Necessary background

Image gradients:

An image gradient is a directional change in the intensity or color in an image. Image gradients may be used to extract information from images.

In graphics software for digital image editing, the term gradient is used for a gradual blend of color which can be considered as an even gradation from low to high values, as used from white to black in the images to the right. Another name for this is color progression.

Mathematically, the gradient of a two-variable function (here the image intensity function) is at each image point a 2D vector with the components given by the derivatives in the horizontal and vertical directions. At each image point, the gradient vector points in the direction of largest possible intensity increase, and the length of the gradient vector corresponds to the rate of change in that direction.

Since the intensity function of a digital image is only known at discrete points, derivatives of this function cannot be defined unless we assume that there is an underlying continuous intensity function which has been sampled at the image points. With some additional assumptions, the derivative of the continuous intensity function can be computed as a function on the sampled intensity function, i.e., the digital image. It turns out that the derivatives at any particular point are functions of the intensity values at virtually all image points. However, approximations of these derivative functions can be defined at lesser or larger degrees of accuracy.
For a point \((x, y)\) in a grayscale image \(I\) the partial derivative of the intensity with respect to \(x\) at this location can be defined as:

\[
\frac{\partial I}{\partial x} = I(x+1, y) - I(x-1, y)
\]

Similarly:

\[
\frac{\partial I}{\partial Y} = I(x, y+1) - I(x, y-1)
\]

Thus, the gradient becomes a 2D vector \(g\) where:

\[
g = \frac{\partial I}{\partial x} \hat{i} + \frac{\partial I}{\partial y} \hat{j}
\]

This vector has a magnitude \(m\) and orientation \(\theta\) where:

\[
m = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}
\]

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

Where \(\theta\) is the angle the vector makes with the positive \(x\) axis.
Linear Filters and Convolution:

Many important effects can be modeled with a quite simple model. Construct a new array, the same size as the image. Fill each location of this new array with a weighted sum of the pixel values from the locations surrounding the corresponding location in the image, using the same set of weights each time. Different sets of weights could be used to represent different processes — for example, we could use a set of weights that was large at the center and fell off sharply as the distance from the center increased to model the kind of smoothing that occurs in a defocused lens system. The result of this procedure is shift-invariant — meaning that the value of the output depends on the pattern in an image neighborhood, rather than the position of the neighborhood — and linear — meaning that the output for the sum of two images is the same as the sum of the outputs obtained for the images separately. The procedure itself is known as linear filtering.

Convolution

We introduce some notation at this point. The pattern of weights used for a linear filter is usually referred to as the kernel of the filter. The process of applying the filter is usually referred to as convolution.

In particular, given a filter kernel $H$, the convolution of the kernel with image $F$ is an image $R$. The $i$’th and $j$’th component of $R$ are given by:

$$ R_{(i,j)} = \sum_{u,v} H_{(i-u, j-v)} F_{(u,v)} $$

We carefully avoid inserting the range of the sum; in effect, we assume that the sum is over a large enough range of $u$ and $v$ that all non-zero values are taken into account. We will use this convention — which is common — regularly in what follows.
Example on Convolution:

For an input grayscale image $I$ where:

\[
I = \begin{bmatrix}
100 & 27 & 163 & 235 & 102 \\
115 & 255 & 178 & 156 & 24 \\
29 & 41 & 210 & 154 & 147 \\
90 & 94 & 74 & 24 & 36 \\
41 & 86 & 35 & 85 & 18
\end{bmatrix}
\]

And a Gaussian kernel (see later sections for information on the Gaussian) $k$ where:

\[
k = \begin{bmatrix}
0.0177 & 0.0862 & 0.0177 \\
0.0862 & 0.6366 & 0.0862 \\
0.0177 & 0.0862 & 0.0177
\end{bmatrix}
\]

The result of convolving $I$ with $k$ at pixel $(3, 3)$ is:

\[
R_{(3,3)} = 255 \cdot 0.0177 + 178 \cdot 0.0862 + 156 \cdot 0.0177 + 41 \cdot 0.0862 + 210 \cdot 0.6366 + 154 \cdot 0.0862 + 94 \cdot 0.0177 + 74 \cdot 0.0862 + 24 \cdot 0.0177
\]

