HALCON
a product of MVTec

Gray Value Interpolation



HALCON 21.05 *Progress*

This technical note describes the methods for gray value interpolation that are used in HALCON operators, Version 21.05.0.0.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Copyright © 2013-2021 by MVTec Software GmbH, München, Germany



Protected by the following patents: US 7,062,093, US 7,239,929, US 7,751,625, US 7,953,290, US 7,953,291, US 8,260,059, US 8,379,014, US 8,830,229. Further patents pending.

Microsoft, Windows, Windows Server 2008/2012/2012 R2/2016, Windows 7/8/8.1/10, Microsoft .NET, Visual C++, and Visual Basic are either trademarks or registered trademarks of Microsoft Corporation.

All other nationally and internationally recognized trademarks and tradenames are hereby recognized.

More information about HALCON can be found at: http://www.halcon.com/

About This Technical Note

This technical note describes the methods for gray value interpolation that are used in HALCON operators for image transformations, measure objects, or the determination of gray values at specific positions.

This technical note is divided into the following parts:

- **Introduction**
  A short introduction to gray value interpolation and the interpolation methods.
- **Interpolation Methods**
  A detailed description of each interpolation method provided by HALCON operators. Furthermore, an explanation of the influence of the system parameter *'int_zooming'*.
- **Examples**
  Some examples, which illustrate the effects of the different interpolation methods.

Chapter 1

Introduction

Gray value interpolation is a method that is necessary for image transformations because images are no continuous functions but digitized, i.e., consist of pixels associated with (discrete) gray values. If an image is, e.g., rotated and downscaled as in figure 1.1, a resulting pixel of the output image does not correspond to exactly one pixel of the input image anymore. Instead, it may contain parts of more than one pixel of the original image. Therefore, a method to calculate the gray value of the resulting pixel is required. One possible method is to use the gray value of exactly one pixel of the original image, e.g., the one whose center lies closest to the center of the resulting pixel. This means, however, that information contained in the original image, and with it image quality, might be lost in the transformation process. The alternative is to interpolate between the gray values of more than one pixel of the original image.

There are several ways to interpolate between more than one pixel, depending on the number of pixels that are considered and their influence on the resulting gray value. A detailed description of the different interpolation methods that are used in HALCON operators is given in the following chapter.

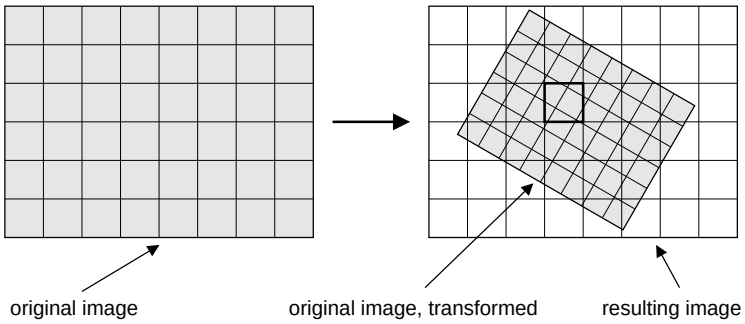

Figure 1.1: Image transformation consisting of rotation and downscaling.

Chapter 2

Interpolation Methods

2.1 Overview

The method *'nearest_neighbor'* is the simplest and fastest method for interpolation. It computes the resulting gray value based on only one pixel in the original image as described in section 2.2. It should only be used for time-critical applications.

The method *'bilinear'* considers four neighboring pixels for the interpolation (see section 2.3 on page 8). It produces better results with less undesired jagged edges for image transformations that include upscaling and/or rotation but no downscaling.

The methods *'constant'* and *'weighted'* perform a multi-step interpolation with an integrated smoothing as described in section 2.4.1 on page 9 and section 2.4.2 on page 9, respectively. They produce good results for both upscaling and downscaling. Therefore, they are best used for image transformations with an application-dependent scaling factor, which may lead to upscaling as well as to downscaling.

The methods *'bicubic'* and *'bicubic_clipped'* consider 16 neighboring pixels of the original image for determining the gray values (see section 2.5 on page 10). They provide the highest quality for upscaling at the cost of higher run times.

