DIGITAL CULTURE
ITMO UNIVERSITY

Image Comparison: Local Features

High School of Digital Culture
ITMO University
dc@itmo.ru

Contents

1 Image features and similarity: traditional computer vision

1.1 Local features, keypoints

Global and local features - recap We looked at several examples of global features for color and texture. There exist many more different approaches to describe color and texture. Also, there have been proposed many shape descriptors. But in recent years with rise of deep learning less and less of those features are used in practical applications.

In the beginning of our discussion about image features, I've mentioned that they can be divided into global and local. Both global and local features can represent a particular characteristic of an image - color, texture, shapes, edge orientations. But global features are derived from the whole image and local features describe a particular neighborhood, only part of an image. Let's now look at why local features are important and learn a couple of most prominent examples of local features.

When global features are not good enough There are many cases when global features are not reliable enough for image matching. For example, when there are two images of the same object but at different scale, with global features it is hard to match a region in one image to another image. If images are taken from different view points, with different lighting conditions, with occlusions - we need to be able to match local image patches of one image to local image patches of another.

Local features to the rescue With local features we can match parts of images even when global features don't match. Ok, it might be a good idea to match image fragments instead of whole images. But which fragments should be compared? How do we choose right most informative fragments? And also what's the best representation for those fragments? We need to find a way to decide which image fragment to pick, and then how to build feature vectors representing those fragments.

How to choose and match fragments? So, how can we match fragments from different images? Let's say we want to identify that an object on this second smaller image matches an object in the back on the first image. A naive way is to scan both images with sliding windows of variable sizes and compare every possible pair of windows. This will probably work - we will find matching fragments as we will go through every possible pair of fragments. But this approach is too slow -

just imagine how may possible positions and sizes of a sliding window we'll have to check! And we'll need to compare every possible fragment on one image to every possible fragment on another. This approach is prohibitively expensive.

What if we choose only a subset of fragments from both images and do sparse matching? This will obviously be faster! We can choose the fragments in such a way that they are isolated, so there is no intersection between different fragments from the same image. This sounds good. But since we don't compare all possible fragments, we need to make sure that we choose the most important and representative ones. How to do that? How can we find image locations where we can reliably find correspondences with other images? We need to find locations that are likely to match well in other images. Textureless patches are nearly impossible to localize, so they are not good candidates here. Patches with large contrast changes are easier to localize. We need specific locations in the images, such as specific characters, mountain peaks, building corners - non-uniform, special, informative points. These kinds of points are often called keypoints or interest points.

Characteristics of good keypoints What are some important properties of good keypoints? First, there should not be too many of them. We want many fewer keypoints than image pixels.

Next, each keypoint should be distinctive, unique. If there are many very similar locations, it will be hard to find proper matches across two images. As in this example, it's impossible to say which one among the three red points on the right image is a proper match to a red point on th left image.

Good keypoints are local. A keypoint should occupy a small area of the image to be robust to clutter and occlusion.

And last, keypoints should be repeatable. We need the same keypoint to be found in different images despite geometric and photometric transformations. If there are no matching keypoints, it won't be possible to match images. We need at least a subset of keypoints from one image to match a subset of keypoints on another. And keypoint detection should happen independently in two images. That means that our keypoint detection algorithm should repeatedly detect the same keypoints when applied to the same image, and when applied to different images.

Applications Keypoints detection and matching are an essential component of many computer vision applications till these days. Imagine that we would like to align the two images so that they can be seamlessly stitched into a panorama. To do that automatically we need to detect keypoints on all the images that we want to stitch together, find matching locations and connect properly.

Another important application is a 3D reconstruction from 2D images. With keypoints we can establish a dense set of correspondences so that a 3D model can be constructed or an in-between view can be generated.

Many motion tracking algorithms use keypoints. By detecting and matching keypoints on subsequent frames one can detect and track motion.

Keypoints are also used in robotics for robot navigation. Local features based on keypoints are still used in some image retrieval engines. They also was heavily used for object detection and recognition just a decade ago. But now most of object detection applications (and many image retrieval ones) leveraging advances of deep learning.

Image matching with keypoints and local features Image matching with local features can be divided into three separate stages. The first stage is keypoint detection. During this stage each image is searched for locations that are likely to match well in other images. The second stage is feature description. At this stage regions around every detected keypoint location are converted into more compact descriptors that can be matched against other descriptors. And third is the feature matching stage, when an algorithm searches for likely matching candidates in two images.

Let's look at every of these stages one by one, starting from the keypoint detection.

Corners as distinctive interest points Intuitively, we think of a corner as a rapid change of direction in a boundary curve. Corners are highly effective features because they are distinctive and reasonably invariant to viewpoint. Because of these characteristics, corners are frequently used as keypoints. There have been proposed several algorithms for corner detection. The basic idea in most of them is to run a small window over an image and compute intensity changes when window is slightly shifted.

If the pixel is in a region of uniform intensity, then the nearby positions of a window will look similar. If the pixel is on an edge, then nearby shifting the window in a direction perpendicular to the edge will look quite different, but shifting the window in a direction parallel to the edge will result in only a small change. If the window is over a corner, then shifting is in any direction will produce a large change in intensity.

One of the earliest corner detection algorithms, Moravec corner detection algorithm, is based on computing the sum of squared differences between two neighbor positions of the window. Harris and Stephens improved upon Moravec's corner detector by considering image gradients at every point, instead of using shifted patches. Essentially, they took this simple idea of changed intensities in a small neighborhood to a mathematical form.

Harris detector: mathematics . Harris Corner Detector basically finds the difference in intensity for a window when it is shifted by (u, v). Let $I(x, y)$ denote an intensity of an image I in point (x, y). Consider taking a window over the area (x, y) and shifting it by (u, v). The weighted sum of squared differences (WSSD) between these two windows is given by:

$$E(u, v) = \sum_{x,y} w(x, y) \left[I(x + u, y + v) - I(x, y)\right]^2 \tag{1}$$

Here $w(x, y)$ is a window (or weighting) function which gives weights to pixels underneath. It is usually either a rectangular window, when it is 1 inside the window and 0 elsewhere, or a Gaussian window. The rectangular window is used when computational speed is important and the noise level is low. The Gaussian window is used when data smoothing is important.

