Chapter 4

Image Processing

4.1 Thresholding and Binary Image Processing

4.1.1 Creating binary images through thresholding

Binary images contain the least information per pixel that we can sensibly assign – one bit. Each pixel is either black or white, and binary images can thus be represented by zero-one matrices. By convention, in these matrices, the value 1 stands for an object pixel and is shown in black, and the value 0 indicates a non-object (background or hole) pixel, shown in white.

Given the ability of modern graphics cards to display at least 16 million different colors and the steadily increasing memory and processing capacities of today’s computers, using black-and-white representations of visual images seems to be an outdated concept. However, this impression is far from reality. While it is certainly true that using binary images would be ill-advised for some computer vision problems such as face recognition or natural scene understanding, other tasks can benefit from it. Whenever we are interested in the contours, positions, sizes, or numbers of non-overlapping objects that can be separated from their background, binary images are our ideal choice. Although these conditions seem to rarely occur in natural scenes, we can often create them for specific computer vision applications.

For instance, imagine a tool factory whose machines produce a certain proportion of misshaped tools. A computer vision system could be used to inspect the newly made tools as they are traveling on a conveyor belt and quickly decide which ones show shape anomalies and should be sorted out. For this system, lighting conditions could be arranged in such a way that – ideally – the intensity (brightness) of image pixels belonging to the tools clearly differs from the intensity of background pixels. This setup would allow us, using a simple thresholding operation as described below, to represent the tools’ contours using binary images. Another example is the counting of bacteria in microscopic
images. In order to study the growth of bacterial cultures, it is sometimes necessary to obtain such a count. For a researcher, this is a very tedious and error-prone task. Using appropriate backlighting of the slide and digitization of the resulting images, it is possible to separate objects (bacteria) and background in terms of pixel intensity. Using thresholding, we can once again generate binary images to represent this information. Subsequently, an efficient algorithm, such as the one presented in Section 4.1.2, can be deployed to count the individual objects in a given image.

Besides their efficient use of memory, the main advantage of binary images is that simple and highly efficient algorithms can be applied to them in order to analyze object contours. Many of these algorithms are only applicable to binary images, exploiting their simple on-or-off representation of objects. When designing a computer vision application, it is thus advisable to consider whether at least a part of the problem could be addressed most efficiently using binary images.

In the ideal case, the object pixels in our original image $A$ are either darker or brighter than all other pixels, and we can use a single threshold $\theta$ to create the binary image $A_\theta$. Let us assume that the object pixels are darker than the background. Then $A_\theta$ can simply be generated as follows:

$$A_\theta[i, j] = \begin{cases} 1, & \text{if } A[i, j] \leq \theta \\ 0, & \text{otherwise} \end{cases}$$

How should we choose the value of $\theta$? Let us take a look at the example image shown in Fig. 4.1. Obviously, the pixels of the statue tend to be darker than those of the sky in the background. This is an almost ideal case for applying a single threshold. Due to the clear separation in brightness between object and background, we could simply estimate a threshold value and would quickly find an appropriate one.
A more systematic approach to determining $\theta$ is based on the analysis of the intensity distribution in the original image. With a few lines of code, we can count the number of pixels for each of the 256 intensity levels that can typically occur in grayscale images. For images that are well-suited for object-background segmentation, we expect a bimodal shape of the resulting intensity histogram. Fig. 4.2 shows the intensity histogram for the picture in Fig. 4.1.
Clearly, the image pixels are largely separated into two groups, with the darker ones presumably belonging to the statue and the brighter ones to the sky. While it is barely visible in Fig. 4.2, it is important to note that this separation is not perfect, that is, the number of pixels per intensity level never drops to zero between the two distinct pixel groups. The smallest value in this range is found at intensity level 180, which includes 475 out of the total 5.5 million image pixels. Consequently, this level is our best guess for the optimal threshold, and we set $\theta = 180$. The binary image that results from thresholding at this level is shown in Fig. 4.3.

![Image](image.png)

Fig. 4.3 Result of thresholding the image in Fig. 4.1 using a threshold $\theta = 180$.

The image looks perfect except for a few tiny light reflections on the statue whose intensity could not be separated for that of the background, leading to white pixels within the object. Nevertheless, as noted above, our example image is a near-optimal case for a real-world, outdoor photograph. In other cases, not all background pixels may be brighter than the object, but some may be darker than the object. If this situation is reflected in a trimodal intensity distribution, then we could determine the two thresholds, $\theta_1$ and $\theta_2$ with $\theta_1 < \theta_2$, that yield the best separation of the three pixel groups. The binary image $\mathbf{A}_\theta$ is then obtained as follows:
In other cases, the distribution of intensity values in object and background may be less favorable, but it may still be possible to identify a set \( L \) of intensity levels that are mostly associated with object pixels. In that case, we can generate the binary image \( A_L \) using the following equation:

\[
A_L[i, j] = \begin{cases} 
1, & \text{if } \theta_1 \leq A[i, j] \leq \theta_2 \\
0, & \text{otherwise}
\end{cases}
\]

If the separation of objects and background is difficult, additional heuristics as well as spatial information from the image can be considered. A variety of approaches for determining object contours that can be used for binarizing color or grayscale images will be discussed in Chapter 6 in the context of image segmentation.

### 4.1.2 Connected components in binary images

In the previous example, we only wanted to separate one object from its background. Often, however, an image contains multiple objects that need to be separated from both the background and from each other. To accomplish this, we first generate a binary image as described in the section above. Subsequently, an algorithm labels the individual objects in the binary image. Before we can devise an efficient algorithm for this purpose, we need to introduce a few important concepts in binary image processing.

First, the concept of the neighborhood of a pixel needs to be introduced. There are actually two common ways to define the neighborhood: the 4-neighborhood (or 4-connection) and the 8-neighborhood (or 8-connection). Whereas the former concept only considers the four neighbors above, below, to the left, and to the right of a pixel, the latter also includes the four diagonal neighbors (see Fig. 4.4).
For a pixel \([i, j]\) in an image, …

\[
\begin{array}{c|c|c}
[i-1,j] & [i,j] & [i+1,j] \\
\hline
[i,j-1] & [i,j] & [i,j+1] \\
\hline
[i+1,j-1] & [i+1,j] & [i+1,j+1] \\
\end{array}
\]

…these are its 4-neighbors (4-neighborhood or 4-connection),

\[
\begin{array}{c|c|c|c}
[i-1,j-1] & [i-1,j] & [i-1,j+1] \\
\hline
[i,j-1] & [i,j] & [i,j+1] \\
\hline
[i+1,j-1] & [i+1,j] & [i+1,j+1] \\
\end{array}
\]

…and these are its 8-neighbors (8-neighborhood or 8-connection).

Fig. 4.4 Neighbors of pixel \([i, j]\) in the 4- and 8-neighborhoods.

Based on the definition of neighborhood, we can adapt the notion of a path from graph theory to the field of image processing. Here, pixels correspond to vertices, and neighboring pixels correspond to vertices that are connected by an undirected edge. Consequently, a path is defined as follows:

**Definition 4.1** A path from a pixel \([i_0, j_0]\) to a pixel \([i_n, j_n]\) is a sequence of pixels \([i_0, j_0], [i_1, j_1], [i_2, j_2], \ldots, [i_n, j_n]\) in which pixels \([i_p, j_p]\) and \([i_{p+1}, j_{p+1}]\) are neighbors for all \(0 \leq p < n\).

In other words, if we travel on an image by only making transitions between neighboring pixels, the sequence of our steps forms a path. The are two important points to be noted: First, by definition we are not limited to moving along 1-pixels (object pixels); the path can also include 0-pixels or even entirely consist of them. Second, the definition of a path depends on whether a 4-neighborhood or an 8-neighborhood is considered, resulting in a 4-path or an 8-path, respectively. Fig. 4.5 provides one example for each case. It follows from the definitions of neighborhood and path that every 4-path is also an 8-path, whereas the reverse is not always true.
The foreground of a binary image is defined as the set $S$ of all its 1-pixels. Whenever there is a path from a pixel $p \in S$ to a pixel $q \in S$ that entirely consists of pixels in $S$, we say that $p$ and $q$ are connected. Importantly, pixel connectivity is an equivalence relation on $S$. As a reminder, an equivalence relation is a binary relation that is reflexive, symmetric, and transitive, which are properties that any sensible definition of equivalence needs to have. Clearly, connectivity is reflexive, as every 1-pixel is connected to itself, symmetric, because $p$ is connected to $q$ if and only if $q$ is connected to $p$, and transitive, since connections between $p$ and $q$ and between $q$ and $r$ imply a connection between $p$ and $r$.

Equivalence relations have the interesting property of partitioning their domain into disjoint subsets (partitions) whose union includes the entire domain. In other words, an equivalence relation $R$ cuts its domain into smaller pieces so that all elements within the same piece are related to each other under $R$, and elements from different pieces are never related to each other under $R$. In the case of our connectivity relation, these pieces or partitions are groups of pixels in $S$ that are connected to one another. We refer to them as connected components. Loosely speaking, connected components are the objects in a binary image, that is, individual patches of 1-pixels that are separated from each other by 0-pixels. For example, ideally, in a binarized microscopic image, each connected component corresponds to an individual bacterium.

Similarly, we can define connected components in $\overline{S}$, the complement of $S$, which is the set of all 0-pixels. Here, two pixels in $\overline{S}$ are connected whenever there is a path between them that entirely consists of pixels in $\overline{S}$. The set of all connected components in $\overline{S}$ that include points on the border of the image is called the background. All other connected components in $\overline{S}$ are referred to as holes. For example, the image in Fig. 4.3 above contains, as far as we can tell, only one large object. It also contains a large background that spans the space.
above the object and to its left and right. Furthermore, there are two large holes that are formed by the horse’s legs and the pedestal, as well as many little holes throughout the object.

In another example, Fig. 4.6 shows an extreme case of a 9-pixel image illustrating the impact of the choice of neighborhood on the resulting objects and holes. According to the 4-neighborhood, the image contains four objects and one hole. However, if we assume an 8-neighborhood, we find one object and no hole. Besides being ambiguous, these results are also contradictory. If one interprets the four 1-pixels as individual objects, the 0-pixel in their center is not a hole in any object but belongs to the background. On the other hand, if the four 1-pixels are assumed to belong to the same object, then the central 0-pixel should be considered a hole. To obtain more consistent results, we can use the 4-neighborhood for the foreground and the 8-neighborhood for background. The example in Fig. 4.6 is then interpreted as four objects and no hole. In the following, we will always use this principle.

There are a few more notions that need to be introduced: The pixels in $S$ that have at least one 4-neighbor in $\overline{S}$ form the set $S'$, referred to as the boundary of $S$. All pixels in the set $(S - S')$, that is, those that are in $S$ but are not in $S'$, are called the interior of $S$. Finally, the set $T$ of pixels is said to surround $S$ if any 4-path from any pixel in $S$ to the boundary of the image includes at least one pixel in $T$. Fig. 4.7 illustrates these concepts by example.
Having studied the idea of connected components as objects in binary images, let us now return to the problem of identifying and labeling all connected components in a given image. We would like to devise an efficient algorithm that assigns a unique numerical label to each connected component and enters it in a matrix of the same size as the image, as shown in Fig. 4.8. This would allow subsequently applied computer vision algorithms to distinguish among the objects in the image.