Example: Smoothing by Averaging

Images typically have the property that the value of a pixel is usually similar to that of its neighbor. Assume that the image is affected by noise of a form where we can reasonably expect that this property is preserved. For example, there might be occasional dead pixels; or small random numbers with zero mean might have been added to the pixel values. It is natural to attempt to reduce the effects of this noise by replacing each pixel with a weighted average of its neighbors, a process often referred to as smoothing or blurring.

At first guess, we could model the blurring process as replacing the value of a function at each point with an unweighted (or uniform) average taken over a fixed region. For example, we could average all pixels within a $2k + 1 \times 2k + 1$ block of the pixel of interest. For an input image $F$, this gives an output

\[
R_{(i,j)} = \frac{1}{(2k+1)^2} \sum_{u=i-k}^{i+k} \sum_{v=j-k}^{j+k} F_{(u,v)}
\]

This is the same as convolution with a kernel that is a $2k+1 \times 2k+1$ block of ones, multiplied by a constant — you can establish this point by close attention to the range of the sum.
This process is a poor model of blurring — its output does not look like that of a defocused camera. The reason is clear. Assume that we have an image in which every point but the center point was zero, and the center point was one. If we blur this image by forming an unweighted average at each point, the result will look like a small bright box — but this is not what defocused cameras do. We want a blurring process that takes a very small bright dot to a circularly symmetric region of blur, brighter at the center than at the edges and fading slowly to darkness. As the figure suggests, a set of weights of this form produces a much more convincing defocus model.

Although a uniform local average may seem to give a good blurring model, it generates effects that are not usually seen in defocusing a lens. The images above compare the effects of a uniform local average with weighted average. The image at the top shows a view of grass. On the left in the second row, the result of blurring this image using a uniform local model and on the right, the result of blurring this image using a set of Gaussian weights. The degree of blurring in each case is about the same, but the uniform average produces a set of narrow vertical and horizontal bars — an effect often known as ringing. The bottom row shows the weights used to blur the image, themselves rendered as an image; bright points represent large values and dark points represent small values (in this example the smallest values are zero).
Example: Smoothing with a Gaussian

A good formal model for this fuzzy blob is the symmetric Gaussian kernel:

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

As illustrated in the figure. \(\sigma\) is referred to as the standard deviation of the Gaussian (or its “sigma”!); the units are inter-pixel spaces, usually referred to as pixels. The constant term makes the integral over the whole plane equal to one and is often ignored in smoothing applications. The name comes from the fact that this kernel has the form of the probability density for a 2D Gaussian random variable with a particular covariance.

This smoothing kernel forms a weighted average that weights pixels at its center much more strongly than at its boundaries. One can justify this approach qualitatively: smoothing suppresses noise by enforcing the requirement that pixels should look like their neighbors; and by down-weighting distant neighbors in the average, we can ensure that the requirement that a pixel look like its neighbors is less strongly imposed for distant neighbors. A qualitative analysis gives:

• If standard deviation of the Gaussian is very small — say smaller than one pixel — the smoothing will have very little effect, because the weights for all pixels off the center will be very small;

• For a larger standard deviation, the neighboring pixels will have larger weights in the weighted average, this means in turn that the average will be strongly biased toward a consensus of the neighbors — this will be a good estimate of a pixel’s value and the noise will largely disappear, at the cost of some blurring;

• Finally, a kernel that has very large standard deviation will cause much of the
image detail to disappear along with the noise.

In applications, a discrete smoothing kernel is obtained by constructing a $2k + 1 \times 2k + 1$ array whose $i^{th}$ and $j^{th}$ value is:

$$H_{(i,j)} = \frac{1}{2\pi\sigma^2} \exp \left(-\frac{(i-k-1)^2 + (j-k-1)^2}{2\sigma^2}\right)$$

Notice that some care must be exercised with $\sigma$; if $\sigma$ is too small, then only one element of the array will have a non-zero value. If $\sigma$ is large, then $k$ must be large, too, otherwise we will be ignoring contributions from pixels that should contribute with substantial weight.