$I(, y)$ is intensity of the original points, and $I(x + u, y + v)$ is intensity of shifted points.

For nearly constant regions, where nearby windows don't change, $E(u, v)$ will be near 0. For regions where intensities are very different for slightly shifted windows, $E(u, v)$ will be larger. Hence, we want windows where $E(u, v)$ is large.

For small (u, v), $I(x+u, y+v)$ can be approximated by a Taylor expansion. Let I_x and I_y be the partial derivatives if I, such that

$$I(x + u, y + v) \approx I(x, y) + I_x(x, y)u + I_y(x, y)v \tag{2}$$

This produces an approximation for $E(u, v)$:

$$E(u, v) \approx \sum_{x,y} w(x, y) \left[I_x(x, y)u + I_y(x, y)v\right]^2 \tag{3}$$

And this equation can be written in matrix form as:

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{4}$$

where

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{5}$$

Matrix M sometimes is called the Harris matrix.

What does this matrix reveal? How can it be used to tell a presence of a flat region, an edge or a corner? Let's first consider an axis-aligned corner. So the only changes in intensity are along x and y axis. This means that product of partial derivatives $I_x I_y$ is zero for all positions (x, y). And matrix M looks like

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & 0 \\ 0 & I_y^2 \end{bmatrix} = \sum_{x,y} w(x,y) \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \tag{6}$$

Since we have 0 values everywhere besides the main diagonal, this means that the main diagonal contains eigenvalues of that matrix.

Corner-like locations in an image will be characterized by large values of both eigenvalues - this means that intensity is changing along both directions, x, and y. If either eigenvalue is close to 0, this means that this location is not corner-like.

In this case we assumed that a corner is axis-aligned. But what if we have a corner that is not aligned with the image axes?

Since M is symmetric, we can represent it as follows:

$$M = \sum_{x,y} w(x,y) X \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} X^T \tag{7}$$

Since M is a real symmetric matrix, its eigenvectors point in the direction of maximum data spread, and the corresponding eigenvalues are proportional to the amount of data spread in the direction of the eigenvectors. So, the eigenvalues reveal the amount of intensity change in the two principal orthogonal gradient directions in the window.

Harris detector: classification via eigenvalues Now we can analyze the eigenvalues of M to detect corners. A corner (or in general an interest point) is characterized by a large variation of E in all directions of the vector $\begin{pmatrix} u & v \end{pmatrix}$, which corresponds to larger magnitudes of eigenvalues of M. M should have two "large" eigenvalues for an interest point. Based on the magnitudes of the eigenvalues, the following inferences can be made:

- If both eigenvalues λ_1 and λ_2 are close to 0, then E is almost constant in all directions, and this means that pixel (x, y) has no features of interest and belongs to a flat homogeneous region.
- If one of the eigenvalues has some large positive value, and another is close to 0, or when one of the eigenvalues is much larger than another ($\lambda_1 >> \lambda_2$ or $\lambda_2 >> \lambda_2$) then an edge is found.
- And finally when both eigenvalues have large positive values, none of them is significantly larger than another, this means that E is large in all directions, and a corner is found.

Harris detector: corner response measure Thus, we see that the eigenvalues of the matrix formed from derivatives in the image patch can be used to differentiate between flat regions, edges and corners. But Harris and Stephens noted that exact computation of the eigenvalues is computationally expensive. So instead they suggested to use a following function as a measure of corner response:

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2 = \det M - k(\mathrm{trace} M)^2 \tag{8}$$

It is based on the fact that the trace of a square matrix is equal to the sum of its eigenvalues, and its determinant is equal to the product of its eigenvalues. So we don't have to actually compute the eigenvalue decomposition of the matrix$M$ and instead it is sufficient to evaluate the determinant and trace of M to find corners, or rather interest points in general.

k here is an empirical constant. Its range of values depends on the implementation. In the literature the values in the range $0.04\breve{}0.15$ have been reported as feasible. You can interpret k as a "sensitivity factor;" the smaller it is, the more likely the detector is to find corners.

Measure R has large positive values when both eigenvalues are large, indicating the presence of a corner; it has large negative values when one eigenvalue is large and the other small, indicating an edge; and its absolute value is small when both eigenvalues are small, indicating that the image patch under consideration is flat. Typically, R is used with a threshold, T. We say that a corner at an image location has been detected only if $R > T$ for a patch at that location.

The Harris corner detector This slide outlines all the steps of the Harris corner detector.

1. Compute the horizontal and vertical derivatives of the image$_I$x and I_y at each pixel with Gaussian smoothing. As we know from previous lectures this can be done by convolving the original image with derivatives of Gaussians.

2. Compute matrix M in a Gaussian window around each pixel.

3. Compute corner response function R.

4. Threshold R to keep only points with large enough values of R

5. Find local maxima of response function using nonmaximum suppression

Scheme for corner detector These steps are common for many corner detectors and shown here once again in a diagram. Again, we start with an input image, apply a corner operator, which produces a "cornerness" map. then we feed that cornerness map to a thresholding operator and get a thresholded cornerness

map. Which is then fed to a non-maximum supression operator, which outputs positions of corners.

Example 1 Here is one example of these steps applied to a familiar Lena image. We see how we go from the original image to cornerness map, to thresholded map and finally to corners map. An image in the bottom shows detected corner locations superimposed on the input image.

Example 2 Let's look at one more example. Let's assume we have two images if the same object, but one is a rotated version of another. Lightning conditions are also different.

Let's compute Harris corner response function R for both of these images. Red points here correspond to high value of R and blue points correspond to low values of R, with orange, yellow and green in between.

The next step is to leave points with R values above a chosen threshold. So we leave all points that were red or orange on the previous slide and discard yellow, green and blue.

And the last step is perform non-maximum suppression - to find and keep on points of local maxima of R.

Now let's superimpose keypoints detected with Harris corner detector on the original images. You can notice that there are many pairs of corresponding keypoints, which is exactly what we wanted. Both images have keypoints detected where eyes and nostrils are, as well as along the boundaries of black spots, and in many other distinctive areas.