2.1.1 Interpolation Methods for Image Transformations

Table 2.1 lists the interpolation methods that are provided by HALCON operators that perform image transformations. Additionally, the characteristics of each interpolation method with respect to quality and speed as well as its recommended usage are stated.

Interpolation method	**Quality**	**Runtime** (relative to *nearest_neighbor*)	**Recommended usage**
nearest_neighbor	low	1	Only for time-critical applications
bilinear	high (for upscaling)	3	If downscaling must not be slower than upscaling
constant	enhanced for downscaling	3 (upscaling) 10 (downscaling)	If the image quality should be consistent for downscaled and upscaled images
weighted	best for downscaling	3 (upscaling) 20 (downscaling)	If the image quality of downscaled images is critical for the application
bicubic	very high (for upscaling)	6	If the image quality of upscaled images is critical for the application

Table 2.1: Interpolation methods for image transformations and their characteristics with respect to quality, runtime, and recommended usage. Note that the runtime factors may vary depending on the downscaling factor and on your hardware.

2.1.2 Interpolation Methods for Measure Objects and the Determination of Gray Values at Specific Positions

Table 2.2 list the interpolation methods that are provided for measure objects (see Solution Guide III-A) and the determination of gray values at specific positions (e.g., `get_grayval_interpolated`, `get_grayval_contour_xld`).

Interpolation method	**Quality**	**Runtime** (relative to *nearest_neighbor*)		**Recommended usage**
		Measure	Det. gray values	
nearest_neighbor	low	1	1	For time-critical applications
bilinear	high	1.3	3	Standard case
bicubic / *bicubic_clipped*	very high	1.8	6	For highest accuracy

Table 2.2: Interpolation methods for measure objects and the determination of gray values at specific positions and their characteristics with respect to quality, runtime, and recommended usage. Note that the runtime factors may vary depending on your hardware.

2.2 Nearest Neighbor Interpolation (*'nearest_neighbor'*)

If *'nearest_neighbor'* is selected, no interpolation over multiple pixels is performed. Instead, the resulting pixel is assigned the gray value of the pixel in the original image whose center lies closest to the center of the resulting pixel. The process is illustrated in figure 2.1:

In this example, the resulting pixel receives the gray value of pixel p10. Nearest neighbor interpolation is the simplest and fastest approach for determining the gray value of the resulting pixel. But typically, it leads to undesired effects like aliasing and jagged edges if the image is rotated or scaled.

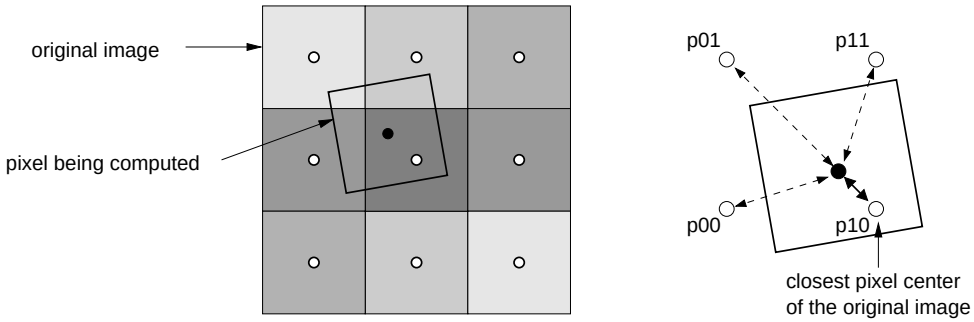

Figure 2.1: Determining gray values from the nearest neighbor.

2.3 Bilinear Interpolation (*'bilinear'*)

If *'bilinear'* is selected, the gray values are determined using bilinear interpolation. In contrast to nearest neighbor interpolation, in bilinear interpolation the gray value is determined from the four surrounding neighboring pixels, as depicted in figure 2.2.