**Sample Gaussian matrix**

This is a sample matrix, produced by sampling the Gaussian filter kernel (with $\sigma = 0.84089642$) at the midpoints of each pixel and then normalizing. Note that the center element (at $[0, 0]$) has the largest value, decreasing symmetrically as distance from the center increases.

0.00000067 0.00002292 **0.00019117** 0.00038771 **0.00019117** 0.00002292 0.00000067 0.00002292 0.00078633 0.00655965 0.01330373 0.00655965 0.00078633 0.00002292 **0.00019117** 0.00655965 0.05472157 0.11098164 0.05472157 0.00655965 **0.00019117** 0.00038771 0.01330373 0.11098164 **0.22508352** 0.11098164 0.01330373 0.00038771 **0.00019117** 0.00655965 0.05472157 0.11098164 0.05472157 0.00655965 **0.00019117** 0.00002292 0.00078633 0.00655965 0.01330373 0.00655965 0.00078633 0.00002292 0.00000067 0.00002292 **0.00019117** 0.00038771 **0.00019117** 0.00002292 0.00000067

**Example**

![Original image](image1.png) ![image blurred using Gaussian blur with $\sigma = 2$.](image2.png)
**Image Pyramids**

Sometimes, we do not know what the appropriate resolution for the image should be. Consider, for example, the task of finding a face in an image (x14.2). Since we do not know at what scale the face will appear, we need to generate a whole pyramid of differently sized images and scan each one for possible faces.

Such a pyramid can also be very helpful in accelerating the search for an object by first finding a smaller instance of that object at a smaller (coarser) level of the pyramid and then looking for the full resolution object only in the vicinity of coarse-level detections.

An image pyramid is a group of images where in pyramid-like structure where the lowest level of the pyramid contains the largest image and the higher levels contain smaller (scaled-down) images.

Scaling down an image is usually done by first smoothing the image with a smoothing filter, then subsampling it.

Subsampling is done by taking every $k^{th}$ pixel in the smoothed image to produce a new image of the required size. In other words, if we have an image I of size 320*240 and it's required to produce a new image that is 0.75 the size of this image. This can be done by following these steps:
• Smooth the input image I with a Gaussian or averaging kernel.
• Create a new image L of the new size (240*180).
• Fill the new image using this equation:
  \[ L(x, y) = I(x / 0.75, y / 0.75) \]

Left: original image  
Right: a resized version where scale=0.32
Corners as Interest Points:

Many applications require relating two or more images in order to extract information from them. For example, if two successive frames in a video sequence taken from a moving camera can be related, it is possible to extract information regarding the depth of objects in the environment and the speed of the camera. The brute force method of comparing every pixel in the two images is computationally prohibitive for the majority of applications. Intuitively, one can image relating two images by matching only locations in the image that are in some way interesting. Such points are referred to as interest points and are located using an interest point detector. Finding a relationship between images is then performed using only these points. This drastically reduces the required computation time.

Many different interest point detectors have been proposed with a wide range of definitions for what points in an image are interesting. Some detectors find points of high local symmetry; others find areas of highly varying texture, while others locate corner points. Corner points are interesting as they are formed from two or more edges and edges usually define the boundary between two different objects or parts of the same object. Many corner detectors have been developed and this website investigates some of the more important ones.

Applications of Corner Detectors

The use of interest points (and thus corner detectors) to find corresponding points across multiple images is a key step in many image processing and computer vision applications. Some of the most notable examples are:

- stereo matching
- image registration (of particular importance in medical imaging)
- stitching of panoramic photographs
- object detection/ recognition
- motion tracking
- robot navigation

A corner can be defined as the intersection of two edges. A corner can also be defined as a point for which there are two dominant and different edge directions in a local neighborhood of the point.