A straightforward, naïve approach to this problem is the following algorithm:

**Algorithm 4.1 Naïve component labeling**

1. *Scan the image systematically until an unlabeled 1-pixel is found, and assign it a new label L. If no such pixel is found, terminate.*

2. *Starting at the same pixel, use a recursive procedure to assign label L to the set P of all unlabeled 1-pixels in its 4-neighborhood and then call itself for each pixel in P.*

3. *Go to Step 1.*

Unfortunately, this recursive algorithm is rather inefficient, which could be problematic for applications such as real-time processing of high-resolution video. To solve this problem, we could simply scan the image in a left-to-right, top-to-bottom manner and, as before, assign a new label if we encounter an unlabeled pixel. However, if the current pixel is a 4-neighbor of a previously labeled pixel, that is, if its upper or left neighbors are already labeled, we assign the same label to the current pixel. This way we could label all connected components within one linear scan.
There is one case, however, that requires special treatment: If a connected component contains a pixel in row $r$ that is further left than any of its pixels in row $(r - 1)$, the algorithm will assign a new label to this pixel. Consequently, pixels of the same object will have different labels. For example, we face this problem when entering the second row of the bottom component in Fig. 4.8. To resolve this issue, we can wait for the inevitable situation of encountering a pixel of the same object whose upper and left neighbors have different labels. Then we copy the upper neighbor’s label onto the current pixel and mark both neighbors’ labels as equivalent in an equivalence table. This table could be set up in various ways, for example as a two-column table with one row for every label used in the image. Its first column contains an individual label $L$, and the second column holds the set of all labels that occur in the same object as $L$. Using this table, we can perform a second linear scan of the image and replace its labels to assign all pixels of the same object the same, unique label. This algorithm can formally be described as follows:

**Algorithm 4.2 Efficient component labeling**

1. **Scan the image systematically from left to right and top to bottom until an unlabeled 1-pixel is found.** If no such pixel is found, go to Step 4.
2. **Label the current pixel based on its upper and left neighbors:**
   
   a. If only one of them is labeled, or they both have the same label, then copy the label.
   b. If they have different labels, copy the upper neighbor’s label and enter the two labels in the table as being equivalent.
   c. If neither of them is labeled, assign a new label and enter it in the table.
3. **Go to Step 1.**
4. **For each label in the table, determine the lowest equivalent label.**
5. **Scan the picture once again while replacing each label with its lowest equivalent.**

Note that Algorithm 4.2 may not assign consecutive numbers to the objects in the same image. For example, an image contains five objects may receive the labels 1, 3, 7, 10, and 15. This is undesirable, for instance, for applications whose
purpose is the counting of objects. To generate consecutive labels, the set of lowest labels in each row of the equivalence table could be ordered. In the second scan of the image in Algorithm 4.2, its labels could then be replaced with the positions of the lowest equivalent labels in their ordered sequence.

### 4.1.3 Removing noise from binary images

Most digital images contain a certain amount of noise due to imperfect sensors, dust particles, or a variety of other factors. In binary images, obviously, noise can only have two possible effects: Pixels that ideally should be 0-pixels are shown as 1-pixels (positive noise) or 1-pixels that should be 0-pixels (negative noise).

Let us consider these effects for one of the most common problems in computer vision, optical character recognition (OCR). When scanning written text, the most common sources of noise are tiny particles on the paper and local inconsistencies in its surface leading to gaps during printing, scanning, or both. Both factors tend to create local patches of positive or negative noise rather than large areas of incorrect pixel values. Fig. 4.9a shows the example of a question mark affected by local spots of both positive and negative noise.

![Fig. 4.9](image.png)

Fig. 4.9  (a) Binarized question mark character in Arial font containing both positive and negative noise; (b) same image after applying a size filter with threshold $\tau = 10$; (c) original image after applying a size filter with threshold $\tau = 20$, which is too high for the given image.

One common technique for reducing this type of noise is the application of a size filter. A size filter first applies a component labeling algorithm, such as Algorithm 4.2, to the input image. Then it performs an additional scan of the
image in which it computes the area of each component, i.e., the number of 1-pixels it includes. If this number is below a given threshold $\tau$, these 1-pixels are turned into 0-pixels. As a consequence, all local patches of 1-pixels below a certain size will be removed, which removes much of the positive noise from the image if the threshold $\tau$ is chosen appropriately. In order remove negative noise, we can subsequently label the connected components in $\hat{S}$ and replace those smaller than $\tau$ with 1-pixels. Rather than writing two variants of the size filter for positive and negative noise, we could simply invert the image, i.e., switch 0-pixels and 1-pixels, after positive noise filtering, then run the same size filter algorithm again, and invert once more. Fig. 4.9b shows the effect of applying a size filter with $\tau = 10$ to the original, noisy image. The noise removal worked almost perfectly, except that the size filter cannot repair any “damage” to the contour of the character such as the one on the lower right of the question mark. Fig. 4.9c demonstrates that we have to be careful when setting the threshold $\tau$. In the given example, applying a size filter with $\tau = 20$ removes an essential part of the question mark character.

Another approach for removing noise from binary images is based on the expanding and shrinking operations, which are defined as follows:

**Definition 4.2** Expanding operation: For all pixels in the image, change a pixel value from 0 to 1 if any of its neighbors are 1-pixels.

**Definition 4.3** Shrinking operation: For all pixels in the image, change a pixel value from 1 to 0 if any of its neighbors are 0-pixels.

Typically, these operations use the 4-neighborhood. They have the interesting property of, loosely speaking, being “approximately” inverse operations of each other. In other words, if we expand a binary image and then shrink it, the result will often look similar to the original image, unless the expanding operation fused some objects that could not be divided through shrinking. As a side effect, small holes in the objects may have been filled during expansion and become invisible to the shrinking operation, leading to their removal. Fig. 4.10 demonstrates this effect on the noisy question mark example.
The result looks nice with even a slight reduction in the gap on the lower right of the question mark. Unfortunately, the positive noise is still present. Similarly to using a sequence of expanding and shrinking for negative noise removal, a sequence of shrinking and expanding should eliminate some of the positive noise. While this approach is generally feasible, we have to be careful, especially when the image has low resolution and contains fine lines. To illustrate this, Fig. 4.11 shows the effect of a shrinking operation on the original noise question mark image. Since this question mark contains holes, shrinking enlarges these holes to such an extent that no expanding operation or sequence of them could restore the original shape of the character.
Under such conditions, it is thus advisable to first perform the expanding-shrinking sequence, followed by a shrinking-expanding sequence. Fig. 4.12a shows once again the result of expanding and subsequent shrinking of the original noisy question mark character. If we perform shrinking (Fig. 4.12b), followed by expanding (Fig. 4.12c) on this image, we effectively remove the negative noise without severely damaging the question mark character.
Furthermore, Fig. 4.12c also illustrates that expanding and shrinking are, even in the absence of noise, not perfectly inverse to each other. While the resulting question mark image is almost free of noise, its shape has slightly changed. This effect is most clearly visible for the dot at the bottom, which changes from a square into a disc. Fortunately, in higher-resolution images, this effect is less pronounced. On the other hand, higher resolution may also include patches of noise that are too big to be removed with a single expanding or shrinking operation. In that case, we can use sequences that include multiple consecutive applications of expanding (E) and shrinking (S). For example, we could apply the sequence E-E-S-S-S-E-E to remove even large patches of positive and negative noise. It is obviously important for applying such a sequence that the resolution of our image and the line width of objects are sufficiently high. If we tried it on our example image, the result would be catastrophic.

4.1.4 Basic techniques for object standardization in binary images

Let us assume that we have a template of an object in the form of a binary image, and our task is to devise an algorithm that decides for other binary images whether they contain the same object or not. Such an algorithm would be useful, for example, for some initial quality control of manufactured tools being photographed while passing by on a conveyor belt. A rather naïve approach to solving this problem could be a comparison of the template and the input image in terms of individual pixels in corresponding locations. For example, the algorithm could count the number of mismatches, that is, a 1-pixel in one image and a 0-pixel in the other at the same position. If the count is below a certain threshold, the algorithm determines that the object is present, and otherwise that it is absent.

This approach could be successful if it is certain that the object will always appear in exactly the same size, position, and orientation. For most object recognition tasks this is not guaranteed, and therefore, the algorithm needs to be improved. One way of accomplishing invariance to the parameters of size, position, and orientation could be the generation of a large number of templates, each showing the object for a different set of parameters. If enough templates are created to densely cover the parameter space, an algorithm could find the best pixel-by-pixel match of any template with the input image and then use a threshold to make its decision. For some applications, this method could succeed
but would be extremely inefficient if a large parameter space, i.e., many different combinations of parameter values, had to be covered.

A more efficient technique for achieving parameter invariance is the standardization of a binary object image. The basic idea underlying this method is to transform both the template and the input images to have the same size, position, and orientation. Afterwards, a simple pixel-by-pixel comparison can be used to decide about the presence of the template object in the input image.

The area $A$ of an object, representing its size, is defined as the number of 1-pixels that it includes. To standardize an object of area $A$ to have an approximate area $A_0$, we can linearly scale the original image $O$ both horizontally and vertically by a scaling factor $s$, which is given by:

$$s = \sqrt{\frac{A_0}{A}} \quad (4.1)$$

The scaled $M\times N$ image $P$ can then be computed as:

$$P[i, j] = O[\lfloor i/s \rfloor, \lfloor j/s \rfloor] \quad \text{for } i = 0, ..., (M - 1) \text{ and } j = 0, ..., (N - 1), \quad (4.2)$$

where $M$ and $N$ have to be chosen sufficiently large to contain the resulting image. For low-resolution images, additional rounding and averaging schemes could be used to obtain better results in some cases. Note that this scaling operation will not usually generate an object of exactly area $A_0$ but only a close approximation, which is sufficient for most applications.

In the next step, we would like to standardize the position of the object. A common way of accomplishing this is to translate the object so that its center of gravity coincides with the center of the image. The center of gravity (or center of mass) $[i_c, j_c]$ of the object has the following property: If we printed the object on paper and cut it out, we could ideally balance it in horizontal orientation on the tip of a pencil if that pencil touched it at position $[i_c, j_c]$. (Note, however, that $[i_c, j_c]$ might lie outside the object, as in the letter “O.”) The center of gravity can be computed as the arithmetic means of the x- and y-coordinates of all object pixels in image $P$:

$$i_c = \frac{\sum_{i=0}^{M-1} \sum_{j=0}^{N-1} i \cdot P[i, j]}{A_p} \quad \text{and} \quad j_c = \frac{\sum_{i=0}^{M-1} \sum_{j=0}^{N-1} j \cdot P[i, j]}{A_p}, \quad (4.3)$$
where $A_P$ is the area of the object in image $P$. Afterwards, $i_c$ and $j_c$ can be rounded to the nearest integer and used to shift the object in such a way that its center of gravity lies in the center $[i_Q, j_Q]$ of a new image $Q$:

$$Q[i, j] = P[i + i_c - i_Q, j + j_c - j_Q]$$  (4.4)

Here, pixels outside of $P$ should be regarded as 0-pixels. After this translation, the final step will be the rotation of the object into a standardized orientation. For this purpose, we determine the object’s axis of greatest elongation, that is, the direction in which it extends the most. A simple method for accomplishing this is to systematically test different candidate axes, all of which pass through the object’s center of gravity at different angles. The axis with the minimum sum of squared distances from all 1-pixels is taken as the axis of greatest elongation. In the case of an elongated object such as a screwdriver, this axis best describes its orientation in the image plane. We can now standardize the orientation of the object by rotating it in such a way that its axis of greatest elongation is vertical. For an angle $\alpha$ between the axis and the vertical direction, the object is rotated by mapping it onto a new image $R$ of the same size as $Q$ as follows (cf. rotation matrices, Chapter 3):

$$R[i, j] = Q[i_Q + (i - i_Q)\cos \alpha - (j - j_Q)\sin \alpha, j_Q + (i - i_Q)\sin \alpha + (j - j_Q)\cos \alpha]$$  (4.5)

After this standardization, pixel-by-pixel matching of template and input image can be applied to decide about the presence of the target object in the image. Note that the axis of greatest elongation can be aligned with the vertical direction in two different ways; for example, if the angle between the axis and the vertical is $30^\circ$, we can rotate the object by $-30^\circ$, as described in Eq. 4.5, or we could rotate it by $150^\circ$. Therefore, it may improve the performance of our object recognition algorithm if it finds the best match of the template with both the image resulting from Eq. 4.5 and a version of it that is rotated by $180^\circ$ around the image center.

