Allied Vision

TECHNICAL PAPER

Alvium Convolution Mode

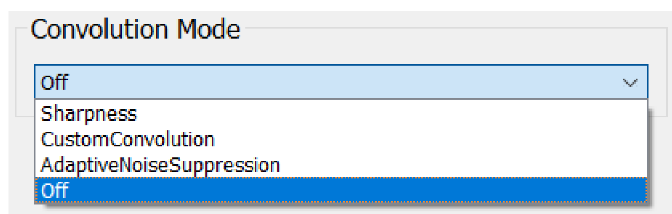
V1.0.0
Jan 2023

Benefits of Alvium on-camera convolution

Image convolution is a versatile image processing tool with a broad variety of use cases. But on a CPU, the computational cost of image convolution is a major drawback. This applies especially to scenarios with high frame rates, high image resolution, or large convolution kernels.

The complete Alvium camera product line with GenAPI support features an on-board hardware convolution engine, enabling 5x5 convolution image processing directly on the camera at full bandwidth. As of this writing, the Custom Convolution matrix is unique in the market.

The Alvium convolution engine can be controlled in the following modes:



You can apply your own convolution algorithms using a GUI or code.

Image Convolution – technical backgrounds

Mathematically, image convolution is a discrete 2D convolution:

$$f(x,y) = \kappa * g(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} g(x+i, y+j)\, \kappa(i,j)$$

Where $f(x,y)$ is the resulting filtered image, κ is a 2D kernel matrix with a size of $m = 2a + 1$ times $n = 2b + 1$ and $g(x,y)$ is the original image.

Figure 1 shows an example how a single output pixel value is calculated from the input image and the filter kernel:

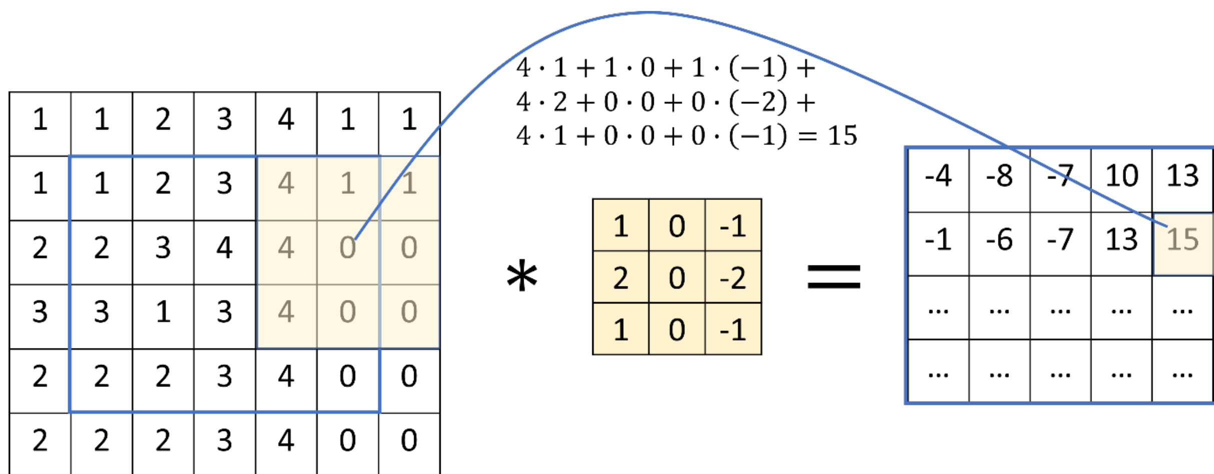


Figure 1 Example Convolution

It shows an image convolution with a 3x3 vertical edge detection kernel. Each pixel of the resulting image is calculated only with the original pixel and its neighbors by multiplication with the corresponding kernel coefficient. However, every pixel needs to be calculated with the sum of $m \cdot n$ multiplications. In a simple implementation of a not separable convolution, a source image with a resolution of $X \cdot Y$ would therefore lead to a complexity of $O(XYmn)$.