Properties of the Harris corner detector Let's now look at properties of the Harris corner detector. Ideally, we would like keypoint detectors to be invariant with respect to rotation, intensity changes and scale, so that the same keypoints can be found in rotated, lightened or darken, scaled versions of the images. What about Harris corner detector. Is it invariant to all these changes?

It is rotation invariant, as we just saw at an example with rotated toy. Eigenvalues remain the same for rotated versions of an image.

What about affine intensity changes? Yes, it is invariant to these changes too. Since only derivatives are used to compute corner response, it is invariant to intensity shift. And change in intensity scale can be handled by adaptive thresholding.

And what about scale? Unfortunately, the Harris detector is not scale invariant. Depending on the scale, the same fragment of an image can be considered as containing edges only, or as containing corners. Like on this example on the slide. On the left all windows along the contour will catch an edge, but there are

no edges. If we scale down the same fragment, no we have an edge fitting into a window - on the right.

Scale selection for scale invariant detector One solution to the problem is to extract features at a variety of scales. If we can change scale of a window (or scale of an image), then we will be able to find scales such that fragments will match. The same curve on the left as on the previous slide can be considered an edge and will match a curve on the right if a proper scale is selected.

This can be done by performing the same operation of corner detection at multiple resolutions in a pyramid of images, obtained by smoothing and sub-sampling input images.

But running corner detection for multiple resolutions and then performing $N \times N$ pairwise comparisons for N scales to find the best match is expensive. Instead of extracting features at many different scales and then matching all of them, it is more efficient to extract features that are stable in both location and scale. Harris detector is an example of a detector stable in location. What about scale? How can we choose the best scale for each fragment?

Automatic scale selection So, how that best scale can be automatically detected? Let's use a scale invariant function, so that the value of that function is the same for corresponding fragments, even if these fragments are of different scale. One example of such a function is average intensity. It will be the same for both scaled-up and scaled-down versions of the same image fragment. Let's call it a signature function.

Then we can look at this function as a function of a region size. For example, if Image 2 is a scaled-down version of an Image 1 with scaling factor of 1/2, then this function for images 1 and 3 will look something like these plots on the slide.

Then let's find a local maximum of that function and a region size that produces this local maximum. Local maximum is invariant to scale, so the region size that produces local maximum is a point of characteristic scale.

Characteristic scale Let's look at an example. Here we have two images at different scale. Let's pick a point in left image marked as a yellow cross. Now, let's change a size of window centered at that point and compute some signature function f for windows of different size. Below is a plot of that function f depending on the window size. We see that first it grows when we take larger and larger neighborhood around that yellow cross, but then starts to decrease. We can do the same for the image on the right. Now let's look at what window size the value of f was the largest for the left image and at what window size f produced the larges value for the right image. This point of local maximum

is characteristic scale. And a ratio between window sizes corresponding to local maxima on left and right images is a ratio of scales of these images.

Implementation Instead of computing function f for larger and larger windows, we can implement this search for characteristic scale using a fixed window size with a Gaussian pyramid. Gaussian pyramid is a type of multi-scale image representation, where an image is subject to repeated smoothing and subsampling. In a Gaussian pyramid, subsequent images are obtained using Gaussian blur and scale-down.

A common choice of the "signature" function f (1) What can be a good "signature" function f? The scale-normalized Laplacian-of-Gaussian (LoG) is a popular choice for a scale selection filter. As you may recall from the previous lectures, the LoG is the Laplacian operator $\nabla^2 f$ applied to the Gaussian function G. The 2-D Gaussian function is defined as

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{9}$$

The term $\frac{1}{2\pi\sigma^2}$ in front of the two-dimensional Gaussian kernel is the normalization constant. With the normalization constant this Gaussian kernel is a normalized kernel, i.e. its integral over its full domain is unity for every σ.

LoG is defined as

$$\nabla^2 G = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2} \tag{10}$$

LoG of normalized Gaussian is not scale invariant - it depends on σ, or the amount of blur. The response of a derivative of Gaussian filter to a perfect step edge decreases as σ increases. To keep response the same (scale invariant), we must multiply Gaussian derivative by σ. Laplacian is the second Gaussian derivative, so it must be multiplied by σ^2. So, to get scale independence, a scale-normalized LoG is introduced:

$$L(x, \sigma) = \sigma^2 \nabla^2 G \tag{11}$$

.

A common choice of the "signature" function f (2) The Laplacian-of-Gaussian detector is a scale-normalized LoG. As you can see on the slide, the LoG filter mask corresponds to a circular center-surround structure, with positive weights in the center region and negative weights in the surrounding ring structure. Thus, it will yield maximal responses if applied to an image neighborhood that

contains a similar (roughly circular) blob structure at a corresponding scale. If we look at blobs marked as img1, img2 and img3, they will be detected by a LoG filter of an appropriate scale.

Characteristic scale with Laplacian So to detect characteristic scale with Laplacian for a given point in the image, we need to look at Laplacian responces at different scale. And the scale that produces peak of Laplacian response is the characteristic scale.

The LoG detector Note that for blobs, a repeatable keypoint location can also be defined as the blob center. The LoG can thus both be applied for finding the characteristic scale for a given image location and for directly detecting scale-invariant regions by searching for 3D (location + scale) extrema of the LoG. This procedure is visualized on the slide. The image is convolved with LoG filters of different scale, producing a 3D response map. Maxima points in this 3D response map will point to scale and location of keypoints-blobs. The LoG detector is also sometimes called blob detector, because it detects blobs as regions of interest, and not corners as Harris detector.

Example: Scale-space blob detector This slide shows an example of LoG scale-space blob detector in action. Note different scales and locations for key-points.

Approximation of LoG with DoG Calculating second-order derivatives in Laplacian of Gaussian is computationally intensive. But is possible to approximate the LoG by a difference of two Gaussians (DoG):

$$\nabla^2 G \approx G(x, y; \sigma_1) - G(x, y; \sigma_2) \tag{12}$$

If we subtract one Gaussian from another, with the first having a greater standard deviation than the second, we will get a curve very similar to LoG. On the slide you can see plots of two Gaussians, one with $\sigma = 3.2$ in light blue, and another with $\sigma = 2.0$ in dark blue, their difference plotted in red is very similar to a Mexican Hat function (or LoG) plotted on the right.