An interest point is a point in an image which has a well-defined position and can be robustly detected. This means that an interest point can be a corner but it can also be, for example, an isolated point of local intensity maximum or minimum, line endings, or a point on a curve where the curvature is locally maximal.

In practice, most so-called corner detection methods detect interest points in general, rather than corners in particular. As a consequence, if only corners are to be detected it is
necessary to do a local analysis of detected interest points to determine which of these are real corners.

**Moravec Operator**

This operator was developed by Hans P. Moravec in 1977 for his research involving the navigation of the Stanford Cart through a clustered environment. Moravec defined the concept of "points of interest" as being distinct regions in images and concluded these interest points could be used to find matching regions in consecutive image frames. This was a vital low-level processing step that allowed him to determine the existence and location of objects in the vehicle's environment.

The Moravec operator is considered a corner detector since it defines interest points as points where there is a large intensity variation in every direction. This is the case at corners. However, Moravec was not specifically interested in finding corners, just distinct regions in an image that could be used to register consecutive image frames. Many have commended this relaxation in the "definition" of what a corner is, since the concept of a corner is not well-defined for gray scale images.

How is intensity variation $V$ measured at a particular position $P$ in the image? Moravec proposed measuring the intensity variation by placing a small square window (typically, 3x3, 5x5, or 7x7 pixels) centered at $P$ and than shifting this window by one pixel in each of the eight principle directions (horizontally, vertically, and four diagonals). The intensity variation for a given shift is calculated by taking the sum of squares of intensity differences of corresponding pixels in these two windows. The following figure shows this calculation for a diagonal shift on an isolated black pixel (intensity equal to 0) on a white background (intensity equal to 255) and on an ideal corner. The red square indicates the original window and the blue box indicates the shifted window. Intensity variation at $P$ is the *minimum* intensity variation calculated over the eight principle directions.

$$V = \sum_{i} (I_i - I'_i)^2$$

$$V = \sum_{i} (I_i - I'_i)^2 = 2 * 255^2$$

$$V = \sum_{i} (I_i - I'_i)^2 = 3 * 255^2$$
To understand why the Moravec operator is a corner detector, the next figure shows windows at four different types of positions. Position A is interior to an object (or on the background) where it is assumed the image intensity will be relatively constant within the window, so shifting the window in any direction results in only a small intensity variation. For a window straddling an edge, as in position B, shifting the window perpendicular to the edge will result in a large intensity variation, but shifting it along the edge will result in only a small intensity variation. Both positions C and D, which respectively correspond to a corner and an isolated pixel, will give a large intensity variation for all shift directions. This shows that the Moravec operator is indeed a corner detector, but that it may be sensitive to detecting isolated pixels as corners. Note that having a large intensity variation in every direction is equivalent to the shift direction giving the minimum intensity variation being large.

<table>
<thead>
<tr>
<th>A. Interior Region</th>
<th>B. Edge</th>
<th>C. Edge</th>
<th>D. Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little intensity variation in any direction</td>
<td>Little intensity variation along edge, large variation perpendicular to edge</td>
<td>Large intensity variation in all directions</td>
<td>Large intensity variation in all directions</td>
</tr>
</tbody>
</table>

Moravec's corner detector is a relatively simple algorithm that was used by Moravec and others, but is now generally considered obsolete. It is not rotationally invariant (a property prevalent even in more modern corner detectors) as the response is anisotropic, is considered to have a noise response, and is susceptible to reporting false corners along edges and at isolated pixels so is sensitive to noise. However, it is computationally efficient which was critical for Moravec as he was interested in a real-time application and had minimal computational power at his disposal.

Due to the increase in computational power over the last few decades, most applications now employ corner detectors with better performance although at the cost of increased computation. One of the most widely used corner detectors today is the Harris/Plessey corner detector. This corner detector is computationally demanding, but directly addresses many of the limitations of the Moravec corner detector.
Harris/Plessey Operator

Introduction

This operator was developed by Chris Harris and Mike Stephens in 1988 [3] as a low-level processing step to aid researchers trying to build interpretations of a robot's environment based on image sequences. Specifically, Harris and Stephens were interested in using motion analysis techniques to interpret the environment based on images from a single mobile camera.