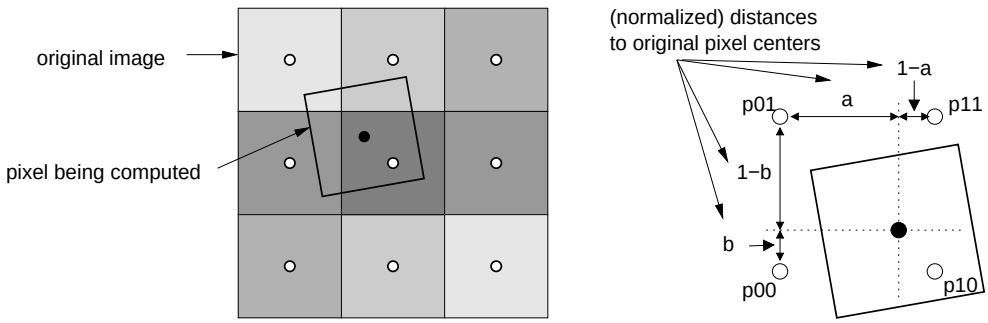

Figure 2.2: Determining a gray value using bilinear interpolation.

The influence of a neighboring pixel depends on the distance between its center and the center of the resulting pixel. Therefore, first the horizontal and vertical distances of the transformed coordinates to the adjacent pixel centers are determined. Then, the resulting gray value is computed as follows:

$$h_{res} = b \cdot (a \cdot h_{11} + (1-a) \cdot h_{01}) + (1-b) \cdot (a \cdot h_{10} + (1-a) \cdot h_{00})$$

with h_{res} denoting the gray value of the resulting pixel and h_{ij} denoting the gray values of the neighboring pixels. Because in this case, the resulting gray value is influenced by the gray values of all four neighboring pixels, bilinear interpolation produces smoother images than nearest neighbor interpolation. However, it also requires longer computation times.

Note that bilinear interpolation may lead to unexpected results due to aliasing if the image is downscaled. To reduce these undesired effects, an interpolation method with integrated smoothing (see section 2.4) should be used if a high quality of downscaled images is required.

2.4 Bilinear Interpolation with Integrated Smoothing

If the image transformation includes a downscaling of the image, so-called *aliasing* effects can appear. These effects result from a loss of information and may produce undesired artifacts in the downscaled image.

To reduce the effects of aliasing, the image must be smoothed before it is scaled down, e.g., using a mean or a Gaussian filter. Alternatively, the smoothing can be integrated into the gray value interpolation, which is done automatically, if the interpolation method *'constant'* or *'weighted'* is selected. In the following sections, more information about this interpolation with integrated smoothing is given.

2.4.1 Equally Weighted Bilinear Interpolation (*'constant'*)

If *'constant'* is selected, the gray values are determined using equally weighted bilinear interpolation. To reduce the effects of aliasing in downscaled images, the smoothing of the image is integrated into the gray value interpolation in the following way: First, the resulting pixel is subsampled, i.e., it is divided into multiple parts. In the example depicted in figure 2.3, the back-transformed resulting pixel has the size $[1.5, 1.5]$ with respect to the pixels in the input image. It is sampled twice in each direction.

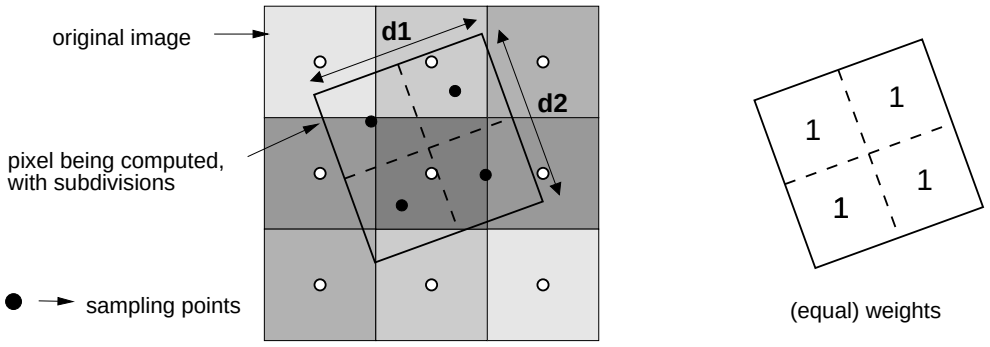

Figure 2.3: Determining a gray value using equally weighted interpolation.

Then, the gray value at each sample point is determined by bilinear interpolation in the input image. Finally, the gray value of the resulting pixel is computed by calculating the mean of the gray values of all sample points.