### 4.2 Intensity Transformation

When we intend to work with intensity resolutions greater than 1-bit, usually the first step in image preprocessing is an intensity transformation. This is the simplest and most limited kind of transformation, because it is point-based: The intensity of a pixel in the transformed image only depends on the intensity of the
same pixel in the original (input) image. In order to determine the intensity mapping function that is applied to every individual pixel, information about the intensity distribution across the input image is used. The purpose of such transformation is to improve the visibility of the image content to human observers, to prepare the data for processing by a particular algorithm, or both.

4.2.1 Gamma transformation

In Section 2.2, we briefly discussed the problem of mapping the vast dynamic range of real-world light intensity onto a limited number of discrete levels in a digital representation. The common solution to this problem is the use of a power law, a so-called gamma transformation. It accommodates for both the dynamic range and the corresponding variation in human perceptual sensitivity. However, for some data the usual amount of gamma correction is insufficient. To demonstrate this, let us first consider the gamma transform in the general formulation

\[ I' = n \left( \frac{I}{m} \right)^\gamma, \]

where \( I \) and \( I' \) are the original and transformed intensities with maximum values \( m \) and \( n \), respectively, and the exponent \( \gamma \) determines the shape of the transformation’s non-linearity. The equation is kept simple by assuming that the minimum value is zero for both \( I \) and \( I' \). Fig. 4.13 illustrates this function for several values of \( \gamma \) determines. Decreasing \( \gamma \) below 1 leads to an increasing compression of high intensity input values in the transformed output. Vice versa, increasing \( \gamma \) above 1 causes a more and more pronounced compression of low intensity input values.
Furthermore, the gamma transformation can easily be inverted by a transformation with parameters $m' = n$, $n' = m$, and $\gamma' = 1/\gamma$. If we apply this transformation to intensity $I'$ as obtained in Eq. 4.6, the result is intensity $I''$, which turns out to be identical to the original intensity $I$:

$$I'' = m \left( \frac{I'}{n} \right)^{1/\gamma} = m \left( \frac{I}{m} \right)^{1/\gamma} = m \frac{I}{m} = I. \quad (4.7)$$

Consequently, in Fig. 4.13, the functions for $\gamma$-values 0.05, 0.2, and 0.5 are inverse to those with $\gamma$-values 20, 5, and 2, respectively. Knowing this general gamma transform and its inverse, we can now even transform data of immense
dynamic ranges to visualize them for human observers or prepare them for processing by computer vision algorithms. To give an extreme example, let us assume that we want to visualize Pascal’s triangle in the form of intensity data. This data structure contains in its top row a single 1 and infinite numbers of 0s to its left and right. The numbers in lower rows are always given by the sum of the two numbers above them. If we disregard the 0s, the first five rows look at follows:

\[
\begin{array}{c}
1 \\
1 \\
1 \\
1 \\
1 \\
\end{array}
\]

While the downward increase of values seems slow at first, it speeds up very quickly. Since the horizontal sum of numbers doubles from row to row, the numbers follow an exponential growth pattern as we move down the triangle. If we want to visualize a triangle with 50 rows, the maximum value to account for is greater than 63 trillion or \(6.3 \times 10^{13}\). If we simply use linear scaling to map this pattern onto a scale of 256 displayable gray levels, only the greatest values at the bottom of the triangle are visible (see Fig. 4.14a). Note that increasing the intensity resolution of the image would not solve this problem, because the human visual system could not distinguish the subtle differences in gray levels across most of the image.
Fig. 4.14 Grayscale visualization of the first 50 rows of Pascal’s triangle using (a) linear scaling and (b) gamma transformation with $\gamma = 0.065$.

Using gamma transformation instead of linear scaling yields clearly better results. The image in Fig. 4.14b was obtained by applying Eq. 4.6 with parameters $m = 6.33 \times 10^{13}$, $n = 255$, and $\gamma = 0.065$ to the raw input data. In this visualization, the triangular pattern of values and their relative sizes can easily be recognized.

The data from Pascal’s triangle are certainly an extreme example of a nonlinear distribution. However, they are not unrealistic, as some common operations in computer vision such as the Fourier transform often generate comparable distributions. Even for less extreme cases, the gamma transformation can be useful for better representing visual information. For example, as shown in Fig. 4.15 (left), when we take a photo of a scene while directing the camera at the brightest spot, the camera’s automatic adjustment of intensity levels may lead to most of the image being too dark for recognizing objects in it. This effect is reflected in the intensity histogram being skewed towards darker levels. In such cases, a “milder” gamma transformation can improve the overall clarity of the image. Using a transformation with $m = 255$, $n = 255$, and $\gamma = 0.5$, the result in Fig. 4.15 (right) is obtained. The dark areas have been mapped onto a wider intensity interval, making individual objects more distinguishable. At the same time, the intensity of the window area increased only slightly.
Fig. 4.15 Effect of a gamma transformation with $\gamma = 0.5$ on an image whose intensity histogram is skewed toward dark levels. Left panel: original image and histogram; Right panel: transformed image and histogram.

Fig. 4.15 also demonstrates that the gamma transformation puts “gaps” into the intensity histogram. Since there is only a limited number of intensity levels in the original image, the part of the histogram that is “stretched” will receive holes, while some individual levels of the “compressed” part will show strongly increased proportions of pixels. Gamma transformation is a straightforward but usually not ideal method to optimize the distribution of gray levels in a given digital image. Modern photo editing software uses more sophisticated methods that are beyond the scope of this book. For computer vision applications, such fine-grained optimization is not necessary. We have all the necessary tools for intensity transformation once we are familiar with the gamma transformation and linear histogram scaling, which will be explained in the following section.

4.2.2 Linear histogram scaling

When we applied a gamma transformation to the image in Fig. 4.15 above, the situation was difficult, because its pixel intensities spanned the entire displayable range. Therefore, a non-linear intensity transformation was required. Other problems are easier to tackle - the most common one being the limitation of intensity values to a small interval within the range of available gray levels. Such images have low contrast and are often overly dark or bright. They can be
difficult to visually examine and cannot serve as input to computer vision algorithms that expect the full range of intensities to be utilized.

In these cases, \textit{linear histogram scaling} is the simplest and often the best solution. In order to perform this scaling, we first determine the minimum intensity $I_{\text{min}}$ and maximum intensity $I_{\text{max}}$ in the original image. Let us refer to the desired intensity range in the output image as $[a, b]$; in many cases, it will be $[0, 255]$. Then the following equation is used to map the intensity $I$ of every pixel in the input image onto intensity $I'$ of the corresponding pixel in the output:

$$I' = \frac{b - a}{I_{\text{max}} - I_{\text{min}}} (I - I_{\text{min}}) + a \quad (4.8)$$

In order to obtain better results in some situations, it is recommended not to use the very darkest and brightest pixels in the image to determine $I_{\text{min}}$ and $I_{\text{max}}$, respectively. The image could contain individual outliers, for example, pixels of intensity 0 or 255, whereas almost all other pixels are in a much narrower intensity range. In such a case, Eq. 4.8 would be ineffective. Therefore, we should choose $I_{\text{min}}$ as the greatest intensity level so that the number of pixels with lower intensities remains below a given threshold $\theta$. Similarly, $I_{\text{max}}$ is chosen as the least intensity level so that the number of pixels with higher intensities remains below the same threshold $\theta$. A typical choice for $\theta$ is 0.5% of the image pixels. By using this threshold, a maximum of 1% of all pixels will lie outside of the interval $[I_{\text{min}}, I_{\text{max}}]$ and thus outside of $[a, b]$ in the output image. Intensities below $a$ or above $b$ are set to $a$ and $b$, respectively. This method leads to a nonlinear mapping of at most 1% of the pixels and makes the histogram scaling more robust against the influence of intensity outliers.

Fig. 4.16 shows the effect of linear histogram scaling on two sample images. In both cases, contrast and clarity are substantially improved. Similar to the gamma transformation, linear histogram scaling also leads to gaps in the intensity histograms. As mentioned above, more advanced techniques for histogram optimization exist but are less relevant for the field of computer vision.
Fig. 4.16 Effect of linear histogram scaling on a low-contrast image (top left) and an overexposed photo (bottom left) with threshold $\theta = 0.5\%$. The output images are shown on the right, and an intensity histogram is displayed below each image.

4.3 Spatial Filtering

The intensity transformations that we explored in Section 4.2 were point-based, i.e., for any given pixel they did not depend on the intensities of its neighbors. For most purposes, even in image preprocessing, operations on images need to consider the neighborhood of a pixel. Preprocessing often considers only a small neighborhood, sometimes as small as four pixels, and more often including nine or 25 pixels. Such operations perform smoothing, sharpening, noise removal, or more advanced processing steps such as the detection of local edges. Later
processing stages in a computer vision system often integrate the output from earlier stages across wider areas in the input image.

To some extent, this computational scheme resembles the human visual processing hierarchy. In the retina and the low-level processing areas in the brain, each individual neuron only “sees” a small cutout of the visual field, known as its receptive field. In other words, the neuron’s activity will depend only on signals relating to its receptive field. Even in area V1 in visual cortex, receptive fields do not exceed a diameter of 1° of visual angle, which is approximately the size of our thumbnail when we straighten our arm. In the following sections, we will study the implementation of such local functions in a technical vision system.

4.3.1 Linear filtering: correlation and convolution

Let us first study a limited but important class of functions for implementing local operations on images – the linear functions. We would like to develop a scheme for applying any linear function to the neighborhood of a pixel \( p \) that outputs a singular scalar value. This output will become the intensity of the corresponding pixel \( p' \) in the output image.

The intensities of pixels around \( p \) can be considered to form a vector. If we define the neighborhood of \( p \) as its 8-neighborhood, then the resulting vector consists of nine elements – the eight neighbors and \( p \) itself. As we know from Chapter 3, any linear function that maps a vector \( v \) onto a scalar can be represented by a multiplication (inner product) of \( v \) with a vector of the same size. This means that each intensity value in the neighborhood of \( p \) is multiplied by a specific factor, and the sum of these products is the output of the operation. This output is entered as the intensity of pixel \( p' \) in the output image. In order to compute the entire output image, we have to process each pixel in the input image in the same way, using the same vector for multiplication. With this procedure, the intensity at each position in the output image is a linear function of the intensities in its neighborhood, including itself, in the input image. By varying the multiplicative vector, we can apply any linear function to a given image.