The best approximation of LoG can be obtained when the standard deviation of the first Gaussian is about 1.6 times larger than the standard deviation of the second, so $\sigma_1/\sigma_2 = 1.6$, as it is for Gaussians on the left plot.

This approach was suggested by David Lowe as part of his Scale-Invariant Feature Transform (SIFT) algorithm.

Difference-of-Gaussian Similar to Laplacian of Gaussian, the difference of Gaussians can be utilized to increase the visibility of edges in an image. This slide shows the results of convolution of an image in the right top corner with two different Gaussian kernels and with their difference. Note, that convolving an image with a DoG kernel will produce the same result as subtracting the results of convolution of an image with two original Gaussian kernels:

$$I(x,y) \star G_1 - I(x,y) \star G_2 = I(x,y) \star (G_1 - G_2) \tag{13}$$

Blurring an image using a Gaussian kernel suppresses high-frequency spatial information. Subtracting one kernel from the other preserves spatial information that lies between the range of frequencies that are preserved by the two original filters. Thus, the DoG is a spatial band-pass filter.

Blob detection with DoG-1 So, we've just seen that LoG can be approximated with DoG. This means that Differences of Gaussians can also be used for blob detection in the scale-invariant feature transform. And it can be efficiently computed by subtracting adjacent scale levels of a Gaussian pyramid. Let's see how.

To detect image locations that are invariant to scale change, DoG detector uses a cascade filtering approach. It searches for an extrema in scale space, where scale space is a representation of an image as a family of smoothed images. These smoothed images simulate the loss of detail that would occur as the scale of an image decreases. The scale parameter controls the smoothing. Gaussian kernels are used to implement smoothing, and the scale parameter is the standard deviation, σ. So our scale space is a Gaussian pyramid: the input image is successively convolved with Gaussian kernels having standard deviations σ_1, $k\sigma_1$, $k^2\sigma_1$, $k^3\sigma_1$, . . . to generate a "stack" of Gaussian-filtered (smoothed) images that are separated by a constant factor k, as shown in the bottom of the slide.

Scale space is subdivided into octaves, with each octave corresponding to a doubling of σ, just as an octave in music theory corresponds to doubling the frequency of a sound signal. Then each octave is further subdivided into an integer number s of intervals, so $k = 2^{1/s}$. $s + 3$ images are produced in the a stack of blurred images for each octave. On the slide octave 1 consists of five images, thus $s = 2$ and $k = \sqrt{2}$. The input image is successively smoothed by convolution with Gaussians with standard deviations of σ_1, $\sqrt{2}\sigma_1$, $2\sigma_1$, etc. You can see on the left of the slide the resulting set of scale space images. Then adjacent Gaussian images are subtracted to produce the difference-of-Gaussian images shown on the right. Once a complete octave has been processed, we resample the Gaussian image that has twice the initial value of σ - it will be 2 images from the top of the stack, by taking every second pixel in each row and column.

It can be shown that when two adjacent scales are separated by a constant factor if k, the computation already includes the required scale normalization. So unlike LoG, there is no need to do additional scale normalization.

As in the case of the LoG detector, DoG interest regions are defined as locations that are simultaneously extrema in the image plane and along the scale coordinate of the $D(x, \sigma)$ function. Such points are found by comparing the $D(x, \sigma)$ value of each point with its 8-neighborhood on the same scale level, and with the 9 closest neighbors on each of the two adjacent levels, above and below, as shown on the right. Summarizing, the DoG region detector searches for 3D scale space extrema of the DoG function.

Example This slide shows images of the first three octaves of scale space in. The entries in the table are values of standard deviation used at each scale of each octave. For example the standard deviation used in scale 2 of octave 1 is $k\sigma_1$, which is equal to 1.

And this slide shows examples of DoG images. Here a total of $s+3$ Gaussian-filtered images, $s + 2$ difference-of-Gaussians formed in each octave from the adjacent pairs of Gaussian-filtered images in that octave. These differences can be viewed as images, and one sample of such an image is shown for each of the three octaves on the slide. As you might expect, the level of detail in these images decreases the further up we go in scale space.

Keypoints detection: invariance summary Let's summarize what we've learned about keypoints detection so far. To remind you, that the goal of every keypoint detection algorithm is given two images of the same scene to find the same keypoints (or interest points) independently in each image. And ideally we'd like that algorithm to be invariant to things like illumination change, rotation, scale and viewpoint. We've discussed three algorithms in details. First, the Harris corner detector, which detects corners as keypoints. We've learned that this detector is invariant to changes in illumination and rotation, but it is not scale-invariant.

Then we've discussed two scale-invariant detectors: LoG and DoG. Laplacian operator is rarely used on its own, as it is similar to DoG but more expensive to compute. However, it is sometimes used in combination with the Harris operator. This Harris-Laplace operator was proposed for increased discriminative power compared to the LoG or DoG operators. It combines the Harris operator's specificity for corner-like structures with the scale selection mechanism of LoG.

As mentioned before, DoG detector has been proposed by David Lowe as part of SIFT feature extraction algorithm, so it's also often referred to as SIFT detector.

Scale-invariant detection: summary Comparing these two scale-invariant detectors, Harris-Laplace and SIFT, they both target cases when we expect images of the same scene with a large difference in scale, and both of these methods search for maxima of certain a certain function (or functions) in both scale in space over the image.

Harris-Laplace method builds up two separate scale spaces for the Harris function and the Laplacian. The first version of this method uses the Harris function to localize candidate points on each scale level and selects those points for which the Laplacian simultaneously attains an extremum over scales. The resulting points are robust to changes in scale, image rotation, illumination, and camera noise. In addition, they are highly discriminative. As a drawback, this original Harris-Laplace detector typically returns a much smaller number of points than the LoG or DoG detectors. This is a result of the additional constraint that each point has to fulfill two different maxima conditions simultaneously. For many practical object recognition applications, the lower number of interest regions may be a disadvantage, as it reduces robustness to partial occlusion. For this reason, an updated version of the Harris-Laplace detector has been proposed based on a less strict criterion. Instead of searching for simultaneous maxima, it selects scale maxima of the Laplacian at locations for which the Harris function also attains a maximum at any scale. As a result, this modified detector yields more interest points at a slightly lower precision, which results in improved performance for applications where a larger absolute number of interest regions is required.

