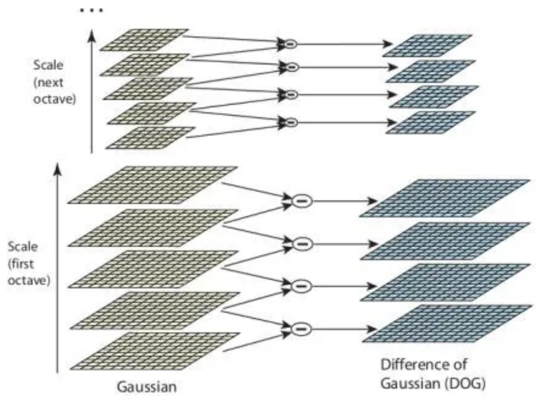
SIFT Detector - Keypoint detection

The first step of the SIFT algorithm is to detect keypoints in an image. Keypoints are distinctive features in an image that can be used to identify the same object in different images.

An octave is a set of images of the same size blurred at different scales. Each octave's image size is half the previous one.

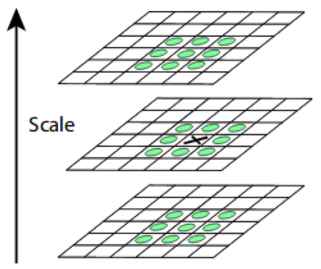


Now we use those blurred images to generate another set of images, the Difference of Gaussians (DoG). These DoG images are great for finding out interesting keypoints in the image. The difference of Gaussian is obtained as the difference of Gaussian blurring of an image with two different σ, let it be σ and $k\sigma$. This process is done for different downsampling layers. This is what the process looks like:



SIFT Detector - Keypoint localization

We have generated a scale space and used the scale space to calculate the Difference of Gaussians. Those are used to approximate Laplacian of Gaussian (LoG).



One pixel in an image is compared with its 8 neighbors as well as 9 pixels in the next scale and 9 pixels in previous scales. This way, a total of 26 checks are made. If it is a local extrema, it is a potential keypoint. It basically means that keypoint is best represented in that scale (the scale of the considered blurred image in the DoG set σ). The keypoint is thus identified by $\mathbf{x} = \{x, y, \sigma\}$.

SIFT Detector - Keypoint localization - Keypoint stability

The previous step produces a lot of keypoints. Some of them lie along an edge, or they don't have enough contrast. In both cases, they are not as useful as features, so we get rid of them.

Removing low contrast features: if the magnitude of the intensity of the blurred image at the current pixel in the DoG is less than a certain value, then the pixel is rejected (typically a pixel with a magnitude intensity that is less than 0.1 the max value). This is done because low contrast is generally less reliable than high contrast for feature points.

SIFT Detector - Orientation assignment

The next step consists in assigning an orientation to each keypoint; this provides rotation invariance.

For all the levels:

- **compute the magnitude and orientation**: a neighborhood is taken around the keypoint location (its size depends on the scale of the level), and the gradient magnitude and direction is calculated in that region;
- **compute the weighted magnitude + orientation histogram**: an oriented histogram with 36 bins covering 360 degrees is created; the amount of the orientation added to a certain bin is proportional to the magnitude of the gradient at that point;
- **find the peak of the histogram**: the bin of the highest peak is used for the orientation of the keypoint.

  Furthermore, any peaks above 80% of the highest peak are converted into a new keypoint; This new keypoint has the same location and scale as the original, but its orientation is different.

SIFT Descriptor - Keypoint description

At this point, each keypoint has a location, scale, orientation. Next is to compute a descriptor for the local image region about each keypoint that is highly distinctive and invariant as possible to variations such as changes in viewpoint and illumination.

The steps of building the SIFT descriptor are as following:

- use the **Gaussian blurred image** associated with the keypoint's scale;
- take **image gradients over a** 16×16 **square window** around the detected feature;
- **rotate the gradient** directions AND locations relative to the keypoint orientation (given by the dominant orientation);
- **divide the 16x16 window into a** 4×4 **grid of cells**;
- **compute an orientation histogram** with 8 orientations bins for each cell bins (summing the weighted gradient magnitude);
- the **resulting SIFT descriptor is a length** 128 **vector** representing a 4 histogram array with 8 orientation bins per histogram;

SIFT Descriptor Properties

- **invariant to rotation** because we rotated the gradients; we are assuming the rotated image will generate a key point at the same location as the original image;
- **invariant to scale** because we worked with the scaled image from DoG;
- **invariant/robustness to illumination variation** since we worked with the orientation and we don't take in consideration the magnitude of the gradient (alone);
- **slight robustness to affine transformation and to noise** (empirically found);

Keypoint matching

Keypoint matching is a technique used in computer vision to identify and match the corresponding keypoints in multiple images. The process involves identifying keypoints in multiple images and then finding the corresponding keypoints between them.

- **Find all keypoints** in a target image. Each keypoint will have 2D location, scale and orientation, as well as invariant descriptor vector (x,y,s,theta,d). This can be done using various algorithms such as SIFT, SURF, ORB, and AKAZE.
- **For each keypoint, search similar descriptor vectors** in the other image(s).
- Once the keypoints are matched, **they can be used to compute the transformation between the two images**.

One problem is that **the descriptor vector may match more than one reference in the database**.

Feature matching

Given a feature in I_1, how to find the best match in I_2?