Note: The camera hardware convolution engine provides an efficient real-time solution for convolution, but it increases the power consumption of the camera.

The figure also shows that the resulting image turns out smaller than the original. This is reasoned in the missing neighborhood pixel values when the kernel overlaps at the image edges and corners. A calculation cannot be performed there without further consideration of the problem. Different strategies for this issue exist. To obtain the original image size in the result we are using a mirroring approach on the source image.

To perform a 5x5 convolution, 2-pixel rows and columns are mirrored. The 4 pixels in each image corner are mirrored diagonally. This strategy is inducing a lower error than setting the unknown pixels to zero. Nevertheless, convolution results at the image edges must be interpreted with care.

The above convolution example results in negative pixel values. This shows that the result of the bare convolution operation does not necessarily lead to valid images. To achieve valid images a constant e.g., 128 for 8-bit pixel format, is added if the sum of the convolution kernel κ is 0 resulting in a grayscale image. Otherwise, it is assumed that no change to the average image intensity is intended, and the convolution kernel is normalized in the following way:

$$\kappa' = \frac{1}{\sum_{i,=-a}^{a} \sum_{j=-b}^{b} \kappa(i,j)} \kappa$$

Note: Normalization is calculated in the camera. When specifying a kernel matrix, the user is free to use any integer from -256 to 255 without considering normalization.

Convolution of color images

In many cases, convolution of an RGB image is done by separate convolutions of each color channel. Because this further increases the computational complexity, we chose another approach for the on-camera convolution. Generally, the interesting information for image convolution is the brightness relation to neighborhood pixels, e.g., for enhancing sharpness. The YUV color model is separating this luminance information in the Y channel from the chroma information. Therefore, the convolution in our Alvium cameras is done only on the Y-channel and the original chroma values stay untouched. After applying the convolution, the camera performs the conversion to back to RGB for convenient output formats.

Note: Kernels which sum up to zero are *DC free* and removing the luminance information in uniform areas. It is recommended to use **mono pixel format** for DC free kernels!

Getting started

Prerequisites

For an easy start with Alvium's convolution mode, you need:

- Vimba X SDK
- Alvium GigE, USB, or 1800-C CSI2 cameras
- Alvium camera firmware 00.10.00.6c9062b1 or higher

Technically, you can also use the Vimba SDK. However, only Vimba X contains a GUI to easily apply the desired convolution settings.

Get started

Step 1: Get your camera up and running

- Check the firmware version. If a newer firmware is available, update it.
- Start the Viewer provided with the SDK, acquire some images, and apply the basic camera settings for your application such as the exposure time. GigE cameras: For best performance, follow the instructions of the user guide.

Step 2: Use Vimba X Viewer

On the *Image Processing tab*, you can select the convolution presets or apply a custom convolution with the matrix.

Note: For sharpening, blur (negative sharpening) and adaptive noise reduction, Allied Vision provides extra image processing features. These are using the same convolution hardware and cannot be used simultaneously with a custom convolution.

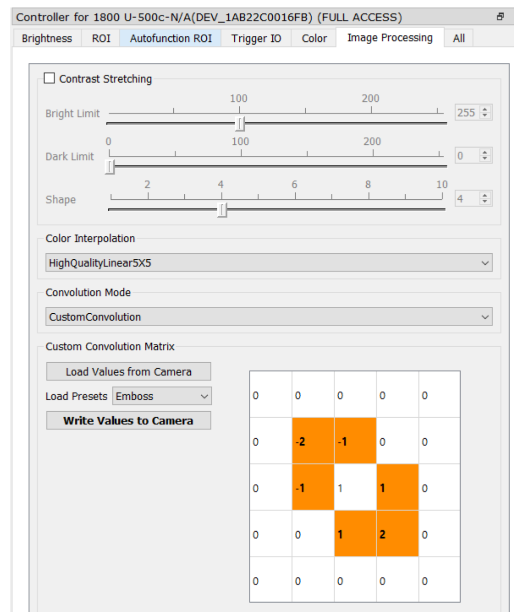


Figure 2 Vimba X Viewer Convolution Widget

The widget enables the user to choose between several presets. These are then loaded in the kernel matrix widget and can be further edited there. The final values must then be written to the camera. After this step, the convolution is configured, and the result can be observed by clicking *Acquisition Start.* The convolution coefficients are not changeable during acquisition. Kernels written to the camera can be stored in user sets. Custom presets can be added to the dropdown menu by editing *Convolution_Presets.json* which is located in *\bin\plugins* of the VimbaX installation folder.