DoG (or SIFT) detector searches for maximums of Difference of Gaussians over both scale and space.

Local descriptors Now we know how to detect keypoints on different images independently. The next question is how to match these points, how to find corresponding pairs. We need local descriptors for these keyponts, such that they can be compared across two images using some similarity function. As with global features, we want local descriptors, to be invariant to changes in illumination, pose, scale, etc. But at the same time, these descriptors should be highly distinctive to allow a single feature to find its correct match with good probability in a large database of features.

As we know by now, scale-invariant keypont detectors can give us information about scale. So we can use that information to pick a neighborhood of a proper scale at keypoint location to compute a descriptor. Global descriptors which we've discussed earlier, such as color or intensity histograms, filter responses, and others can be used here too, when applied to this local patch of an image of a given scale and at a given location provided by the keypoint detection algorithm. But many of these features are either sensitive to small variations of location and pose, or poorly distinctive, or both. One of the most successful local descriptors

was proposed by David Lowe together with the DoG keypoint detector. It is well-known SIFT descriptor.

SIFT descriptor: main steps The major stages of of computation SIFT descriptors are as the following:

1. Scale-space extrema detection. We we've discussed this stage in details. It is implemented using Difference-of-Gaussians.

2. Keypoint localization. At each candidate location, a detailed model is fit to determine location and scale. Keypoints are selected based on measures of their stability, edge responses are eliminated.

3. Orientation assignment. One or more orientations are assigned to each keypoint location based on local image gradient directions. All future operations are performed on image data that has been transformed relative to the assigned orientation, scale, and location for each feature.

4. Keypoint descriptor. The local image gradients are computed at the selected scale in the region around each keypoint. And these are transformed into a representation that allows for significant levels of local shape distortion and change in illumination.

Let's take a more detailed look at steps 3 and 4.

Keypoints orientation To characterize the image at each key location, the smoothed image at each level of the pyramid is processed to extract image gradients and orientations. At each pixel, A_{ij} , the image gradient magnitude, M_{ij} , and orientation, R_{ij}, are computed using pixel differences:

$$M_{ij} = \sqrt{(A_{ij} - A_{i+1,j})^2 + (A_{ij} - A_{i,j+1})^2} \tag{14}$$

$$R_{ij} = atan2(A_{ij} - A_{i+1,j}, A_{i,j+1} - A_{ij}) \tag{15}$$

Each key location is assigned a canonical orientation so that the image descriptors are invariant to rotation. In order to make this as stable as possible against lighting or contrast changes, the orientation is determined by the peak in a histogram of local image gradient orientations. This histogram is formed from the gradient orientations of sample points in a neighborhood of each keypoint. The histogram has 36 bins covering the 360◦ range of orientations on the image plane. Each sample added to the histogram is weighed by its gradient magnitude, and by a circular Gaussian function with a standard deviation 1.5 times the scale

of the keypoint. Peaks in the histogram correspond to dominant local directions of local gradients. The highest peak in the histogram is detected and any other local peak that is within 80% of the highest peak is used also to create another keypoint with that orientation. Thus, for the locations with multiple peaks of similar magnitude, there will be multiple keypoints created at the same location and scale, but with different orientations.

The image on the slide shows keypoints superimposed on the image with keypoint orientations as arrows. Note the consistency of orientation of similar sets of keypoints in the image. For example, look at the keypoints on the right, vertical corner of the building. The lengths of the arrows vary, depending on illumination and image content, but their direction is unmistakably consistent.

SIFT descriptor So, we have assigned an image location, scale, and orientation to each keypoint. These parameters impose a repeatable local 2D coordinate system in which to describe the local image region, and therefore provide invariance to these parameters. The next step is to compute a descriptor for the local image region that is highly distinctive and at the same time is as invariant as possible to remaining variations, such as change in illumination or 3D viewpoint.

The approach used by SIFT to compute descriptors is based on experimental results suggesting that local image gradients appear to perform a function similar to what human vision does for matching and recognizing 3-D objects from different viewpoints. This slide illustrates the computation of the keypoint descriptor.

First the image gradient magnitudes and orientations are sampled around the keypoint location, using the scale of the keypoint to select the level of Gaussian blur for the image. In order to achieve orientation invariance, the coordinates of the descriptor and the gradient orientations are rotated relative to the keypoint orientation. For efficiency, the gradients are precomputed for all levels of the pyramid and used for both keypoint orientation detection and computation of the descriptor. The image gradients are illustrated with small arrows at each sample location in the center of the slide. A Gaussian weighting function with σ equal to one half the width of the descriptor window is used to assign a weight to the magnitude of each sample point. This is illustrated with a circular blue window in the middle of the slide, although, of course, the weight falls off smoothly. The purpose of this Gaussian window is to avoid sudden changes in the descriptor with small changes in the position of the window, and to give less emphasis to gradients that are far from the center of the descriptor.

The keypoint descriptor is shown on the right side of the slide. Orientation histograms are created over 4x4 sample regions. The figure shows eight directions for each orientation histogram, with the length of each arrow corresponding to the magnitude of that histogram entry. A gradient sample in the central image can shift up to 4 sample positions while still contributing to the same histogram

on the right, so this descriptor allows for local positional shifts. The descriptor is formed from a vector containing the values of all the orientation histogram entries, corresponding to the lengths of the arrows on the right image. The figure on the right shows a 2x2 array of orientation histograms, but according to the original paper the best results are achieved with a 4x4 array of histograms with 8 orientation bins in each. Therefore, the SIFT descriptor is usually a 4x4x8 = 128 element feature vector for each keypoint.

1.2 Model fitting

Keypoints and local descriptors are useful tools to detect and describe small salient regions of an image. But these descriptors are often noisy and can't tell much about a larger scale picture. Often we know in advance shapes of objects we are interested in. One example of most common shapes we might be interested in detecting are straight lines. The man-made world is full of lines. Detecting and matching these lines can be useful in a variety of applications. Among those are lane detection and mode general road markings detection for assisted and autonomous driving. For example, parking spot detection. Other common examples are detection of field lines in sports video analysis, architectural modeling, camera pose estimation in urban environments, the analysis of printed document layouts, and many others. Circles are also among common simple shapes. Like on this image on the slide, if the task is to automatically count coins, one would benefit from the knowledge that coins are expected to be circular.