1. Define distance function that compares two descriptors;
   - Simple approach: $L2$ norm.

     $$d = |f_1 - f_2|$$

     Can give small distances for ambiguous (incorrect) matches;
   - Better approach: distance ratio.

     $$d = \frac{|f_1 - f_2|}{|f_1 - f_2'|}$$

     f_2 is the best match to f_1 in I_2
     f_2' is the second best match to f_1 in I_2

     To further improve this method we can **set the distance ratio threshold (ρ) to around** 0.5, which means that we require our best match to be at least twice as close as our second best match to our initial features descriptor.
2. Test all the features in I_2, find the one that minimizes the distance function;

Homografy

Homography refers to a mathematical transformation that relates the coordinates of points on a plane to the coordinates of those same points when viewed from a different perspective or angle.

In computer vision and image processing, homography is commonly used to correct distortions caused by perspective projection, such as those that occur when viewing a three-dimensional object from different angles or when taking photographs with a camera that is not aligned perfectly with the scene being photographed.

A homography can be represented by a 3×3 matrix and is typically computed by finding correspondences between points in two different images or frames. Once a homography has been computed, it can be used to warp or transform one image to match the perspective of the other image, or to estimate the pose or position of a camera relative to a scene.

RAndom SAmple Consensus (RANSAC)

RANSAC, which stands for **RAndom SAmple Consensus**, is a robust algorithm commonly used in computer vision and machine learning to estimate parameters of a mathematical model from a set of observed data that contains outliers or noise.

The basic idea behind RANSAC is to randomly sample a subset of the data, fit a model to the sample, and then use the model to classify the remaining data as inliers or outliers. This process is repeated multiple times, and the model with the largest number of inliers is selected as the final estimate of the parameters.

RANSAC can be applied to a wide range of problems, such as estimating the parameters of a homography, detecting and removing outliers in data, or fitting a line or curve to a set of noisy points.

One of the key advantages of RANSAC is its ability to handle outliers and noise in the data, which can cause traditional methods to fail or produce unreliable results. However, **RANSAC does have some limitations**, such as the need to choose appropriate parameters for the algorithm, and the potential for the algorithm to get stuck in local optima or to converge to suboptimal solutions.

The main steps are the following:

- randomly choose s samples; Typically s is the minimum sample size that lets you fit a model;
- fit a model (e.g. line) to those samples;
- count the number of inliers that approximately fit the model;
- repeat n times;
- choose the model that has the largest set of inliers;

Each hypothesis gets voted on by each data point; the best hypothesis wins.

Image alignment and Image stitching

Image alignment and image stitching are two related concepts in computer vision that are used to combine multiple images into a larger panorama or composite image. While they are often used together, they refer to different aspects of the image processing pipeline.

Image alignment

Image alignment refers to the process of aligning two or more images so that their content overlaps as closely as possible. This is typically necessary because images that are taken from different viewpoints or at different times may have different orientations, scales, and perspectives.

Image alignment techniques typically involve:

- **Compute image features for image A and image B**;
- **Match features between A and B**;
- **Compute a transformation that maps one image to the other (homography)**; this can be done using **RANSAC**;

Image stitching

Image stitching refers to the process of combining multiple aligned images into a single composite image. This typically involves blending the overlapping regions of the images together to create a seamless transition, and possibly correcting for distortions and artifacts that may arise during the alignment and blending process. Image stitching is often used in panoramic photography, where a series of images taken from a single viewpoint are stitched together to create a wide-angle view of a scene.

Transformations

A transformation refers to the process of modifying the appearance or geometry of an image to achieve a desired effect. **A transformation T is a coordinate-changing machine**.

There are various types of transformations that can be applied to images, including geometric transformations such as scaling, rotation, translation, and perspective transformation, as well as non-geometric transformations such as brightness adjustment, contrast adjustment, color space conversion, and filtering.

- **Geometric transformations involve modifying the shape or position of an image or its parts**. For example, rotation can be used to change the orientation of an image, scaling can be used to change its size, and translation can be used to shift its position. Perspective transformation can be used to correct for distortion due to camera angle or lens distortion.

- **Non-geometric transformations modify the visual appearance of an image without changing its shape or position**. For example, brightness and contrast adjustment can be used to enhance the visibility of an image, while color space conversion can be used to change the color representation of an image. Filtering operations can be used to smooth or sharpen an image or to remove noise or other artifacts.

A geometric transformation can be global or non-global:

- A **global transformation** is the same for any point of the image;
- A **non-global transformation** is instead applied locally to specific regions or parts of an image;

Let's focus on global transformations. A global transformation can be:

- **linear** if it can be described by a 2×2 matrix;

$$p' = Tp$$

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

- **non-linear**

Linear transformations

Common linear transformations include:

- **Uniform scaling**:
$$S = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$$
- **Rotation by angle θ**:
$$R = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$
The inverse of which is $R^{-1} = R^T$
- **Mirror about y axis**:
$$T = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$
- **Mirror about x axis**:
$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Translation is not a linear transformation on 2D coordinates.

Affine transformations

Any transformation represented by a 3×3 matrix which has $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ as last row, is called an **affine transformation**. We can express the previous transformations in terms of 3×3 matrices, as well as translation.

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

$$Translate$$

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

$$Scale$$

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

$$Rotate$$

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

$$Shear/Deform$$

Affine transformations can be sees as:

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

Homografies/Planar Perspective Maps/Projective transformations can be sees as:

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