To illustrate this principle, let us choose a small neighborhood that only includes pixel \( p \) and its eight immediate neighbors. These values form a nine-element vector, which is multiplied with a vector of the same size that represents the linear function to be applied to the image. It is more intuitive and practical to think of this process as using an array of the same size of the neighborhood, i.e.,
3×3 pixels in our example. This array, referred to as a filter, contains the numbers by which each of the intensities in the image is to be multiplied. The procedure is carried out by shifting this filter across the input image and entering the resulting values into the corresponding positions in the output image (see Fig. 4.17).

Most filters have an odd number of rows and columns so that one pixel in the center can serve as its “anchor,” i.e., indicate the position for storing the output value. Fig. 4.17 also illustrates a problem with this method: If we require the entire filter to be placed entirely within the input image, the anchor pixel cannot reach every pixel in the same-size output image. For our 3×3 filter, this means that the first and last rows and the first and last columns cannot receive output values. Some literature recommends reducing the size of the output image accordingly. In our example, this would lead to a 3×3 output image for a 5×5 input. However, we often want to relate the results of our filter operations and other algorithms to the original image. For instance, when we develop an object detection algorithm, we would like to know where in the original image the object is located. If the size of the image changes with every operation, we would have to keep track of the correspondence between pixels in the original image and each of the images derived by filtering or other operations. Another solution is to allow parts of the filter to leave the image and only perform the multiplications for those cells that are superimposed on image pixels, possibly followed by some normalizing computation. The problem with this approach is that it may lead to an entirely different kind of operation being performed near the borders of the image.

Therefore, in this book we will use the following approach: The output image is identical in size to the input image, the filter is always completely inside the image, and those pixels in the output image that cannot be filled with values will be set to zero. This method avoids the problems above but will result in a black frame in the output image. If necessary, we can keep track of how wide this frame is at any time to treat these areas appropriately in further processing.
Chapter 4: Image Processing

Fig. 4.17 Illustration of applying a 3×3 linear filter to a 5×5 input image, resulting in a 5×5 output image with zeroes in its border pixels. (a) The filter is placed in the upper-left corner of the input image, and summing the products of the matching values of image and filter yields the first output value. (b) After shifting the filter across the entire input image and computing output values, the process ends with the filter being in the lower-right corner.

The operation described above is called correlation. While most textbooks and research literature refer to it as convolution, this is incorrect. The difference between correlation and convolution is subtle but important for our understanding of this fundamental operation in image processing and computer vision. While correlation seems to be the more straightforward approach, convolution has the more desirable mathematical characteristics. Moreover, if we think of these operations as a type of local multiplication of two functions or images, the result of the correlation has a particular drawback. Let us
demonstrate this using a larger input image and an asymmetric filter as shown in Figs. 4.18a and b.

![Input Image](image1)

**Fig. 4.18** Demonstration of the output differences between the correlation and convolution operations. (a) Input image; (b) Filter; (c) Result of correlation of input image and filter; (d) Result of convolution of input image and filter. In all panels, nonzero values are emphasized with gray backgrounds.

For each of the few nonzero pixels in the input, we expect to find in the output local copies of the filter, with values scaled by the intensities of these input pixels. Is that the actual result? Proceeding in the usual left-to-right and top-to-bottom fashion, it is the lower-right element of the filter, containing value 9, that hits the 1-pixel first, resulting in an output value of 9, followed by values...
Chapter 4: Image Processing

8 and 7. After the correlation is completed, we can confirm that, as expected, each of the nonzero pixels in the input image created a local copy of the filter in the output, and that its values are scaled by the intensity of these pixels (see Fig. 4.18c). However, these local copies are rotated by 180°!

In order to get the desired result, we just have to rotate the filter by 180° before performing the correlation. This type of operation is called correlation, and Fig. 4.18d shows that it produces the correct output. Besides having this nice spatial characteristic, it has other useful mathematical properties that we will explore as we apply convolution in different contexts. Particularly important is its commutative property: If we pad the input and the filter with infinitely many zeroes in all directions, then swapping input and filter, i.e., shifting the input across the filter, will lead to the same output as shifting the filter across the image as usual. This is not the case for correlation. For these reasons, we will always use convolution instead of correlation. Therefore, whenever applying a given filter, we always rotate it by 180° before shifting it across the input.

To describe convolution mathematically, let us assume that the filter is of size \((2m + 1) \times (2n + 1)\), where \(m\) and \(n\) are integers, so that there is a center pixel that serves as the anchor for storing outputs. Furthermore, let us use the “\(\star\)” symbol to indicate convolution. For an input image \(A\) and a filter \(F\), the value of output image \(A^*\) at position \([i, j]\) is computed by:

\[
A^*[i, j] = F[i, j] \star A[i, j] = \sum_{k=-m}^{m} \sum_{l=-n}^{n} F[k, l] A[i-k, j-l]
\]  

(4.9)

Here, \(F[i, j]\) and \(A[i, j]\) indicate that the filter is to be centered on position \([i, j]\) in the input. Eq. 4.9 has to be applied to every pixel in \(A\) to generate the output image \(A^*\). We can also use the more concise notation of \(A^* = F \star A\). Due to the commutative property of convolution, it is also true that \(A^* = A \star F\). However, note that due to our definition of spatial filtering, we could never apply a filter to an image that is smaller than itself; we would have to pad the image with zeroes first.

4.3.2 Gaussian convolution filters

What can these linear filters do for us? It turns out that they enable a variety of important operations that we will employ throughout the remainder of this book. Let us first consider the output image in Fig. 4.17. The output values are clearly larger than the input values and do not reveal large variation among them. It is
not surprising that the output values are large; after all, each of them is the sum of nine products of input and filter values. Since the sum of filter values is \(4 \cdot 1 + 4 \cdot 2 + 1 \cdot 4 = 16\), the output values are on average 16 times as large as the input values. This leads to an interesting idea: If we divided each output by 16, the resulting operation would compute a weighted mean of the intensity in the filter area. This can easily be achieved by dividing each entry in the filter by 16. It is most convenient to consider filters to be matrices and to use a scalar factor to normalize them. Then the resulting filter \(G_3\) is given by:

\[
G_3 = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}
\]

Applying this filter to the image in Fig. 4.17 yields the following output \(I^*\):

\[
I^* = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 5.6 & 5.8 & 5.8 & 0 \\ 0 & 5.5 & 5.6 & 6.1 & 0 \\ 0 & 5.1 & 4.8 & 5.9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}
\]

Clearly, the variance in the output image is lower than in the input data. Computing the weighted intensity mean within a local area decreases the differences in intensity, which leads to a smoothing of the image. The filter \(G_3\) in Eq. 4.10 is commonly used for smoothing. It gives greater weight to the more central pixels in order to conserve some of the original information while lowering the local contrast. The filter is derived from a two-dimensional Gaussian function that has its peak in the center and decreases with greater eccentricity. Therefore, it is known as a Gaussian filter. As shown in Fig. 4.19 (left and center panels), applying it to a natural image has the effect of blurring, similar to objects being out of focus. At the same time, it reduces local noise and makes surfaces appear more homogeneous, which facilitates subsequent processing by computer vision algorithms.
Fig. 4.19 Effects of Gaussian smoothing, demonstrated on a low-resolution, noisy image. Left: original image; Center: result of smoothing with a 3×3 Gaussian filter; Right: Result of smoothing with a 5×5 Gaussian filter.

For more pronounced blurring, a 5×5 Gaussian filter $G_5$ can be applied:

$$
G_5 = \frac{1}{273} \begin{bmatrix}
1 & 4 & 7 & 4 & 1 \\
4 & 16 & 26 & 16 & 4 \\
7 & 26 & 41 & 26 & 7 \\
4 & 16 & 26 & 16 & 4 \\
1 & 4 & 7 & 4 & 1
\end{bmatrix}
$$

The effect of $G_5$ is illustrated in the right panel of Fig. 4.19. Both its noise reduction and blurring are slightly stronger than for $G_3$. Which filter size is most appropriate depends on the noise level, the image size, and the intended application. Sometimes we may want to use Gaussian filters of much larger size, which leads to high computational complexity of the convolution algorithm, since the entire filter has to be applied to each individual pixel. Fortunately, the mathematical characteristics of the Gaussian function make it possible to speed up the computation. The idea can be demonstrated by rearranging the equation
for Gaussian filtering. A discrete, two-dimensional Gaussian function with center 
[0, 0] and standard deviation \( \sigma \) is given by:

\[
G[i, j] = e^{-\frac{j^2 + j^2}{2\sigma^2}}
\]  

(4.13)

To simplify the presentation, Eq. 4.13 does not include normalization of the 
Gaussian, which can be accomplished by multiplication with a constant. Let us 
assume that \( G \) is of size \((2m + 1) \times (2n + 1)\). According to Eq. 4.9, applying the 
Gaussian filter to an input image \( A \) leads to the following 
equation:

\[
G[i, j] \ast A[i, j] = \sum_{k=-m}^{m} \sum_{l=-n}^{n} e^{-\frac{k^2 + l^2}{2\sigma^2}} A[i-k, j-l]
\]  

(4.14)

\[
= \sum_{k=-m}^{m} \sum_{l=-n}^{n} e^{-\frac{k^2}{2\sigma^2}} e^{-\frac{l^2}{2\sigma^2}} A[i-k, j-l]
\]

\[
= \sum_{k=-m}^{m} e^{-\frac{k^2}{2\sigma^2}} \left( \sum_{l=-n}^{n} e^{-\frac{l^2}{2\sigma^2}} A[i-k, j-l] \right)
\]

The individual exponential terms in this equation describe the application of 
one-dimensional Gaussian filters. Therefore, we could first compute the term in 
parentheses, requiring us to convolve \( A \) with a horizontal \( 1 \times (2n + 1) \) Gaussian 
filter. The resulting image is then convolved with a vertical \((2m + 1) \times 1\) Gaussian 
filter, and according to Eq. 4.14, the final output is identical to the result of a 
single convolution of \( A \) with the two-dimensional \((2m + 1) \times (2n + 1)\) filter \( G \). For 
large Gaussian filters, the two consecutive convolutions will be substantially 
more efficient than the original single-filter convolution.

### 4.3.3 Median filters

While Gaussian filters are commonly used for smoothing operations, they are not 
an ideal choice for the removal of certain types of noise. For example, images 
might contain so-called salt-and-pepper noise, which is created by pixels of 
 extreme intensities, usually black and white (see Fig. 4.20, left panel). As shown 
in Fig. 4.20 (center panel), applying a \( 3 \times 3 \) Gaussian filter to images containing 
salt-and-pepper noise does not have the desired effect, because the extreme 
intensity values strongly affect the weighted mean computed within the small
filter area. In order to remove this noise, a filter that is less strongly affected by extreme values would be more suitable. Therefore, instead of computing the weighted mean, it is preferable to compute the median of the values within the filter area. For a 3×3 filter, the median is computed by sorting all nine intensity values in ascending order and picking the central element, i.e., the fifth one in the list, as the filter output. This operation is not a linear function, and therefore it cannot be accomplished by convolution. Nevertheless, we can use a very similar approach and shift a 3×3 window across the image, compute the median for each position, and place it in the output image. If we use the center of the window as the anchor for placing the output, a 3×3 median filter will also result in zeroes in the first and last rows and the first and last columns, just like a convolution filter.