In these cases, when we know in advance what type of shapes we are looking for, we can choose a parametric model to represent a set of keypoints forming an object of interest, and then find the right parameters of that model, so the model fits the observed data.

Parametric model So what is that parametric model? A parametric model is a function of image data and free parameters. This function defines a class of shapes we are looking in the image. But the exact location of that shape and other properties of that shape, such as size, are parameterized. A goal of model fitting process is to find parameters such that the given model fits the observed data the best.

Line fitting For example, when we are looking for lines, we are given a set of points - this is image data, and our goal is to find a straight line that approximates the given points the best. Our parametric model is a line equation: $y = mx + b$. We know many pairs of (x, y) - these are the keypoints, image data. And our task is to find values of parameters m and b so that the line equation $y = mx + b$ is an accurate fit for all (or most) of the keypoints.

On this slide, imagine that you have detected red keypoints, and your task is to find the best line that can fit these points - like a blue line here.

Model fitting: common steps There are many different model fitting techniques and algorithms, but all of them follow the same set of steps.

- First, one needs to choose a parametric model, which is expected to fit the given data best. If you are looking for lines on an image, your parametric model will be a line. If you are looking for circles (and expect to find them on a given image), then choose parametric equation of a circle or an ellipse as your model.
- Once you choose a model, the next step is to fit that model to your data, to find values for all parameters in your model, so that the model with these parameters approximates your data well. Keep in mind, that the real data is always not precise. You'll be dealing with noise in detected keypoints' locations, with outliers and clutter, with missing points resulting from occlusions. The model should be robust to all these challenges and describe actual data well. It's never feasible to fit the model precisely to all the data points you have. But the best model will be the one which fits as many points as possible. So you need to maximize a number of "inliers" and minimize a number of "outliers" among the data.
- Once you have your parametric model and parameters which fit your data the best, you can filter out data points which don't fit the model. These are outliers.

Different fitting algorithms approaching step number two differently. This is the most important step among the three steps here. So, how do we find "optimal" parameters for a chosen model?

Fitting methods Let's use line fitting task as an example. One of the easiest algorithms used for model fitting is least squares. It can be used, when we know which points belong to the line, and the task is to find "optimal" line parameters. In other words, when we have multiple data points, we know, that they all belong to the same line, and we need to find parameters defining that line. Least squares methods can also be used to find other objects, with other types of models. But this method is very sensitive to outliers, so it only works well when there is no noise and all the data points that you have are expected to be inliers.

When there are outliers, it is better to choose a more robust fitting algorithm. One of the most popular algorithms in this category is RANSAC, which stands for Random Sample Consensus.

If you expect to have multiple instances of an object, like multiple different lines, you can use several modifications of RANSAC, like Sequential RANSAC, or other voting methods. Hough transform is a very popular one for detecting multiple lines. And it can also be used to fit other models, detect objects of other shapes.

Now, let's look at these popular fitting techniques a bit closer.

Least squares line fitting We will start with the method of least squares. This method is a standard approach in regression analysis to approximate the solution of overdetermined systems - when there are more equations than unknowns. The key idea of the method is to minimize the sum of squared residuals made in the results of every single equation.

In data fitting application, a residual is the difference between an observed value, and the fitted value provided by a model. Least squares method is a special sort of minimization - the goal is to minimize squared residuals.

Let's start from the beginning. Our objective is to find the parameters of a model function to best fit a data set. Our data set consists of n points (data pairs): $(x_1, y_1), \ldots, (x_n, y_n)$. The model function has the form of a line equation: $y = mx + b$. The goal is to find the parameter values (m, b) for the model that "best" fit the data. The fit of a model to a data point (x_i, y_i) is measured by its residual, defined as the difference between the actual value y_i and the value predicted by the model: $r_i = y_i - mx_i - b$. Our goal is to minimize sum of squared residuals for all data points:

$$E = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - mx_i - b)^2 \tag{16}$$

If we put everything in vectors and matrices, we can express this equation in matrix form:

$$E = ||Y - XB||^2 \tag{17}$$

where

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \; X = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}, \; B = \begin{bmatrix} m \\ b \end{bmatrix} \tag{18}$$

This can be re-written as follows:

$$E = ||Y - XB||^2 = (Y - XB)^T (Y - XB) = Y^T Y - 2(XB)^T Y + (XB)^T (XB) \tag{19}$$

The minimum of E can be found by setting the partial derivative with respect to B to zero:

$$\frac{\partial E}{\partial B} = 2X^T XB - 2X^T Y = 0 \tag{20}$$

The first term in E, $Y^T Y$ doesn't depend on B, so it's partial derivative with respect to B is zero, the second term gives us $-2X^T Y$, and the last one gives $2X^T XB$.

This gives us the normal equations $X^T XB = X^T Y$. They are called the normal equations because they specify that the residual must be normal (orthogonal) to every vector in the span of X. Rearranging, we have

$$B = (X^T X)^{-1} X^T Y \tag{21}$$

Problems with traditional Least Squares Although traditional (also called ordinary) least squares method is very common and easy to use, it has some problems. First, it is not rotation-invariant. Since we are minimizing the residuals along the y, when image is rotated, residuals along the y axis will change. Second, it fails completely for vertical lines.

Let's also note that the traditional least squares method is asymmetric. We treat Y as a dependent variable and X as an independent variable, and the independent variable is assumed to be error-free.

All these problems can be resolved with another version of the least squares method, called Total Least Squares.

Total Least Squares The least squares and total least squares methods assess the fitting accuracy in different ways: the least squares method minimizes the sum of the squared vertical distances from the data points to the fitting line, while the total least squares method minimizes the sum of the squared orthogonal distances from the data points to the fitting line.