2.4.2 Gaussian Weighted Bilinear Interpolation (*'weighted'*)

If *'weighted'* is selected, the gray values are determined using Gaussian weighted bilinear interpolation. Similar to equally weighted interpolation, the gray value interpolation is executed in three steps: subsampling, determination of the gray values of the sample points, and calculation of the resulting gray value from the gray values of all sample points.

For *'weighted'* interpolation, however, different sampling and weighting methods are applied. The main difference is that now an area *larger* than the pixel is divided into *more* subpixels (see figure 2.4). The determination of the gray value of each sample point is carried out in the same way as for the method *'constant'* by bilinear interpolation. However, in contrast to equal weights used for the method *'constant'*, now a Gaussian smoothing is applied. Figure 2.4 shows the weighting for the case that the resulting pixel is sampled at 3×3 positions.

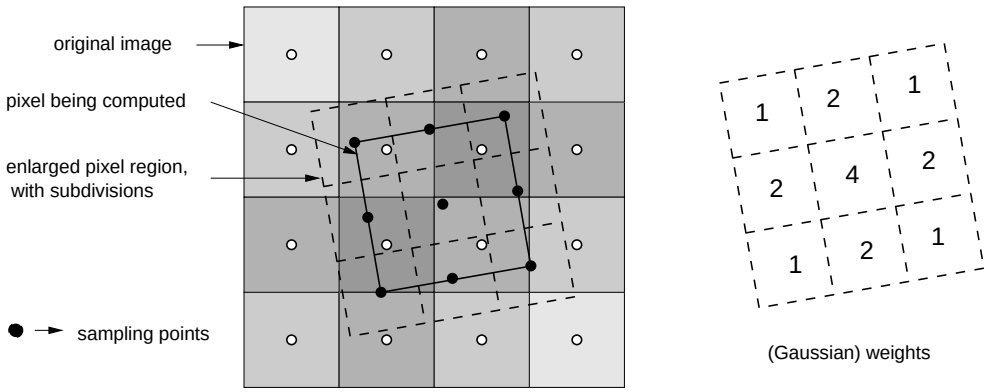

Figure 2.4: Determining a gray value using Gaussian smoothing.

The interpolation method *'weighted'* roughly corresponds to smoothing the image with a Gaussian filter before subsampling. Compared to equally weighted bilinear interpolation, this interpolation method produces output images with a smoother appearance. However, the runtime increases significantly for downscaled images.

2.5 Bicubic Interpolation (*'bicubic'* and *'bicubic_clipped'*)

If *'bicubic'* is selected, the gray values are determined using bicubic interpolation.

Bicubic interpolation considers a weighted average of 16 pixels surrounding the closest corresponding pixel in the image. Based on the gray values of these 16 pixels a bicubic polynomial surface is constructed. This surface is then used to determine the gray value at the requested (sub-)pixel position. This approach produces smoother interpolation results and leads to fewer interpolation artifacts compared to bilinear or nearest neighbor interpolation.

Note that the resulting gray values may lie outside of the gray value range of the given image. If the interpolation method *'bicubic_clipped'* is selected, these gray values are clipped to the gray value range of the input image. However, this method is only available for the determination of gray values at specific positions.

Note that bicubic interpolation may lead to unexpected results due to aliasing if the image is downscaled. To reduce these undesired effects, an interpolation method with integrated smoothing (see section 2.4 on page 8) should be used if a high quality of downscaled images is required.

2.6 Influence of the system parameter *'int_zooming'*

Another way to influence quality and speed of gray value interpolation is given by the system parameter *'int_zooming'*. This parameter determines if image transformations are performed with integer arithmetic or with floating point arithmetic. For *'int_zooming'* set to *'true'*, the transformation is carried out internally using fixed point arithmetic. This leads to much shorter execution times. However, it slightly decreases the accuracy of the transformed gray values. Also, for some operators, pixels with undefined gray values can occur. In some cases the domain region can slightly differ as well, depending on *'int_zooming'*. If *'int_zooming'* is set to *'false'*, the transformation is carried out with floating point arithmetic. This results in more accurate calculations. For byte images, the differences between both calculations is typically less than two gray levels. Correspondingly, for int2 and uint2 images, the gray value differences can be as large as 512 gray levels if the entire dynamic range of 16 bit is used.