Step 3: Use the API

This step is optional, if you want to further develop your custom convolution.

As shown above, the Viewer widget offers a convenient solution for testing convolution kernels. But for an application, it is most likely necessary to specify the convolution kernel with the API. The way to do this is using the XML camera feature interface: The *CustomConvolutionValue* feature represents one value of the kernel at the position defined by the *CustomConvolutionValueSelector* Enum feature. The behavior of it can be tested in the Viewer feature tree. Also, the Enum feature *ConvolutionMode* needs to be set to “CustomConvolution” to activate the convolution engine.

The following code example shows how the Vimba X C++ API can be used to load a 5x5 kernel matrix to the camera.

```
// Code snippet for specifying a custom convolution kernel with VmbCPP API
#include <iostream>
#include "VmbCPP/VmbCPP.h"
using namespace VmbCPP;

int main()
{
    VmbSystem& system = VmbSystem::GetInstance();
    if (VmbErrorSuccess == system.Startup())
    {
        CameraPtrVector cameras;
        if (VmbErrorSuccess == system.GetCameras( cameras ))
        {
            CameraPtr camera = cameras[0];
            if (VmbErrorSuccess == camera->Open( VmbAccessModeFull ))
            {
                // Specify Custom Kernel
                int kernel[5][5] = {    { 0, 1, 2, 3, 4},
                                        {10,11,12,13,14},
                                        {20,21,22,23,24},
                                        {30,31,32,33,34},
                                        {40,41,42,43,44}    };

                // Get necessary Features
                FeaturePtr pConvolutionValueSelector;
                FeaturePtr pConvolutionValue;
                FeaturePtr pConvolutionMode;
                camera->GetFeatureByName( "CustomConvolutionValueSelector", pConvolutionValueSelector );
                camera->GetFeatureByName( "CustomConvolutionValue", pConvolutionValue );
                camera->GetFeatureByName( "ConvolutionMode", pConvolutionMode );

                // Specify Custom Convolution Mode
                pConvolutionMode->SetValue( "CustomConvolution" );

                // Set kernel values via camera features
                int n = 0;
                for (int i = 0; i < 5; ++I)
                {
                    for (int j = 0; j < 5; ++j)
                    {
                        pConvolutionValueSelector->SetValue(n);
                        pConvolutionValue->SetValue( kernel[i][j] );
                        n++;
                    }
                    camera->Close();
                }
            }
        }
        system.Shutdown();
        return 0;
```

Allied Vision

Some kernels and their effects

The following examples were taken with the custom convolution mode of an Alvium 1800 U 500c in 8bit mono pixel format. The images are cropped to 640x640 pixels to increase the visibility of the processing impact. The identity kernel has no effect on the image and is in the following examples shown left as reference image.

Bluring is done by mixing the pixel values within the pixel Neigborhood. This can be done equally, e.g. Box Blur or weighted e.g. Gaussian Blur.