Fig. 4.20  Removal of salt-and-pepper noise. Left: original image; Center: result of smoothing with a 3×3 Gaussian filter; Right: Result of applying a median filter.

The result of applying a 3×3 median filter to the example image with salt-and-pepper noise is shown in the right panel of Fig. 4.20. Obviously, the result is drastically better for the median filter than for the Gaussian filter. The noise has completely disappeared. On the negative side, the median operation tends to modify the contours of objects, which leads to undesirable local effects. With small median filters such as one demonstrated here, these effects are rather limited, and its use for removing noise with extreme outliers is recommended.
4.4 Spatial Frequency Filtering

In the previous section, filtering was performed in the spatial domain. A filter of a certain spatial extent was shifted across the input image, determining the influence of a pixel on a local output based on its relative spatial position. There is a different and somewhat complementary way of analyzing and filtering images, which occurs in the frequency domain. In this domain, signals are not described as functions of spatial location but as linear combinations of cosine and sine functions of various frequencies. The translation of signals between the spatial and frequency domains is accomplished by the Fourier transform, arguably one of the most important tools in modern signal processing and computer science in general. Since our goal is the processing of discrete, i.e., pixel-based images, we will focus on the discrete Fourier transform (DFT). In the following Section 4.4.1, the one-dimensional DFT will be introduced. This concept will be expanded to two dimensions in Section 4.4.2, followed by a description of the characteristics of the DFT and the most common filtering techniques in the frequency domain in Section 4.4.3.

4.4.1 The one-dimensional discrete Fourier transform (DFT)

The basic idea proposed by Joseph Fourier and refined by other researchers is that any periodic, i.e., repetitive, signal can be described, without loss of information, as a linear combination of cosine and sine functions of different frequencies. The coefficients in this linear combination, representing the contribution or weight of each frequency in the signal, form the Fourier transformed signal, also referred to as its frequency spectrum. The inverse Fourier transform converts the frequency spectrum back into the spatial representation of the signal.

For example, audio player software and stereo systems often have displays that show the volume of the currently playing sound in different frequency bands. Individual display bars indicate the strength of the audio signal from low frequencies on the left to high frequencies on the right. When music is played, these displays typically flicker because the contribution of each frequency band to the overall sound volume can change rapidly. At any given point in time, the
display shows a coarse version of the Fourier transformed most recent part of the audio signal.

The discrete variant of the Fourier transform has the nice property of guaranteeing perfect representation of any discrete function with a limited number of cosine and sine coefficients. Specifically, if the function is defined at \( n \) spatial positions, then \( n \) cosine and \( n \) sine coefficients suffice for its representation. In fact, as we will learn very soon, half of these coefficients are redundant due to the symmetry characteristics of the DFT. Let us first use real numbers to describe the DFT and its inverse before we introduce the standard notation that uses complex numbers. We will refer to the original signal, which we could think of as a discrete function describing the intensity values of a row of \( n \) pixels, as \( F[j] \), where \( j \) ranges from 0 to \((n - 1)\). The Fourier transformed signal consists of the functions \( \hat{A}[l] \) and \( \hat{S}[l] \) describing the contribution of cosine and sine functions, respectively, of different frequencies indicated by the value of \( l \) that ranges from 0 to \((n - 1)\). How are these frequencies determined? Basically, \( l \) tells us how many periods of a given cosine or sine function fit into one period of \( A \), i.e., our \( n \) pixels. The relationship between the spatial and frequency domains is given by the following equation:

\[
A[j] = \sum_{l=0}^{n-1} \hat{A}[l] \cdot \cos \left( 2\pi \frac{lj}{n} \right) - \hat{S}[l] \cdot \sin \left( 2\pi \frac{lj}{n} \right)
\]

(4.15)

Eq. 4.15 describes the inverse DFT. Given the frequency domain functions \( \hat{A}[l] \) and \( \hat{S}[l] \), this equation allows us to reconstruct the original spatial domain function \( A[j] \). Basically, we just need to sum up cosine and sine functions of different frequencies, weighted by the corresponding values of \( \hat{A}[l] \) and \( \hat{S}[l] \), and the result will be \( A[j] \). The minus sign in front of the sine term is due to the common notation of the DFT using complex numbers, which will be discussed below. For now, we only need to know that the signs of the coefficients \( \hat{A}[l] \) have to be switched when adding the cosine and sine terms.
Fig. 4.21  Fourier basis functions for function $A[j]$ in the spatial domain with a period $n = 6$. Each row shows one of the frequencies $l = 0..(n - 1)$, with cosine functions in the left column and sine functions in the right column.

What is the shape of these cosine and sine waves, also referred to as basis functions, whose linear combination can assume any discrete function with a
period of $n$ steps? As an example, let us consider the example of $n = 6$, which of course is extremely small but allows us to inspect every single basis function. Fig. 4.21 shows the 6 cosine functions (left column) and the 6 sine functions (right column) that are created by varying the frequency parameter $l$ from 0 to $(n - 1)$, i.e., in our case from 0 to 5. As noted above, $l$ indicates the number of periods that a cosine or sine function completes within $n$ steps. To show this role of frequency $l$, each of the 12 charts shows not only the values of $j$ between 0 and $(n - 1)$, which constitute one period of $A[j]$, but also $j = n$, which is the first value of $j$ in the following period of $A[j]$, and therefore $A[0] = A[n]$. This additional data point is shown in a different shade of gray.

Obviously, for $l = 0$ the cosine and sine functions turn into constants 1 and 0, respectively. While the constant cosine of 1 is useful for adding an identical “offset” to $A[j]$ for all values of $j$, the constant sine of 0 has no effect on $A$, and thus the value of its coefficient $Â_s[0]$ is meaningless. Looking at the cosine functions for increasing $l$, we see that higher frequencies lead to more sign reversals at the discrete points $j = 0..(n - 1)$, shown in Fig. 4.21 as black dots, that are relevant to the DFT. Interestingly, at $l = 4$, despite the higher frequency, there are fewer sign reversals at the discrete points than for $l = 3$. Loosely speaking, the frequency of the cosine is now too high for its sampling at the discrete points to “catch up” with it, and as a consequence, we start losing sign reversals in the discretized cosine function. When we take a close look at the cosine functions for $l = 2$ and $l = 4$, we notice that their discrete versions, indicated by the black dots, are perfectly identical. In other words, $Â_c[2]$ and $Â_c[4]$ have the same effect on $A$, and so do $Â_s[1]$ and $Â_s[5]$. It is generally true that, starting at $l = 0$, increasing $l$ by 1 increases the frequency of the discrete cosine function by 1 as well. However, once we have reached $l = n/2$, every further increase of $l$ by 1 decreases the frequency by 1. Therefore, the highest frequency in the Fourier spectrum is $n/2$, and $Â_c$ is mirror symmetric, which means that approximately half of it is redundant.

For the sine functions, the situation is similar. At first, increasing $l$ raises the frequency of the discrete sine function correspondingly. At $l = 3$ ($l = n/2$ in the general case), the discretization works in such a way that the function is 0 at all values of $j$, rendering the value of $Â_s[3]$ meaningless. Similar to the cosine, further increases in $l$ decrease the frequency of the discrete sine function. If we compare the discrete sine functions for $l = 2$ and $l = 4$, we find that, in contrast to the discrete cosine functions, they are not identical but are mirror images of each other along the horizontal axis. This means that they only differ by a factor of
(-1), and thus one of them is redundant in the linear combination of cosine and sine functions.

In total, out of the 6 cosine basis functions, there are only 4 linearly independent ones, as the ones for frequencies 4 and 5 are identical to those for frequencies 2 and 1, respectively. Moreover, there are only 2 independent sine basis functions, because those for frequencies 0 and 3 cannot contribute anything, and the ones at frequencies 4 and 5 differ from those at frequencies 2 and 1, respectively, only by a factor of (-1). The total number of independent values in the Fourier transformed function, consisting of $\hat{A}_c$ and $\hat{A}_s$, is thus 6. Since our spatial domain function $A$ also has 6 independent output values, it follows that there is a unique solution for both the Fourier transform and its inverse.

What is the best way to represent the Fourier transformed functions, given that $l$ does not represent the actual frequency of the discrete sinusoids and that there are redundant frequency pairs? A common solution is to treat the values of $l$ that exceed $n/2$ as negative frequencies. In our example with $n = 6$, the $l$-values 4 and 5 correspond to frequencies 2 and 1, respectively, and are thus assigned the negative frequencies (-2) and (-1). In the general case, this assignment can be accomplished by subtracting $n$ from all values of $l$ greater than $n/2$. Although the idea of negative frequencies is counterintuitive, it works well mathematically in the present context. In fact, it is consistent with Eq. 4.15 because of the even symmetry of the cosine function ($\cos (-x) = \cos x$) and the odd symmetry of the sine function ($\sin (-x) = -\sin x$). Therefore, using negative frequencies even accounts for the flipped sign in the discrete sine functions for $l > n/2$.

Let us take a look at several examples of functions in their frequency and spatial domain representations shown in Fig. 4.22. The first function has only one non-zero frequency component, which is a cosine function at frequency 1 with amplitude 2 (Fig. 4.22a). Therefore, in order to derive the spatial domain function, we simply take the cosine function for $l = 1$ from Fig. 4.21 and multiply all values by 2. Note that amplitudes can also be negative, which leads to cosine values with inverted signs, corresponding to a phase shift of the cosine function by 180°.
Fig. 4.22 (a) to (d): Four one-dimensional functions in their frequency domain representation by \( \hat{A}_c[l] \) (left column) and \( \hat{A}_s[l] \) (middle column) and their spatial domain representation by \( A[j] \). Note that both \( \hat{A}_c[l] \) and \( \hat{A}_s[l] \) can assume negative values.

The second function (Fig. 4.22b) is identical to the first one, except for an additional cosine component at frequency \( l = 0 \) of amplitude 1. As we noted before, \( l = 0 \) corresponds to a constant function of value 1. Therefore, this
component adds a constant value of 1 to each of the spatial domain points. This has the effect of the corresponding spatial function being shifted upwards by one unit compared to the previous one.

If two sinusoid functions, rather than one sinusoid and one constant function, are added, the result is more complex. Fig. 4.22c shows a function consisting of a cosine function of frequency $l = 3$ and a sine function of $l = -1$. As discussed above, the sine components are actually subtracted from the spatial domain function, and therefore, negative $l$-values correspond to the usual (positive) spatial sine functions. Due to its greater amplitude, the sine function dominates in the spatial domain over the cosine function, which only alters the shape of the sine function locally.