Coming back to our line fitting problem, we can now reformulate it with total least squares method as minimizing the sum of distances between points (x_i, y_i) and line $ax + by = d$, where (a, b) is a unit normal to the line, so $a^2 + b^2 = 1$. Our goal is to find such (a, b, d) which minimize the sum of perpendicular distances:

$$E = \sum_{i=1}^{n} (ax_i + by_i - d)^2 \tag{22}$$

Least squares v.s. total least squares This slides shows the difference between the least squares and total least squares fitting lines to the same set of points. The crosses lying on the lines are data approximations with these two fitting approaches. In the least squares case, the data approximation is obtained by

correcting the second coordinate only: the original data points and their approximations on the line have the same x value, and only y values are different. In the total least squares case, the data approximation is obtained by correcting both coordinates.Total least squares is a type of errors-in-variables regression, a least squares data modeling technique in which observational errors on both dependent and independent variables are taken into account.

Least squares: robustness ot noise There is another problem with least squares methods, both traditional and total least squares. They are very sensitive to noise. These method will fit line to the data very well, when all the data points are close to a line, like on this slide.

But if our data has an outlier, the least squares fit will be far from ideal. Squared error heavily penalizes outliers. And the errors in the data will affect more heavily the total least squares solution than the least squares solution.

Robust estimators When there are just a few outliers, the problem can be solved with robust estimators. Robust estimators rely on robust penalty functions which are more "forgiving" for data points which fall far from their approximations with a given model.

A general approach is to find model parameters θ which minimize a sum of values of a robust function for all data points x_i: $\sum_i \rho(r_i(x_i, \theta), \sigma)$. The robust function ρ is a function of residuals r_i and a scale parameter σ. On a slide you can see one example of such robust function, $\rho(u, \sigma) = \frac{u^2}{\sigma^2+u^2}$. It behaves like squared distance for small values of the residual u, but saturates for larger values of u. So if your data has just a few outliers, with a robust estimator you'll be able to fit the model to the inliers, and the effect of the outliers on the model parameters will be saturated.

Robust fitting is a nonlinear optimization and solved using iterative methods (unlike least squares which has a closed form solution and can be solved analytically). Least squares solution can be used for initialization.

The scale parameter of the robust function should be chosen carefully - with a value too small, the error will be almost the same for every point, and thus it will be hard to find a good fit. When the scale value is too large, it will behave similar to least squares.

RANSAC While M-estimators can definitely help reduce the influence of outliers, in cases with too many outliers it's hard for these methods to converge to the global optimum. A better approach is often to find a starting set of inlier correspondences, i.e., points that are consistent with a dominant parameters estimate. A widely used approach to this problem is called RANSAC, or Random Sample

Consensus. It is a very general framework for model fitting in the presence of outliers.

The outline of this method is as follows.

- Start by selecting uniformly at random a subset of points
- Fit a model to that subset
- Find all remaining points that are "close" to the model - count the number of inliers that are within ϵ of their predicted location by the model. Reject the rest as outliers.
- Do this many times and choose the best model as the one with the largest number of inliers.

RANSAC for line fitting example Let's look at a line fitting example with RANSAC. Let's assume our data points look like pictured on the slide. You can see there there is a clear diagonal line going from left bottom to right top. But there are many outliers and noise.

Least squares method is not capable of finding a good solution due to a very large number of outliers.

With RANSAC, we first randomly select a minimal subset of points, shown in red on this slide. Since we are fitting a line, a minimal subset of points to define a line is two. Then we fit a model to that subset. Then we compute error function for this model and all the data points we have. Next, we select inliers - points consistent with the model. These are the points within small distance ϵ from the line defined by our model. These points shown in green on the slide.

The we repeat this loop, starting from a random draw of a minimal subset of points, them model fitting and inliers count. This slide shows another hypothesis of a line.

Repeat again.

And again.

And again.

And then we will choose a model hypothesis which has the largest number of inliers.

RANSAC for line fitting Summarizing, RANSAC for line fitting consists of the following steps, repeated N times.

- Draw s points uniformly at random. Usually, a minimum number of sample points possible is used. So, for line fitting we will draw two points.
- Fit a line to these s points.

- Find inliers to this line among the remaining points: find points whose distance from the line is less then ϵ.
- If there are d or more inliers, accept the line and refit using all inliers.

RANSAC pros and cons RANSAC is simple, general and robust. It can estimate the model parameters with a high degree of accuracy even when a significant number of outliers are present in the data set. It is used very widely and applicable to many different problems, not only to model fitting. It also often works well in practice.

But it has its disadvantages too. As you've seen on the previous slide, it depends on a very large number of parameters. It requires approaches to determine the best number of iterations, number of points to draw per iteration, the right distance to filter out outliers, etc. And good choice of these parameters is often very important - bad choice can lead to very long compute time or inaccurate results. When the number of iterations is limited, the obtained solution may not be optimal, and it may not even be one that fits the data in a good way. RANSAC offers a trade-off; by computing a greater number of iterations the probability of a reasonable model being produced is increased.

It doesn't work well when there too few inliers and too many outliers. RANSAC is not always able to find the optimal set even for moderately contaminated sets and it usually performs badly when the number of inliers is less than 50%.

Depending on a model, it could be hard to find a good initialization of the model based on the minimum number of samples.

Fitting methods We've discussed so far a few most popular fitting methods. Least squares and its variants are the easiest methods. But they fail when there are too many outliers. Applied to line fitting, least squares works nicely, when most of the data points are close to one line. When there are many outliers, or when data points form multiple lines, least squares methods fail.

Robust estimators are less sensitive to outliers, but still fail when there are too many of them.

RANSAC is a very popular approach which deals reasonably well with outliers. But vanilla RANSAC can only estimate one model for a particular data set. When two (or more) model instances exist, RANSAC will detect only one of the instance in best case, or even may fail to find either one.

What if there are many lines? As you can see on this slide, least squares approaches fail completely in the presence of multiple models and outliers, and RANSAC is capable of detecting one model - one line, out of five. This line may

have more inliers than other lines. Or it may just "got lucky" and detected within the given number of iterations of RANSAC.

Fitting methods RANSAC method can be adapted to multi-model case. One of the most straightforward modifications of RANSAC for the multi-model case is conventionally referred to as Sequential RANSAC. The Hough transform is another alternative robust estimation technique that is commonly used when more than one model instance is present. There are other approaches for multi model fitting too, but we will focus on these two today as the most widely known. They can be commonly referred as voting methods, because they are based on the notion of data points "voting" for a particular model instance.