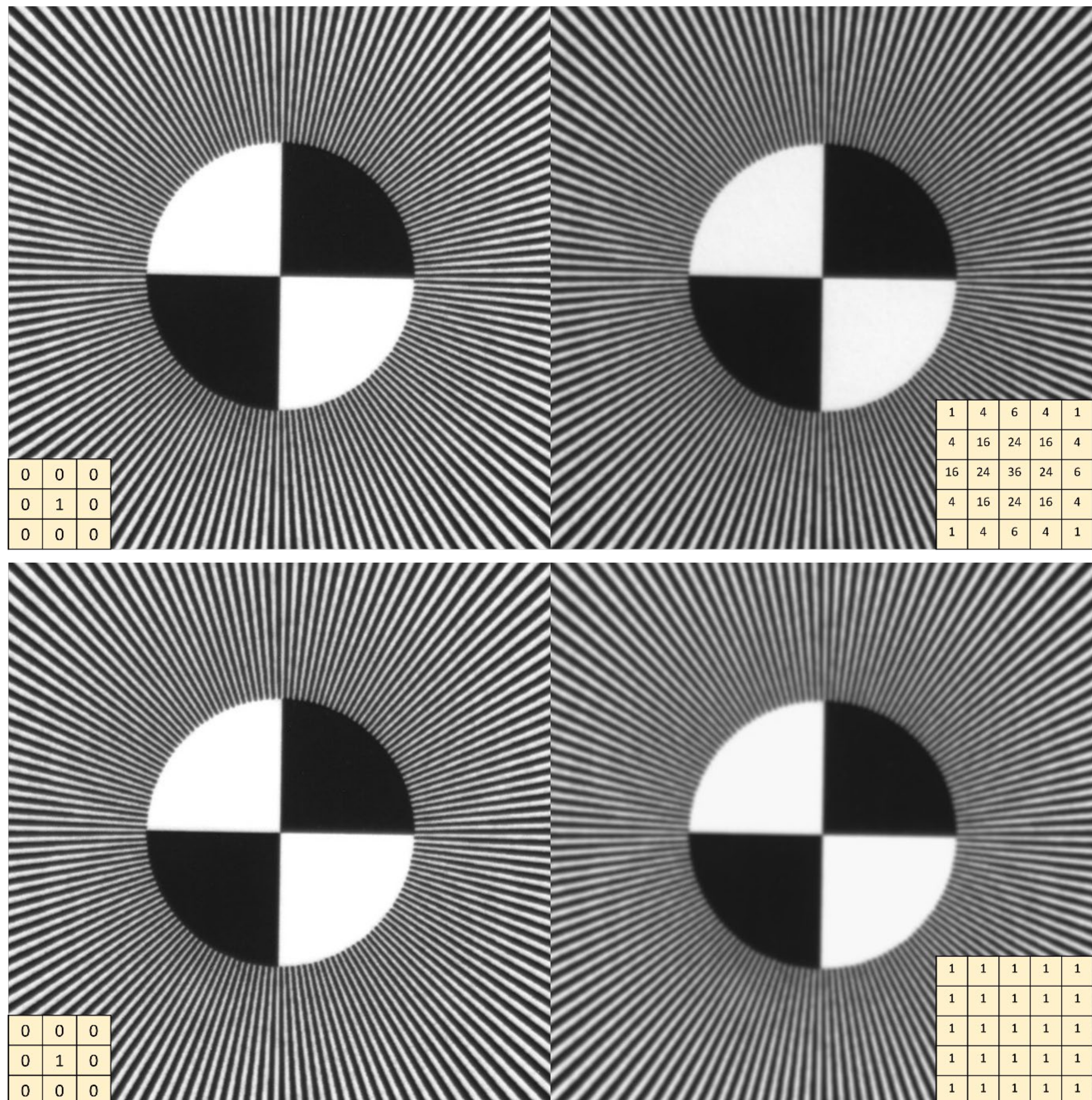


Figure 3: Top: 5x5 Gaussian Blur Convolution - Bottom: 5x5 Box Blur Convolution

Contrary to blurring, the image edges can be enhanced using a variety of kernels. One is a sharpening kernel, increasing the contrast between neighboring pixels. Another option are embossing filters, which create a 3D effect at edges by applying directional differences.