The final example, shown in Fig. 4.22d, is particularly important for the understanding of the Fourier transform. Here, the function consists of a cosine and a sine component, with both of them at the same frequency $l = 1$. As the amplitudes for the cosine and sine functions are 2 and 1, respectively, the spatial domain representation of this function is given by the following equation:

$$A(j) = 2 \cos\left(\frac{2\pi \cdot j}{6}\right) - \sin\left(\frac{2\pi \cdot j}{6}\right)$$

Based on Fig. 4.22d, it seems that the result is another sinusoid with a different phase than either the cosine or the sine function and an amplitude slightly above 2. In fact, the result of adding cosine and sine functions of the same frequency and phase is always another sinusoidal function. The general equation for subtracting a sine from a cosine as required for the inverse DFT is given by:

$$a \cdot \cos(x) - b \cdot \sin(x) = \sqrt{a^2 + b^2} \cdot \cos\left(x + \arctan\left(\frac{b}{a}\right)\right)$$

If $a = 0$, then the arctangent function returns a value of $90^\circ$, which turns the cosine into $-\sin x$. The significance of Eq. 4.17 is that in the spatial domain representation of a function, each pair of cosine and sine functions for the same frequency can alternatively be viewed as a single cosine function of magnitude $m$ and phase $\varphi$, where $m$ and $\varphi$ are computed as:

$$m = \sqrt{a^2 + b^2} \quad \text{and} \quad \varphi = \arctan\left(\frac{b}{a}\right)$$
While the magnitude \( m \) is simply the amplitude of the resulting sinusoid function, its phase \( \phi \) tells us how far it is shifted to the left in comparison to the “usual” cosine function with \( \phi = 0^\circ \) that reaches its maximum for input 0. Let us now consider the example given in Fig. 4.22d and Eq. 4.16 again. Here, \( a = 2 \) and \( b = 1 \), which by Eq. 4.18 gives us:

\[
m = \sqrt{a^2 + b^2} = \sqrt{5} \approx 2.24 \quad \text{and} \quad \phi = \arctan\left(\frac{b}{a}\right) = \arctan0.5 \approx 26.6^\circ \quad (4.19)
\]

In Fig. 4.22d (right panel), the period of a sinusoid with frequency 1 is 6 so that it completes a full 360° cycle between, for example, \( j = 0 \) and \( j = 6 \). A phase of \( 26.6^\circ \) then corresponds to a cosine function shifted to the left by \( 6 \times 26.6/360 \approx 0.44 \) spatial units. We therefore expect to find a minimum value of \((-2.24)\) near \( j = 2.56 \) and a maximum value of 2.24 near \( j = 5.56 \), which is confirmed by Fig. 4.22d.

For humans like us, representing the Fourier components of a function in terms of cosine functions of magnitude \( m \) and phase \( \phi \) is more intuitive than considering separate cosine and sine contributions at each frequency. Therefore, instead of using \( \hat{A}_c \) and \( \hat{A}_s \), the frequency domain representation of a function \( A \) is often given by its magnitude spectrum \( \hat{A}_m \) and its phase spectrum \( \hat{A}_\phi \). The conversion between these function pairs can be derived from Eq. 4.18. For a given frequency \( l \), we get:

\[
\hat{A}_m[l] = \sqrt{\hat{A}_c[l]^2 + \hat{A}_s[l]^2} \quad \text{and} \quad \hat{A}_\phi[l] = \arctan\left(\frac{\hat{A}_c[l]}{\hat{A}_s[l]}\right) \quad (4.20)
\]

\[
\hat{A}_c[l] = \hat{A}_m[l] \cdot \cos(\hat{A}_\phi[l]) \quad \text{and} \quad \hat{A}_s[l] = \hat{A}_m[l] \cdot \sin(\hat{A}_\phi[l]) \quad (4.21)
\]

Note that whenever the magnitude \( \hat{A}_m[l] = 0 \) for a given frequency \( l \), the corresponding phase \( \hat{A}_\phi[l] \) is meaningless and is usually set to zero. Fig. 4.23 shows the example function from Fig. 4.22d using the magnitude-phase representation in the frequency domain. Whereas the magnitude equals the amplitude of the spatial function, the phase only depends on the ratio between sine and cosine values. The magnitude-phase representation is not only easier to understand for us but is also more convenient for applying filter operations in the frequency domain. Therefore, we will use this representation when discussing the 2D DFT and its application to image processing and computer vision.
Before considering its two-dimensional variant, we need to complete our examination of the 1D DFT. It is clear from Eq. 4.15 and the above discussion how to transform the frequency domain representation of a function to the spatial domain, i.e., perform the inverse DFT. But how can we accomplish the actual DFT that translates functions from the spatial to the frequency domain? The DFT is based on the idea of correlation: The more the spatial domain function $A$ correlates with a basis function, i.e., a cosine or sine function of a given frequency $l$, the greater is the contribution of that cosine or sine function to $A$, which should be reflected in a greater value of $\hat{A}_m[l]$ or $\hat{A}_s[l]$, respectively, in the frequency domain representation of $A$. Here, correlation is computed in a way similar to the inner product of two vectors. At each spatial point $j = 0..(n - 1)$, the value of the basis function at that point is multiplied with $A[j]$, and the resulting $n$ products are summed.

Remember that the basis functions are sinusoids with a mean of zero, with the sole exception of the cosine at frequency zero that we will examine further below. Let us now assume that function $A$ is somewhat “in sync” with the basis function, i.e., it tends to be greater for those values of $j$ where the basis function is positive and tends to be smaller wherever the basis function is negative. Then their correlation – the sum of their products across all values of $j$ - will be positive, because the positive values of the basis functions are multiplied by greater values than the negative ones and thereby receive greater weight in the final sum. If $A$ tends to be smaller for greater values of the basis function and vice versa, the weight will shift so that the correlation becomes negative. Finally, if there is no systematic variation of $A$ at the frequency of the basis function, positive and negative summands will tend to cancel each other out, leading to a correlation value close to zero.
Table 4.1 Cosine and sine basis functions for the discrete Fourier transform for input size $n = 6$ (cf. Fig. 4.21). Rows indicate frequency $k$, and columns indicate spatial position $j$.

<table>
<thead>
<tr>
<th>$l$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>1</td>
<td>-0.5</td>
<td>-0.5</td>
<td>1</td>
<td>-0.5</td>
<td>-0.5</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>0.5</td>
<td>-0.5</td>
<td>-1</td>
<td>-0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>-0.5</td>
<td>-1</td>
<td>-0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-0.5</td>
<td>-0.5</td>
<td>1</td>
<td>-0.5</td>
<td>-0.5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Cosine basis functions | Sine basis functions
---|---|---|---|---|---|---|---|---|---|---|---|
0 | -0.87 | 0.87 | 0 | -0.87 | 0.87 |
1 | -0.87 | 0.87 | 0 | 0.87 | 0.87 |
2 | 0 | 0 | 0 | 0 | 0 |
3 | 0 | 0 | 0 | 0 | 0 |
4 | 0 | 0 | 0 | 0 | 0 |
5 | 0 | 0 | 0 | 0 | 0 |

Let us compute the correlations between the spatial domain representations of the four example functions in Fig. 4.22 and the 12 sinusoid basis functions as shown in Fig. 4.21. Table 4.1 lists the 6 values of each basis function at the discrete points $j = 0..5$, which are the only values relevant to the DFT. Fig. 4.24 shows the correlation results for each of the four spatial domain functions. For function (a), its correlation with all cosine and sine basis functions is zero, except for the cosines of frequencies $(-1)$ and 1, whose correlation value is 6. This is a promising outcome, because we know that function (a) is indeed a cosine function of frequency 1. However, there is a discrepancy: When we composed the function using the inverse DFT, we only included the cosine with positive frequency 1 (see Fig. 4.22a), but now we find contributions by two frequencies, $(-1)$ and 1.
Fig. 4.24 (a) to (d): The spatial representation $A[j]$ of the four example functions from Fig. 4.22 (left) and the computation of their correlation with the 6 cosine and 6 sine basis functions (cf. Fig. 4.21 and Table 4.1).
The reason for this result is the redundancy in the frequency domain data that we noticed before. The cosine basis functions for frequencies \((-1)\) and 1 are perfectly identical, and therefore each of them shows the same correlation with function (a). If we assign coefficients to both frequencies and perform an inverse DFT, then Eq. 4.15 tells us that only the sum of these coefficients will matter and determine the amplitude of the resulting spatial function; the share that each frequency receives is inconsequential. Therefore, we can add the two correlation values of 6 that we received for frequencies \((-1)\) and 1 in function (a) to obtain the total contribution of the absolute frequency 1, which is 12.

What does this value of 12 mean? One thing to notice is that these correlation values tend to increase with greater \(n\), because more products enter the sum. This does not seem fair – the correlation between two periodic functions should not depend on the number of places in which they are being compared. Therefore, correlations should be normalized by dividing them by \(n\). In our example, this yields a result of \(12/6 = 2\). It matches the value of \(\hat{A}_c[1]\) that we used to generate the function in the first place. Maybe the normalized correlation that we performed is in fact the DFT? Let us verify this with the remaining three example functions.

Function (b) in Fig. 4.24 leads to the same result as for function (a), except that we find an additional correlation of strength 6 for the cosine of frequency 0. Remember that this basis function equals 1 for all inputs. Therefore, correlating a function with it and dividing the result by \(n\) simply yields the average of all of the function’s outputs within one period. This value is needed in the DFT as an offset for the spatial domain function. Since all sinusoids have a mean value of zero, summing them must always generate a function whose mean value is zero as well. Consequently, without the offset enabled by the cosine of frequency 0 – which is not a sinusoid but a constant – the DFT could only represent functions whose mean is zero.

For function (c), we find non-zero correlations for both cosine and sine functions. The cosine at frequency 3 reveals a correlation value of 4, and its normalization 4/6 or 2/3 is exactly the value that was used in the creation of function (c). The reason for this match is that if \(n\) is even, the highest frequency \((n/2)\) has no redundant counterpart. Therefore, in the case of \(n = 6\), \(\hat{A}_c[3]\) must represent the entire contribution of cosines of frequency 3 to function (c). Furthermore, there are correlations of strengths \((-6)\) and 6, normalized to \((-1)\) and 1, with the sine functions at frequencies \((-1)\) and 1, respectively. The mismatched signs are due to the redundancy among the sine basis functions, which differs
from that of the cosines because of the odd symmetry of the sine function, leading to flipped signs in each redundant pair. Thus, applying the inverse DFT to $\hat{A}_s[-1] = -1$ and $\hat{A}_s[1] = 1$ will yield the same result as for $\hat{A}_s[-1] = -2$ and $\hat{A}_s[1] = 0$ or for $\hat{A}_s[-1] = 0$ and $\hat{A}_s[1] = 2$. We have to remind ourselves, however, that we subtracted the sine components instead of adding them, which means that we have to flip the signs of all coefficients, making them compatible with $\hat{A}_s[-1] = 2$, which we used to generate function (c).

Finally, after removing the redundancies of the cosine and sine functions, we can see that the result for function (d) also matches the data in Figure 4.22d. The cosines at frequency 1 have a total contribution of 12, normalized to 2, and the sines at the same frequency add up to 6, which is normalized to 1. We will accept these examples as evidence for the correlation method to perform the DFT. For a more thorough mathematical treatment of the DFT including various theorems and their proofs, suitable references are provided at the end of this chapter. For our current purpose, we only need to write down the equations of the DFT:

$$\hat{A}_s[l] = \frac{1}{n} \sum_{j=0}^{n-1} A[j] \cdot \cos\left(2\pi \cdot \frac{lj}{n}\right)$$ and $$\hat{A}_c[l] = -\sum_{j=0}^{n-1} A[j] \cdot \sin\left(2\pi \cdot \frac{lj}{n}\right)$$ (4.22)

In Eq. 4.22, the range of $l$ can be chosen as $0..(n - 1)$ to be the perfect counterpart to the inverse DFT as defined in Eq. 4.15. Alternatively, we can use the range $(-n/2 + 1)..(n/2)$ for even $n$ and $(-n + 1/2)..<(n + 1)/2$ for odd $n$ to introduce negative frequencies and treat the redundancies and symmetries of the DFT more adequately. While the two approaches are mathematically equivalent, the former is more convenient from a computational perspective, and the latter is better suited for visualizing and manipulating data in the frequency domain.