Harris and Stephens developed this combined corner and edge detector by addressing the limitations of the Moravec operator. The result is a far more desirable detector in terms of detection and repeatability rate at the cost of requiring significantly more computation time. Despite the high computational demand, this algorithm is widely used in practice.

The literature refers to this detector as both the Harris corner detector and the Plessey corner detector.

Omitting the derivation of the operator, we show the algorithm used to apply it.

Algorithm

The Plessey corner detector is stated formally below:

*Input:* grayscale image, Gaussian variance (window typically has a radius of 3 times the standard deviation), k value, threshold T

*Output:* map indicating position of each detected corner

1. For each pixel \((x, y)\) in the image calculate the autocorrelation matrix \(M\) using a small windows around the pixel:

\[
M = \begin{pmatrix} A & C \\ C & B \end{pmatrix}
\]

Where:

\[
A = \left( \frac{\partial I}{\partial x} \right)^2 \otimes w, \quad B = \left( \frac{\partial I}{\partial y} \right)^2 \otimes w, \quad C = \left( \frac{\partial I \partial I}{\partial x \partial y} \right) \otimes w
\]

\(\otimes\) is the convolution operator

\(w\) is the Gaussian window
2- Construct the cornerness map by calculating the cornerness measure $C(x, y)$ for each pixel $(x, y)$:

$$C = AB - C^2 - k(A + B)^2$$

Where $k$ is a constant.

Empirical testing has indicated that values of $k = 0.04$ to $0.06$ give the best results.

3- Threshold the interest map by setting all $C(x, y)$ below a threshold $T$ to zero.

4- Perform non-maximal suppression (find point which have a higher value than all the other points surrounding them) to find local maxima.

All non-zero points remaining in the cornerness map are corners.

Example image: point in green are detected corners.
Scale-invariant feature transform

Scale-invariant feature transform (or SIFT) is an algorithm in computer vision to detect and describe local features in images. The algorithm was published by David Lowe in 1999.

Applications include object recognition, robotic mapping and navigation, image stitching, 3D modeling, gesture recognition, video tracking, and match moving.

For any object in an image, interesting points on the object can be extracted to provide a "feature description" of the object. This description, extracted from a training image, can then be used to identify the object when attempting to locate the object in a test image containing many other objects. To perform reliable recognition, it is important that the features extracted from the training image are detectable even under changes in image scale, noise and illumination. Such points usually lie on high-contrast regions of the image, such as object edges.

Another important characteristic of these features is that the relative positions between them in the original scene shouldn't change from one image to another. For example, if only the four corners of a door were used as features, they would work regardless of the door's position; but if points in the frame were also used, the recognition would fail if the door is opened or closed. Similarly, features located in articulated or flexible objects would typically not work if any change in their internal geometry happens between two images in the set being processed. However, in practice SIFT detects and uses a much larger number of features from the images, which reduces the contribution of the errors caused by these local variations in the average error of all feature matching errors.

Lowe's patented method can robustly identify objects even among clutter and under partial occlusion, because his SIFT feature descriptor is invariant to scale, orientation, and affine distortion, and partially invariant to illumination change. This section summarizes Lowe's object recognition method and mentions a few competing techniques available for object recognition under clutter and partial occlusion.

The feature detection stage of SIFT will be omitted as it is not needed here (reasons are discussed in later sections). Only the SIFT descriptor for a given point of interest will be explained here.
The SIFT Descriptor:

Given a point of interest in an image, the SIFT descriptor first assigns a consistent orientation to each keypoint based on local image properties, the keypoint descriptor can be represented relative to this orientation and therefore achieve invariance to image rotation.