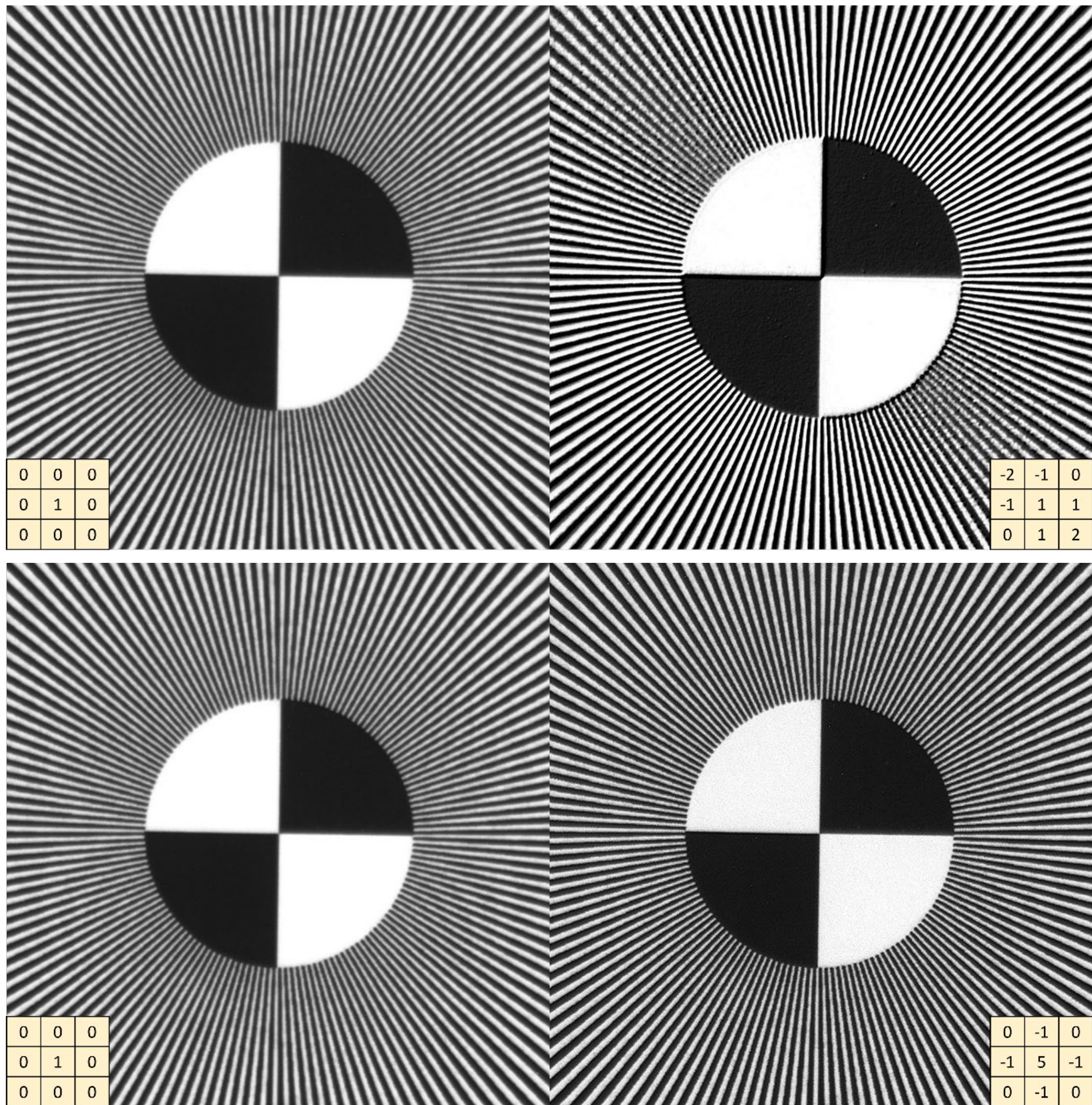


Figure 4: Top: 3x3 Emboss Convolution - Bottom: 3x3 Sharpening Convolution

The Sobel operator is based on two convolutions which approximate the image derivate in one direction. This can be used for edge detection. These Kernels are DC free.