The value of *'int_zooming'* is set to *'true'* by default. To change this value, the operator `set_system` can be used.

The system parameter *'int_zooming'* affects operators regarding geometric transformations of images (`projective_trans_image`, `projective_trans_image_size`, `affine_trans_image`, `affine_trans_image_size`, `polar_trans_image_ext`, `polar_trans_image_inv`, and `rotate_image`), scaling of gray values (`scale_image`) and zooming in on or out of images (`zoom_image_factor` and `zoom_image_size`).

Chapter 3

Examples

The following examples illustrate the effects of the different interpolation methods for pure rotation (see section 3.1), upscaling (see section 3.2), and downscaling (see section 3.3).

3.1 Rotation

In the first example, displayed in figure 3.1, the results for nearest neighbor interpolation, bilinear interpolation, and bicubic interpolation are compared for a pure rotation without scaling of the image.

Figure 3.1a) displays the original image containing a serial number of a bank note, where the characters are not horizontal. The image is rotated so that the serial number is horizontal using the interpolation methods *'nearest_neighbor'*, *'bilinear'*, and *'bicubic'*.

Figure 3.1b) displays the result of the rotation using nearest neighbor interpolation. In the detailed view, the typically undesired jagged appearance at the edges of the characters is clearly visible.

Figure 3.1c) displays the result of the rotation using bilinear interpolation. The edges of the characters now have a very smooth appearance. This is due to the fact, that bilinear interpolation considers four neighboring pixels instead of only one for the determination of the resulting gray value. Note that the results of the interpolation methods *'bilinear'*, *'constant'*, and *'weighted'* are the same for pure image rotations.

Figure 3.1d) displays the result of the rotation using bicubic interpolation. For pure rotations without upscaling, the result is very similar to that of bilinear interpolation.

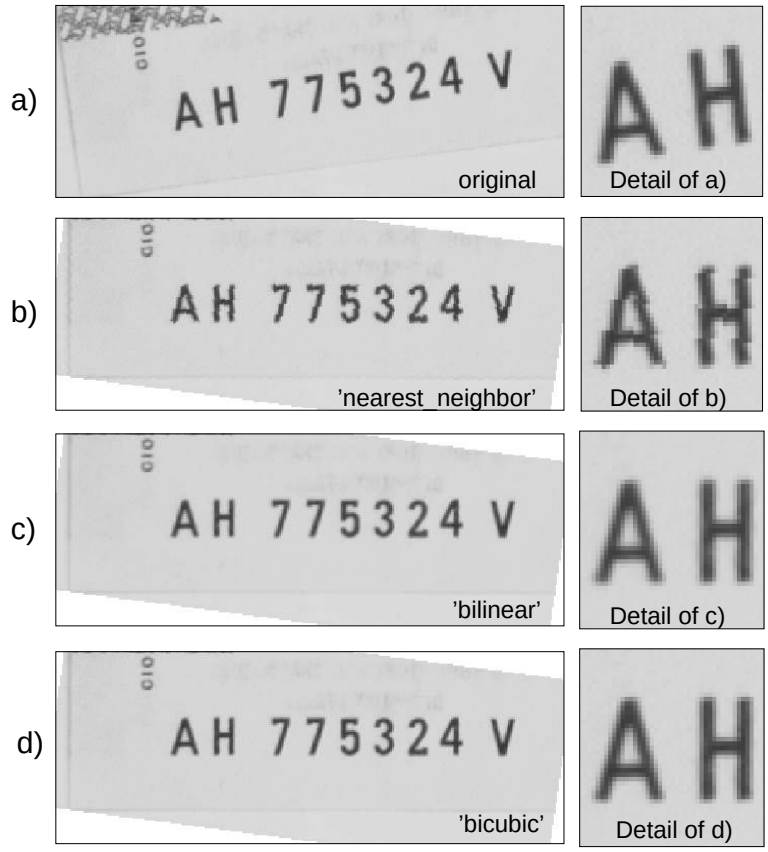

Figure 3.1: Comparison of the results for an image rotation using the interpolation methods *'nearest_neighbor'*, *'bilinear'*, and *'bicubic'*.