When considering the computational aspects of the DFT (Eq. 4.22) and its inverse (Eq. 4.15), it is striking how similar their equations are. Furthermore, it is slightly inelegant to have two separate functions to represent the frequency domain and, for that reason, two equations describing the DFT. In order to exploit the similarity of the equations and use a more concise and elegant notation, the DFT is typically described with the help of complex numbers. As explained in Chapter 3, complex numbers are of the form $a + bi$, where $a$ and $b$ are real numbers and referred to as the real and imaginary parts, respectively, of the complex number. The constant $i$ is the (imaginary) square root of $-1$. We use the Greek letter iota rather than the letter $i$ in order to avoid confusion with the index $i$ that we commonly use to indicate the row of a matrix. The advantage of complex numbers in the present context is that each of them can hold two
scalars, and thus we can define a single complex-valued function $\hat{A}[l]$ such that $\hat{A}[l] = \hat{A}_c[l] + \hat{A}_s[l] \cdot i$. In other words, the real and imaginary parts of $\hat{A}[l]$ represent the contribution of the cosine and sine, respectively, of frequency $l$ to the spatial domain function $A[j]$. $A[j]$ itself is also considered to be complex, with its real parts containing the spatial domain data and its imaginary parts being zero. Moreover, the exponential function in the complex domain has the following interesting characteristic:

$$ e^{\alpha} = \cos \alpha + i \cdot \sin \alpha \quad (4.23) $$

It is also important to remember the rule for multiplying two complex numbers:

$$(a + bi) \cdot (c + dt) = (ac - bd) + (bc + ad)i \quad (4.24)$$

Then it follows from Eqs. 4.22, 4.23, and 4.24 that the DFT can be written in the following form:

$$ \hat{A}[l] = \frac{1}{n} \sum_{j=0}^{n-1} A[j] \cdot e^{-2\pi ij/n} \text{ for } l = 0..(n-1) \quad (4.25) $$

Given Eqs. 4.14, 4.23, and 4.24, the inverse DFT can be stated as:

$$ A[j] = \sum_{l=0}^{n-1} \hat{A}[l] \cdot e^{2\pi jl/n} \text{ for } j = 0..(n-1) \quad (4.26) $$

Eqs. 4.25 and 4.26 are clearly more concise and elegant than our original Eqs. 4.22 and 4.15, respectively. Interestingly, if we implement both $A$ and $\hat{A}$ as complex arrays of size $n$, the DTF and its inverse differ only by the sign of the exponent and the constant $1/n$. Alternatively, we could use the factor $1/n$ in the inverse DFT instead of the forward DFT, or the factor $n^{1/2}$ in both the DFT and its inverse to further emphasize their symmetry. We can now also understand the reason why the sine functions are subtracted from rather than added to the spatial domain function – it is advantageous in terms of the complex notation of the DFT and its inverse. Furthermore, the complex notation facilitates the introduction and computation of the two-dimensional DFT, which will be described in the next section.
4.4.2 The two-dimensional DFT

For our purposes of image processing, it is obvious that a 2D version of the DFT is required. Ideally, we would like to be able to apply a DFT to an $m \times n$ matrix of pixel intensities to obtain two matrices of the same size, holding the cosine and sine components of the Fourier transformed image. These matrices could subsequently be converted into 2D magnitude and phase spectra, which are more convenient for image processing applications. It goes without saying that we would also like to perform all of these operations in the inverse direction.

Let us stick with the complex notation to keep things simple. Then the spatial domain representation of our function is given by a complex $m \times n$ matrix $A[i, j]$ whose real parts contain pixel intensities at any scale and whose imaginary parts are zero. The frequency domain is represented by another complex $m \times n$ matrix $Â[k, l]$ whose real and imaginary parts contain the amplitudes of cosine and sine functions, respectively. Eqs. 4.25 and 4.26 can be adapted toward these two-dimensional inputs and outputs in a straightforward manner:

$$
\hat{A}[k, l] = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A[i, j] \cdot e^{-2\pi j \left(\frac{ki}{m} + \frac{lj}{n}\right)} \quad \text{for } k = 0..(m-1), \ l = 0..(n-1) \quad (4.27)
$$

$$
A[i, j] = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} \hat{A}[k, l] \cdot e^{2\pi j \left(\frac{ki}{m} + \frac{lj}{n}\right)} \quad \text{for } i = 0..(m-1), \ j = 0..(n-1) \quad (4.28)
$$

Eqs. 4.27 and 4.28 are identical to their one-dimensional counterparts except for a change in the scaling factor that accounts for the nested sums, and the addition of the term $\frac{ki}{m}$ in the exponent. This term allows the basis functions to not only form horizontal waves at frequency $l$ – as in the 1D case – but also vertical ones at frequency $k$. The result is that each element in the frequency domain matrix $Â$ represents a 2D wave of particular vertical and horizontal frequencies. More intuitively, we can consider these waves to have a specific frequency and 2D orientation. To illustrate this concept, let us once again consider the notation of positive and negative frequencies. If $m$ and $n$ are even, the vertical and horizontal frequencies are represented in $Â$ as shown in Fig. 4.25. Frequency $[0, 0]$ is located in the center of the matrix, and its associated cosine is once again the only non-zero constant among all basis functions and thus serves as an overall offset. Greater distance from the center vertically and horizontally corresponds to higher frequency in the same direction.
Chapter 4: Image Processing

Fig. 4.25 Representation of vertical frequency $k$ and horizontal frequency $l$ in a Fourier transformed 2D matrix $\hat{A}[k, l]$.

What is the shape of the 2D sinusoid basis functions? Let us take a look at the sine functions at the very lowest frequencies, i.e., those associated with elements near the center of the matrix. To simplify things, let us assume that the underlying matrix is square, i.e., $m = n$. Fig. 4.26 illustrates the sine basis functions for all combinations of vertical frequency $k$ and horizontal frequency $l$ in the interval from (-3) to 3. Each of the 49 panels shows how the entire spatial domain image $A$ is covered by the sine basis function associated with one element $\hat{A}[k, l]$. In other words, if we set the sine component of $\hat{A}[k, l]$ to 1, set its cosine component to 0, and set the components of all other matrix elements to 0 as well, then the panel shows the result of the inverse DFT. Greater brightness indicates higher values, ranging from black (-1) to white (1), with medium gray representing zero, for example, at frequency [0, 0] in the center.
How do these wave patterns arise? Let us examine, as an example, the basis function for frequency $[2, 3]$ near the lower-right corner of Fig. 4.26. Vertically, this function completes two cycles. We can check this by following the function along the right side of its panel, finding that it has two brightness peaks, each of which indicates a separate cycle or period. Doing the same on the panel’s left border or along any vertical line cutting through the panel yields the same result.
Horizontally, we always count three cycles. Thus, the two-dimensional frequency 
[2, 3] still consists of independent vertical and horizontal frequencies 2 and 3. 
The resulting 2D basis function looks like a single 2D sinusoid with a single 
frequency and orientation. As suggested by Fig. 4.26, the orientation of the bright 
and dark “bars” formed by the basis functions for \( \hat{A}[k, l] \) is perpendicular to the 
vector from element \([0, 0]\) to element \([k, l]\). The frequency of the 2D sinusoid is 
given by the length of that vector. Consequently, Fig. 4.26 resembles concentric 
circles that become thinner and more densely packed as their radii increase. For a 
given input image, the 2D DFT determines the coefficients \( \hat{A}[k, l] \) for all of these 
sine and cosine basis functions so that their sum for each pixel matches the 
intensity of the input image.

Before we explore the characteristics and applications of the two-dimensional 
DFT, let us consider some important issues regarding its implementation on a 
computer. If we programmed the DFT exactly as defined in Eqs. 4.27 and 4.28, 
we would need to use four nested loops for the variables \( i, j, k, \) and \( l \). For a square 
input image with \( n \times n \) pixels, the complexity of this computation would be \( O(n^4) \), 
even if we exploit the symmetries of the transform. For images in the megapixel 
range, we could easily have lunch while the result is computing. Fortunately, we 
can use a trick similar to the one for the Gaussian convolution filter in Eq. 4.14 in 
Section 4.3.2. Even for complex numbers, it is still true that \( e^{a+b} = e^a e^b \). 
Therefore, Eq. 4.27 can be reformulated as follows:

\[
\hat{A}[k, l] = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A[i, j] \cdot e^{-2\pi \frac{ki}{m} \frac{lj}{n}}
\]

\[
\hat{A}[k, l] = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A[i, j] \cdot e^{-2\pi \frac{lj}{n}} \cdot e^{-2\pi \frac{ki}{m}}
\]

\[
\hat{A}[k, l] = \frac{1}{m} \sum_{i=0}^{m-1} \left( \frac{1}{n} \sum_{j=0}^{n-1} A[i, j] \cdot e^{-2\pi \frac{lj}{n}} \right) \cdot e^{-2\pi \frac{ki}{m}}
\]

Eq. 4.29 shows that we can first carry out separate one-dimensional DFTs for 
each row of \( A \) and realign the transformed rows to form a new matrix \( A' \).
Subsequently, we apply separate one-dimensional DFTs to each column of \( A' \), 
realign the results into a new matrix, and find that this is the 2D Fourier 
transformed matrix \( \hat{A} \). Alternatively, we could transform the columns first and 
the rows afterwards. Note that the inverse DFT can be performed analogously.
For an \( n \times n \) matrix, this procedure requires a total of \( 2n \) DFTs of \( n \)-element inputs, which means that its complexity is \( O(n^3) \). While this procedure is still considerably complex, the improvement over the literal implementation of the DFT is significant. A 1-megapixel image can be transformed within a few seconds, which is sufficient for experimenting with the DFT. The DFT code in Appendix A is based on this procedure, because it can easily be understood based on the material presented here. If faster performance is required, the Fast Fourier Transform (FFT) is a more appropriate choice and is used in almost all technical applications of the DFT. It is based on a complex divide-and-conquer algorithm that would require much explanation without adding to our understanding of computer vision. Relevant literature on this topic is referenced at the end of this chapter.

### 4.4.3 Characteristics and applications of the two-dimensional DFT

To start our exploration of the DFT for image processing, let us transform a rectangle. Fig. 4.27 shows the input image (left panel) and the results of the DFT in terms of the magnitude spectrum (center panel) and phase spectrum (right panel). Magnitude and phase for each pixel are computed in exactly the same way as for the one-dimensional DFT (see Eqs. 4.20 and 4.21). Magnitude spectra are shown after a gamma transformation with \( \gamma = 0.2 \) in order to increase the visibility of data points near the borders of the image, which can be several orders of magnitude smaller than those near the center. Phase spectra are linearly scaled from \(-180^\circ\) (black) to \(180^\circ\) (white).