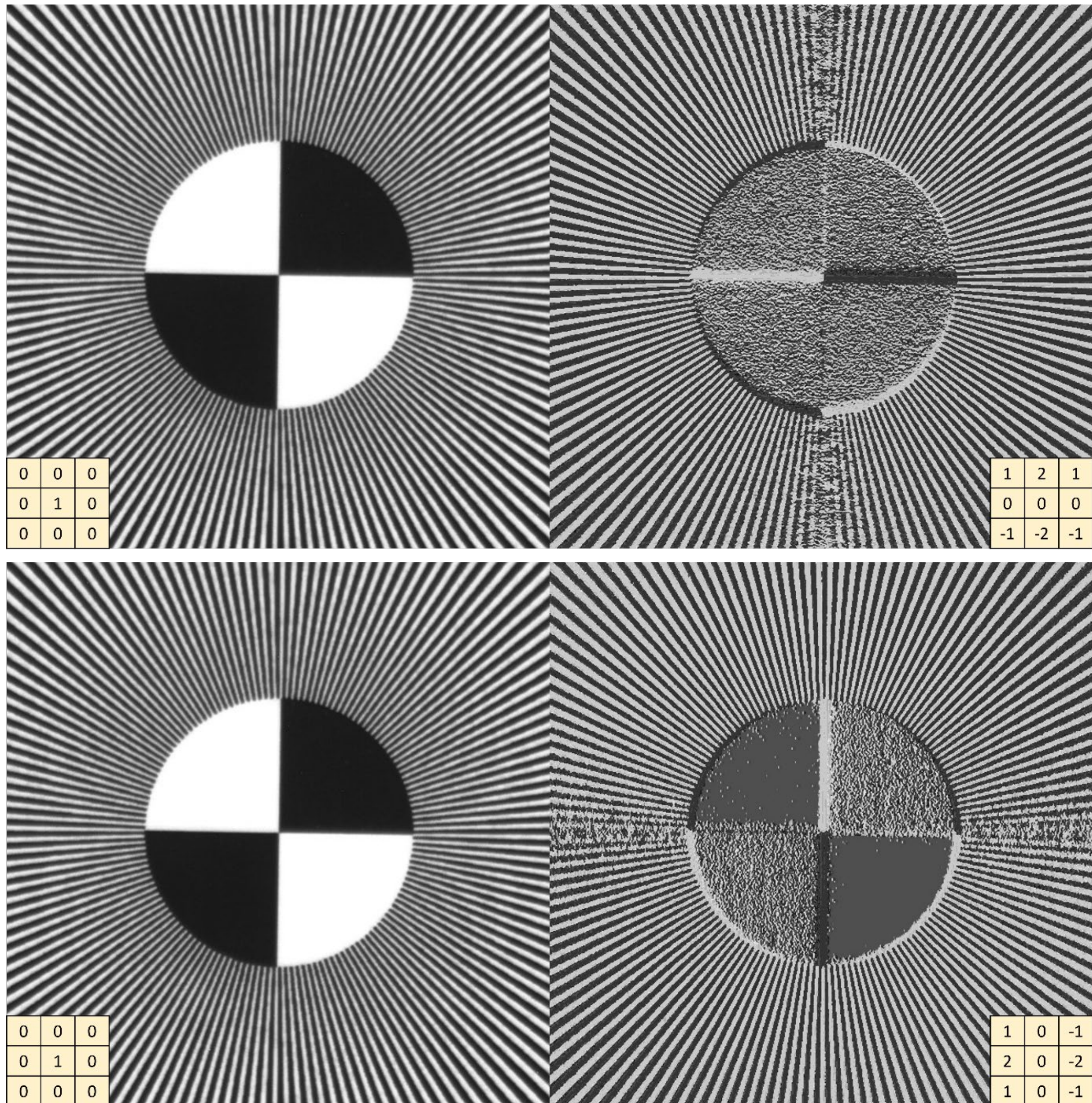


Figure 5: Top Sobel Horizontal Edge Convolution - Bottom: Sobel Vertical Edge Convolution