3.2 Upscaling

The example displayed in figure 3.2 illustrates the effect of an image transformation that includes upscaling with a scaling factor of 10. Here, the results for the interpolation methods *'nearest_neighbor'*, *'bilinear'*, and *'bicubic'* are compared. Figure 3.2a) displays the image to be transformed. It shows a part of a retainer. For a better presentation of the results, only enlarged views of the detail marked in figure 3.2a) are shown.

Figure 3.2b) displays the result when using nearest_neighbor interpolation. The pixel structure of the original image is clearly visible.

Figure 3.2c) displays the result when using bilinear interpolation. Although the edge appears much smoother, you can still perceive the pixel structure of the original image. Note that the results of the interpolation methods *'bilinear'*, *'constant'*, and *'weighted'* are the same for upscaling.

Figure 3.2d) displays the result when using bicubic interpolation. The edge appears nearly perfectly smooth. The pixel structure of the original image is no longer visible.

3.3 Downscaling

The example displayed in figure 3.3 on page 14 illustrates the effect of an image transformation that includes downscaling with a scaling factor of 0.1. Here, the results for the interpolation methods *'nearest_neighbor'*, *'bilinear'*, *'constant'*, *'weighted'*, and *'bicubic'* are compared. Figure 3.3a) on page 14 displays the image to be transformed. It shows two objects, a security ring on the left side and a retainer on the right side. For a better presentation of the results, a detailed view of each object is given for every interpolation method.

Figure 3.3b) on page 14 displays the result of downscaling the image using nearest neighbor interpolation. The result of this massive downscaling is of poor quality. Especially in both detailed views, it can be seen that the