Voting schemes The main principle of all voting schemes is the same. Every data point votes for all the models that are compatible with this point. A model with the largest number of votes wins. For the multi-model case, we can keep several models instead of one winner. Pick several models with number of votes above a certain threshold. And these models are all the winners and represent several instances of the same object present on an image.

These voting schemes work under assumption that the noise data points won't vote consistently for any particular model, rather their votes will be distributed more or less uniformly across all models. And a "good" model will get more votes, because it will get votes of inliers in addition to random votes of noise data points.

Another nice property of the voting schemes is that they are robust to missing data. As long as there are enough data points to agree on a good model, missing data doesn't matter.

Sequential RANSAC RANSAC by design is a voting method. But in as we saw, in it's original formulation it can only fit one model. How can we make it fit multiple model instances? Sequential RANSAC is a way to go. The idea is very simple. We will use vanilla RANSAC to fir a model. When we have the first model instance, let's remove all data points compatible with that model instance. Like on this slide, if RANSAC detected this red line, let's remove all inliers for this model, and start from scratch.

With points corresponding to the first model instance removed, the second execution of RANSAC will find us another model instance with the next largest number of voters. And then we will remove data points consistent with this model, and proceed.

You get the idea. By removing all data points consistent with already detected model instances, we use RANSAC to fit the next model. We can continue until the number of votes for the best model is below a certain threshold, or until

there is no clear winner anymore - if all model candidates get approximately the same number of votes.

Hough transform Another very popular model fitting method based on a voting scheme is Hough Transform. It is probably the most well-known method for line detection, and it can also be used to detect other shapes. Let's take a look how it works.

In its original formulation, each point votes for all possible lines passing through it, and lines corresponding to high accumulator or bin values are examined for potential line fits.

Let (x_1, y_1) denote a point in the xy-plane and consider the general equation of a straight line in slope-intercept form: $y = mx + b$. Infinitely many lines pass through the point (x_1, y_1), but they all satisfy the equation $y_1 = mx_1 + b$ for varying values of m and b. Let's re-write this equation as $b = -x_1m + y_1$ and consider the mb-plane. This is parameter space - a space of all possible values of parameters m and b. We got the equation of a single line in that space, with fixed values of (x_1, y_1).

Now let's add a second point on the xy-plane, (x_2, y_2). It also has a single line in parameter space associated with it, $b = -x_2m+y_2$. This line intersects the line associated with the first point (x_1, y_1) at some point (m_0, b_0) in the parameter space mb. m_0 is the slope and b_0 is the intercept of the line containing both points, (x_1, y_1) and $(x - 2, y_2)$ in the xy-plane. So, a single point (m_0, b_0) in parameter space has the corresponding line $y = m_0x + b_0$. And all points on this line in xy-plane will have lines in parameter space that intersect at (m_0, b_0).

Now it's easy to understand the main principle of the Hough Transform. Let's discretize parameter space mb into bins. Then for each feature point in the image, put a vote in every bin in the parameter space that could have generated this point. And, finally, find bins that have the most votes. m and b values corresponding to these bins define detected lines on the original image.

Parameter space representation This approach is easy to understand and implement, but there are a couple of issues with the mb-space. First, it has unbounded parameter domains. Both parameters, m and b can have any values from $-\infty$ to $+\infty$. This makes it hard to bin the space into a limited number of bins. And second, m, the slope of a line, approaches infinity as the line in xy-space approaches the vertical direction.

One way around these difficulties is to use the normal (or polar) representation of a line: $x\cos(\theta) + y\sin(\theta) = \rho$. On the slide you can see the geometrical interpretation of the parameters ρ and θ. A horizontal line has $\theta = 0\circ$, with ρ being equal to the positive x-intercept. Similarly, a vertical line has $\theta = 90\circ$, with

ρ being equal to the positive y-intercept, or $\theta = -90\circ$, with ρ being equal to the negative y-intercept.

Hough transform in polar coordinates Similarly to mb parameter space, we can now consider $\theta\rho$ parameter space. Each point in xy-plane will have a corresponding sinusoidal curve in $\theta\rho$-plane, and if there are several points lying on the same line in xy-space, their corresponding sinusoidal curves in $\theta\rho$-space will intersect in one point, and coordinates (θ, ρ) of that intersection gives us parameters of the line in xy-space.

To make the Hough transform computationally attractive, we subdivide the $\theta\rho$ parameter space into accumulator cells, or bins. As opposed to mb-space, now we have well defined expected ranges of θ and ρ values, $(\theta_{min}, \theta_{max})$ and (ρ_{min}, ρ_{max}). With $\theta_{min} = -90\circ$ and $\theta_{max} = 90\circ$ we can represent all possible slopes of the lines. And with $-D \leq \rho \leq D$, where D is the maximum distance between opposite corners in an image, we can fit all possible lines within an image.

Algorithm outline Let's outline the algorithm. Initially, set all accumulator cells to zero. Then, for every feature point (x_k, y_k) in the xy-plane of an image, we iterate all allowed θ values on the θ-axis and solve for the corresponding ρ using the equation $\rho = x_k \cos(\theta) + y_k \sin(\theta)$. The resulting ρ values are then rounded off to the nearest allowed cell value along the ρ axis. Then increase counter of the cell (θ, ρ) by one. At the end of the procedure, a value of K in a cell (θ_i, ρ_j) means that K points in the xy-plane lie on the line $x\cos(\theta_i) + y\sin(\theta_i) = \rho_j$. After we iterated all feature points, find accumulator cells corresponding to a local maxima. If a cell (θ_i, ρ_j) corresponds to a local maxima, then the detected line in the image is given by the equation $x\cos\theta_i + y\sin\theta_i = \rho$æ.

The number of subdivisions in the $\theta\rho$-plane (or grid size) determines the accuracy of the colinearity of the points. It can be shown that the number of computations in this method is linear with respect to n, the number of feature points in the xy-plane of an image.

Although we used line detection as an example here, the Hough transform is applicable to any function of the form $g(v, c) = 0$, where v is a vector of coordinates and c is a vector of coefficients. There also exist improved versions of this method, when gradient orientations of edge points are taken into account when iterating over possible values of *theta*. Before Deep Learning era, Generalized Hough Transform method was among popular methods for object detection via templates.