---

**Fig. 4.27** Left: Input image for a 2D DFT showing a white rectangle on black background; Center: Magnitude spectrum; Right: Phase spectrum.
In order to create the sharp edges of the rectangle in the spatial domain, many sinusoids of different frequencies have to be summed, creating the repetitive vertical and horizontal patterns seen in the magnitude spectrum. Since lower frequencies are represented closer to the center of the transformed image, the greater horizontal extent of the rectangle leads to a narrower central pattern and repetitions in that direction as compared to the vertical one. Phase spectra are difficult to interpret by visual inspection; patterns are visible only in cases of very simple input images.

If we shift the rectangle to a different position in the input image, the contributions of different frequencies to the image should not change. Remember that the Fourier transform expects its input signal to be periodic, i.e., repeat infinitely many times to the left, to the right, upward, and downward. It is then clear that shifting the relevant signal essential remains unchanged except for its phase. As shown in Fig. 4.28, the magnitude spectrum of the input image is indeed perfectly identical to the one in Fig. 4.27. In contrast, the phase spectrum has changed quite dramatically. Notice that a shift in the spatial domain corresponds to phase shifts in the frequency domain that differ for each contributing function. The reason is that sinusoids of different frequencies, due to their specific periods, require distinct phase shifts to accomplish the same spatial displacement.

Fig. 4.28 Effect of shifting the rectangle in Fig. 4.27 to a different position. Left: Input image; Center: Magnitude spectrum; Right: Phase spectrum.

What happens if we rotate the rectangle? Fig. 4.29 shows the effect of rotating it clockwise by 45° on the magnitude and phase spectra. Clearly, the magnitude spectrum is also rotated. Generally, magnitude spectra conserve rotations of the input image, i.e., perform an identical rotation themselves. The phase spectrum,
on the other hand, once again undergoes a seemingly unpredictable transformation.

Fig. 4.29 Effect of rotating the rectangle in Fig. 4.27 clockwise by 45°. Left: Input image; Center: Magnitude spectrum; Right: Phase spectrum.

Since the magnitude spectrum does not depend on the absolute position of the spatial pattern in the input, what happens if the same object, such as our rectangle, appears in it multiple times in various positions? Fig. 4.30 shows the result of this experiment. The magnitude spectrum is still very similar to Figs. 4.27 and 4.28 but with more fine-grained variation in its values. Some particularly elevated pixels lead to the remainder of the image looking darker than the magnitude spectra in the previous figures. At any rate, it seems that magnitude spectra can be indicative of particular spatial patterns, regardless of their position or multitude. The frequency spectrum now looks almost completely like random noise. This is the case for most complex input patterns such as natural images. For that reason, from this point on we will only consider the magnitude spectra in our discussion of the DFT.
Fig. 4.30 Fourier transform of multiple, identical rectangles in the same image. Left: Input image; Center: Magnitude spectrum; Right: Phase spectrum.

Moving on to real-world images, Fig. 4.31 shows the Fourier transform of an indoor scene. Despite the gamma transformation, it is still visible that the values are greatest in the center and diminish with increasing eccentricity. It is important to notice that this pattern is not a characteristic of the Fourier transform but a characteristic of our world. If we Fourier transform a random-dot pattern, the resulting magnitude spectrum will not show this center bias. However, the structure of our visual environment is not entirely random. Let us compare the light intensities at any two positions in our visual field that are separated by only a small visual angle, say 0.1°. These two points are very likely to lie on the surface of the same object or texture and thus have similar intensity. In contrast, if we choose two points that are far away from each other, maybe 30°, they are likely to be located on different objects or surfaces, or at least the lighting or shading may differ between the two positions. These factors tend to induce differences in intensity. Consequently, significant and systematic changes in intensity occur more often at larger spatial scales, which correspond to lower spatial frequencies.
Moreover, the magnitude spectrum shows several straight lines that pass through its center. To interpret these lines, we should remind ourselves of Fig. 4.26, where we examined the orientation of the “bars,” i.e., the pattern of light-dark oscillations associated with the sinusoid basis functions. We found that this orientation for frequency \([k, l]\) is perpendicular to the vector from the center \([0, 0]\) to position \([k, l]\) in the spectrum. This means that any basis functions that can create lines or edges of orientation \(\alpha\) are represented in the Fourier spectrum on a straight line perpendicular to \(\alpha\) that passes through the center of the spectrum. Therefore, if we see elevated magnitudes in the Fourier spectrum along a straight line through its center, we can conclude that the input image must contain a large number or strength of edges that are perpendicular to the line.

In the example of Fig. 4.31, the input image contains many linear elements in various directions, such as the flower stems or the edge of the ceiling. And in fact, the magnitude spectrum reveals numerous straight lines in orientations that could be perpendicular to those in the spatial domain. However, there is one odd line, which is perfectly vertical and also crosses the center. According to our discussion, it should indicate that there are one or more strong horizontal lines in the input image, but they do not seem to exist. In order to explain this observation, we must remember that the Fourier transform expects its input to be periodic, i.e., repetitive, in all dimensions. Consequently, the transform “sees” an infinite repetition of the input image. Fig. 4.32 shows the repeated input for the image from Fig. 4.31 for three cycles in each dimension. We can now see the “phantom edge” that caused the vertical line in the magnitude spectrum: Because
the bottom of the image is much darker than its top, the vertical repetition of the image creates perfectly straight horizontal edges! The intensity difference between the image’s left and right borders is less pronounced and thus has a smaller effect on the magnitude spectrum.

![Image](image_url)

Fig. 4.32 The input image from Fig. 4.31 as a repetitive function in vertical and horizontal directions. Only three repetitions are shown, whereas the Fourier transform assumes an infinite repetition.

If we want to use the DFT to analyze the orientation of edges in images – and we will do this in later chapters – then we need to find a way to minimize such artifacts in the Fourier spectrum. One way to achieve this goal is to gradually lower the contrast in the input image with greater distance from its center. This process is also referred to as *windowing*. The subsequent DFT will then no longer “see” any sharp vertical or horizontal edges along the borders of the image. An appropriate tool for such image manipulation is a two-dimensional Gaussian function. Many other functions have been tested for this application as well as for the operations described further below, and they have their advantages in specific
cases. Nevertheless, in virtually all studies, the Gaussian has been found among the best approaches for any of these applications, and therefore it will be our function of choice.

For our purposes, it is most convenient to define the 2D Gaussian function in such a way that its maximum is 1. Let us further assume that the input image is square-shaped and contains $n \times n$ pixels. It is not a necessary condition for the DTF, but it greatly simplifies the interpretation and processing of Fourier spectra. Then the Gaussian function centered at $[n/2, n/2]$ with standard deviation $\sigma$ is given by:

$$G[i, j] = e^{\frac{-(i-n/2)^2 + (j-n/2)^2}{2\sigma^2}} \quad (4.30)$$

In order to let the function drop near zero at the image borders but conserve normal contrast levels in a large central area, a choice of $\sigma = n/4$ is appropriate.

If the maximum intensity in the original image $A$ is given by $I_{\text{max}}$, then the windowed image $W$ can be computed as follows:

$$W[i, j] = \frac{I_{\text{max}}}{2} + G[i, j] \cdot \left( A[i, j] - \frac{I_{\text{max}}}{2} \right) \quad (4.31)$$

The resulting windowed version of the example image and the output of the subsequent DFT are shown in Fig. 4.33. Obviously, the windowing had the desired effect: While the vertical line has disappeared, the other straight lines that are indicative of edge orientations in the input image are still visible.

![Fig. 4.33](image)

**Fig. 4.33** Left: Example image from Fig. 4.31 after windowing with a Gaussian function; Right: Magnitude spectrum of the windowed image.
The most common application of the DFT in image processing is frequency filtering. It involves a three-step procedure: First, a DFT is applied to the input image. Second, the magnitude spectrum is manipulated, for example, by removing the highest frequencies. Third, the inverse DFT is computed on the modified magnitude spectrum and the unaltered phase spectrum. This procedure allows us to perform operations in the frequency domain such as filtering out high frequencies across the image.

Let us start with exactly this type of manipulation, also known as low-pass filtering. Once again, a Gaussian function can be used to accomplish the task. Once we have Fourier transformed the image, we can multiply each value in the magnitude spectrum with that of Gaussian function $G$ in Eq. 4.30 centered on the spectrum. The effect of this operation is that the magnitude of low frequencies will be conserved whereas the magnitude of greater frequencies will be progressively diminished. The extent of this manipulation can be adjusted through the standard deviation $\sigma$. A small value will let only the lowest frequencies pass into the filtered image, whereas a large value will only affect the highest frequencies. Figure 4.34 shows the effects of the low-pass filtering procedure on the example image for $\sigma = 0.1n$ (left panel) and $\sigma = 0.05n$ (right panel). While the left image shows only very slight blurring, the effect is much stronger in the right image. The results look very similar to those of spatial filtering with Gaussian convolution filters of different sizes. In fact, they are identical.

Fig. 4.34 Results of low-pass frequency filtering using a Gaussian filter. Left: $\sigma = 0.1n$; Right: $\sigma = 0.05n$. 
What happens if we filter out the low frequencies instead of the high ones? To find out, we can simply turn our Gaussian filter $G$ upside-down, i.e., multiply all values in the magnitude spectrum with $(1 - G[k, l])$. Note that the value in the center of the spectrum should not be manipulated, because it encodes the overall offset, i.e., the average intensity of the image. If we set it to zero, the average intensity will be black, and only the brightest pixels will remain visible.

With this filter, only high frequencies will pass into the output image without significant reduction, whereas the contribution of low frequencies is diminished. The effect of this high-pass filter will increase, i.e., extend towards higher frequencies, for greater values of $\sigma$. Fig. 4.35 shows the results of high-pass filtering for $\sigma = 0.02n$ (left panel) and $\sigma = 0.1n$ (right panel). The image in the left panel still shows all local details as the original image in Fig. 4.31 such as the leaves and blossoms of the flowers. However, the overall tone of the image is medium gray and does not vary across different regions of the image. We find this outcome because the high frequencies, encoding local details, are still intact, whereas the low frequencies, encoding intensity changes across greater distances, have been filtered out. This effect is even stronger in the right panel, where only the finest structures remain in the image. Such an extreme high-pass filter could also be used as an edge detector.

![Fig. 4.35 Results of high-pass frequency filtering using a Gaussian filter. Left: $\sigma = 0.02n$; Right: $\sigma = 0.1n$.](image-url)
Custom-designed frequency filters can be used to remove particular types of noise that would be difficult to eliminate using spatial filtering. In some cases it is even possible to remove noise from an image by visually inspecting its magnitude spectrum, modifying it appropriately, and transforming it back into the spatial domain. For example, the upper-left panel of Fig. 4.36 shows an image with noise in the form of horizontal stripes, which may have been caused by interfering signals during analog transmission. The upper-right panel shows the magnified central area of this image’s magnitude spectrum. As noted above, the coefficients of the basis functions that are responsible for creating horizontal lines must be located on a vertical line that passes through the center of the spectrum. Indeed, there is a column of pixels above the center – plus its mirror image below the center - that seems to show systematically increased values.
Fig. 4.36 Illustration of noise removal in the frequency domain. Upper-left: Noisy input image; Upper-right: Central part of the magnitude spectrum; Lower-left: Same magnitude spectrum with the suspected noise contributions removed; Lower-right: Restored image after inverse DFT.