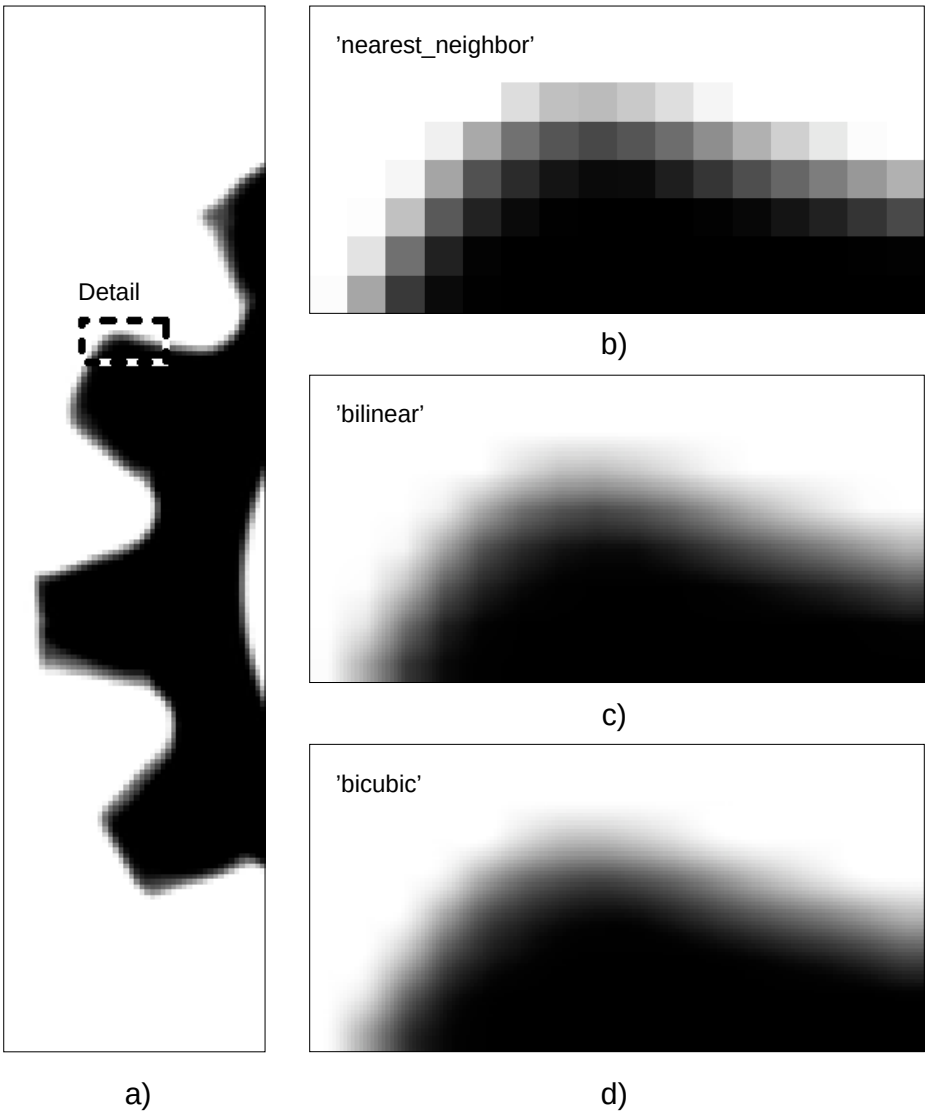

Figure 3.2: Comparison of the results for an image upscaling using (b) *'nearest_neighbor'*, (c) *'bilinear'*, and (d) *'bicubic'* interpolation.

shapes are often nearly straight whereas the shapes at the same part in the original image are completely different and with roundings.

Figure 3.3c) displays the result of downscaling the image using bilinear interpolation. The quality is slightly improved compared to nearest neighbor interpolation, but there are still heavy distortions in the shapes of the objects.

Figure 3.3d) displays the result of downscaling the image using constant interpolation. The downscaled image has a much smoother appearance and the shapes of the objects are similar to the ones in the original image. The quality of the downscaled image is significantly better compared to bilinear interpolation.

Figure 3.3e) displays the result of downscaling the image using weighted interpolation with an integrated Gaussian filter. The image is still smoother compared to *'constant'* interpolation and the shapes are nearly true to original. In the detailed view you can see, that the gradation of the edges is also finer. Thus, the result of *'weighted'* interpolation provides the best quality for image transformations that include downscaling.

Figure 3.3f) displays the result of downscaling the image using bicubic interpolation. For downscaling, the result is very similar to that of *'bilinear'* interpolation.

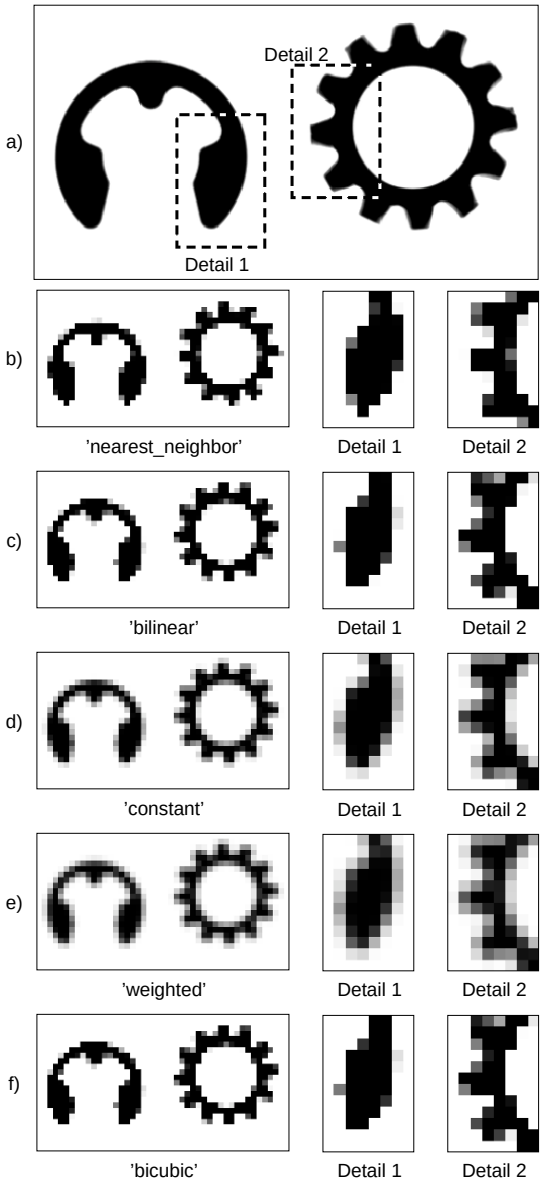

Figure 3.3: Comparison of the results for downscaling an image using *'nearest_neighbor'*, *'bilinear'*, *'constant'*, *'weighted'*, and *'bicubic'* interpolation.