Following experimentation with a number of approaches to assigning a local orientation, the following approach was found to give the most stable results. For each image sample, $L(x, y)$ the gradient magnitude $m(x, y)$, and orientation $\theta(x, y)$ are precomputed using pixel differences:

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$

$$\theta(x, y) = \tan^{-1}\left(\frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)}\right)$$

An orientation histogram is formed from the gradient orientations of sample points within a region around the keypoint. The orientation histogram has 36 bins covering the 360 degree range of orientations. Each sample added to the histogram is weighted by its gradient magnitude and by a Gaussian-weighted circular window with a ___ that is 1.5 times that of the scale of the keypoint.

Peaks in the orientation histogram correspond to dominant directions of local gradients. The highest peak in the histogram is detected, and then any other local peak that is within 80% of the highest peak is used to also create a keypoint with that orientation. Therefore, for locations with multiple peaks of similar magnitude, there will be multiple keypoints created at the same location and scale but different orientations. Only about 15% of points are assigned multiple orientations, but these contribute significantly to the stability of matching.

The next step is to compute a descriptor for the local image region that is highly distinctive yet is as invariant as possible to remaining variations, such as change in illumination or 3D viewpoint.

One obvious approach would be to sample the local image intensities around the keypoint at the appropriate scale, and to match these using a normalized correlation measure. However, simple correlation of image patches is highly sensitive to changes that cause misregistration of samples, such as affine or 3D viewpoint change or non-rigid deformations. A better approach has been demonstrated by Edelman, Intrator, and Poggio (1997). Their proposed representation was based upon a model of biological vision, in particular of complex neurons in primary visual cortex. These complex neurons respond to a gradient at a particular orientation and spatial frequency, but the location of the gradient on the retina is allowed to shift over a small receptive field rather than being
precisely localized. Edelman *et al.* hypothesized that the function of these complex neurons was to allow for matching and recognition of 3D objects from a range of viewpoints. They have performed detailed experiments using 3D computer models of object and animal shapes which show that matching gradients while allowing for shifts in their position results in much better classification under 3D rotation. For example, recognition accuracy for 3D objects rotated in depth by 20 degrees increased from 35% for correlation of gradients to 94% using the complex cell model. The implementation described below was inspired by this idea, but allows for positional shift using a different computational mechanism.

The figure 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.

A Gaussian weighting function with \( \sigma \) 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 window on the left side of the figure. 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, as these are most affected by misregistration errors. The keypoint descriptor is shown on the right side of the figure. It allows for significant shift in gradient positions by creating orientation histograms 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 on the left can shift up to 4 sample positions while still contributing to the same histogram on the right, thereby achieving the objective of allowing for larger local positional shifts.
It is important to avoid all boundary affects in which the descriptor abruptly changes as a sample shifts smoothly from being within one histogram to another or from one orientation to another. Therefore, trilinear interpolation is used to distribute the value of each gradient sample into adjacent histogram bins. In other words, each entry into a bin is multiplied by a weight of \(1 - d\) for each dimension, where \(d\) is the distance of the sample from the central value of the bin as measured in units of the histogram bin spacing.

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 side of the figure. The figure shows a 2x2 array of orientation histograms, whereas our experiments below show that the best results are achieved with a 4x4 array of histograms with 8 orientation bins in each. Therefore, the experiments in this paper use a 4x4x8 = 128 element feature vector for each keypoint.

Finally, the feature vector is modified to reduce the effects of illumination change. First, the vector is normalized to unit length. A change in image contrast in which each pixel value is multiplied by a constant will multiply gradients by the same constant, so this contrast change will be canceled by vector normalization. A brightness change in which a constant is added to each image pixel will not affect the gradient values, as they are computed from pixel differences. Therefore, the descriptor is invariant to affine changes in illumination. However, non-linear illumination changes can also occur due to camera saturation or due to illumination changes that affect 3D surfaces with differing orientations by different amounts. These effects can cause a large change in relative magnitudes for some gradients, but are less likely to affect the gradient orientations. Therefore, we reduce the influence of large gradient magnitudes by thresholding the values in the unit feature vector to each be no larger than 0.2, and then renormalizing to unit length. This means that matching the magnitudes for large gradients is no longer as important, and that the distribution of orientations has greater emphasis. The value of 0.2 was determined experimentally using images containing differing illuminations for the same 3D objects.