Basic Digital Image Processing in Python (using OpenCV and numpy)

Contents

The goal of this practice session is to get familiarized with python and standard scientific computing packages used for computer vision (such as OpenCV and numpy) in the context of performing basic image processing operations, such as filtering.

If you already know these tools, you can skip this session. Its intention is that all of us in class are "on the same page" for the next exercises.

1 Python

Since we will use ROS (Robot Operating System - Kinetic or Melodic versions) and it is starting to transition toward using Python 3 in its Noetic version, we review here using Python 2.7. Feel free to adapt the code to Python 3 if you want (and let your colleagues in class know about it).

Hence, we use Python 2.7 and numpy, opencv2 packages in Linux.

There are several ways to install these.

Installation of numpy using pip: *sudo apt update; sudo apt install python-pip; pip install numpy*

Installation of opencv using pip: *pip install opencv-python*

See https://pypi.org/project/opencv-python/

You may also install OpenCV using a package manager, such as Synaptic Package Manager.

ROS: Another possible way to install opencv2 that will be useful for future sessions is to get it from ROS. The *Desktop Install* version of ROS should suffice (http://wiki.ros.org/kinetic/Installation/Ubuntu). Both versions, ROS Kinetic (2016) or ROS Melodic (2018), should work. Example installation command: *sudo apt-get install ros-kinetic-desktop*

In your own time, feel free to go beyond the exercises in this session. You may follow a basic tutorial in python and numpy to know how to create variables and how to access their content. You may follow OpenCV tutorials to play around with image data.

1.1 Script

For this exercise we use the python code in the script `ex1_improc.py`

The script is incomplete. The parts marked with ??? need to be completed for proper execution.

With the script we aim to learn the following:

- How to load an image an image from disk
- How to write an image to disk (image which may include negative values)
- How to plot image without visual artefacts
- What the spatial and intensity resolutions of an image are
- How to linearly filter an image with a kernel.
- How to compute derivatives of image data:
  - In space, within an image (Sobel operators)
  - In time, using two images (forward difference formula)

2 Basics of Digital Image Processing

A classic reference on Digital Image Processing is the book by Gonzalez and Woods, "Digital Image Processing", now in its 4th edition (2018). The same authors also have a more practical book called "Digital Image Processing Using MATLAB" (3rd edition). These are just some references in case you want to read more about image processing.

2.1 Digital Images

Monochrome (grayscale) images can be modeled by two-dimensional functions $f : \mathbb{R}^2 \to \mathbb{R}$. The amplitude of f at spatial coordinates (x, y), i.e., $f(x, y)$, is called the *intensity* or *gray level* at that point, and it is related to a physical quantity by the nature of the image acquisition device; for example, it may represent the energy radiated by a physical source. We will deal with bounded (i.e., finite) quantities, and so $|f| < \infty$. Common practice is to perform an affine transformation (substituting $f \leftarrow af + b$ for all (x, y) by means of some suitable constants a, b) so that f takes values in a specified interval, e.g., $f \in [0, 1]$.

A *digital image* can be modeled as obtained from a continuous image f by a conversion process having two steps: sampling (digitizing the spatial coordinates x, y) and quantization (digitizing the amplitude f). Therefore, a digital image may be represented by an array of numbers, $M = (m_{ij})$, where i, j and m can only take a finite number of values, e.g., $i = \{0, 1, \dots, W-1\}$, $j = \{0, 1, \dots, H-1\}$ and $m = \{0, 1, \dots, L-1\}$ for some positive integers W, H, L (Width, Height and number of gray Levels). That is, a digital image is a 2-D function whose coordinates and amplitude values are discrete (e.g., integers). Specifically,

$$m_{ij} = q\big(f(x, y)\big) = q\big(f(x_0 + i \cdot \Delta x,\, y_0 + j \cdot \Delta y)\big),$$

where Δx and Δy are the sampling steps in a grid with spatial coordinates starting at some location (x_0, y_0), and $q : \mathbb{R} \to \{0, \dots, L-1\}$ is the input-output function of the quantizer (which has a staircase shape).

Common practice for grayscale images is to use 1 byte to represent the intensity at each location (i, j) (i.e., picture element or "pixel"). Since 1 byte = 8 bits, the number of possible gray levels is

$L = 2^8 = 256$, and so intensities range from $i = 0$ (black) to $i = L - 1 = 255$ (white). Hence, images are typical stored and read as unsigned character (unsined int8 variables). However, to numerically operate with grayscale images, it is convenient to convert the data type of the image values from integers to real numbers, i.e., from 8 bits to single or double precision (float type). Once operations are finished, it may be convenient to convert back to 8 bit format for storage of the resulting image, thus producing a quantization of the data values. Medical images are usually represented with a larger depth (10-12 bits) to mitigate the occurrence of visual artifacts due to the quantization process.

Color images can be represented, according to the human visual system, by the combination of three monochrome images (with the amount of red (R), green (G) and blue (B) present in the image), and so, each pixel is represented by 3 bytes, which provides a means to describe $2^{3\times 8} \approx 16.8$ million different colors. Many of the techniques developed for monochrome images can be extended to color images by processing the three component images individually.

The number of bits required to store a (grayscale) digital image is $b = W \cdot H \cdot \log_2(L)$, and so compression algorithms (such as JPEG) are essential to reduce the storage requirement by exploiting and removing redundancy of the image.

Spatial resolution is a measure of the smallest discernible detail in an image, and it is usually given in dots (pixels) per unit distance. That is, it is the density of pixels over the image, but informally, the image size ($W \times H$ pixels) is regarded as a measure of spatial resolution (although it should be stated with respect to physical units - cm, etc.). It is common to refer to the number of bits used to quantize intensity as the ***intensity resolution***.

2.2 Loading an image

To read an image from disk, we use the OpenCV command imread:

```
>> img = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
```

It will return a variable called `img` of type `uint8` (unsigned integer represented by 8 bits = 1 byte) and it is a multi-dimensional array/matrix of size $N_{\text{rows}} \times N_{\text{cols}} \times N_{\text{channels}}$. If $N_{\text{channels}} = 3$ there are three channels or “bands”, thus, it is a color image (typically in RGB format). In the additional arguments of the function we indicate to read as grayscale image, so it only has one channel.

We can inspect the size of the image by using the command

```
>> print img.shape
```

Let us convert to floating point values (for numerical manipulation), so that we are not restricted to operations with only 8 bits (256 representation levels):

```
>> img = img.astype(float)
```

2.3 Displaying an image and coordinate axes convention

To visualize the image that has been previously loaded in variable `img`, you may use matplotlib.pyplot’s commands `imshow` or `figimage`. The command `imshow` causes some artefacts (e.g., moire patterns) by resizing the image, so we use the command `figimage` instead.

```
>> plt.figure()
>> plt.figimage(???)
>> plt.show()
```

This will create a new window and display the image. By default, it does not use a grayscale colormap, so the image will be colored (even though it is supposed to be a grayscale image). Investigate the documentation of figimage to change the colormap from the default one to a grayscale one.

In images, the coordinate system origin is at the upper left corner, with the positive y axis extending downward (along the rows of the matrix variable `img`) and the positive x axis extending to the right (along the columns).

Note that you may resize the window of the image, but it does not resize the image or change its aspect ratio (does not compress or expand the image in any dimension).

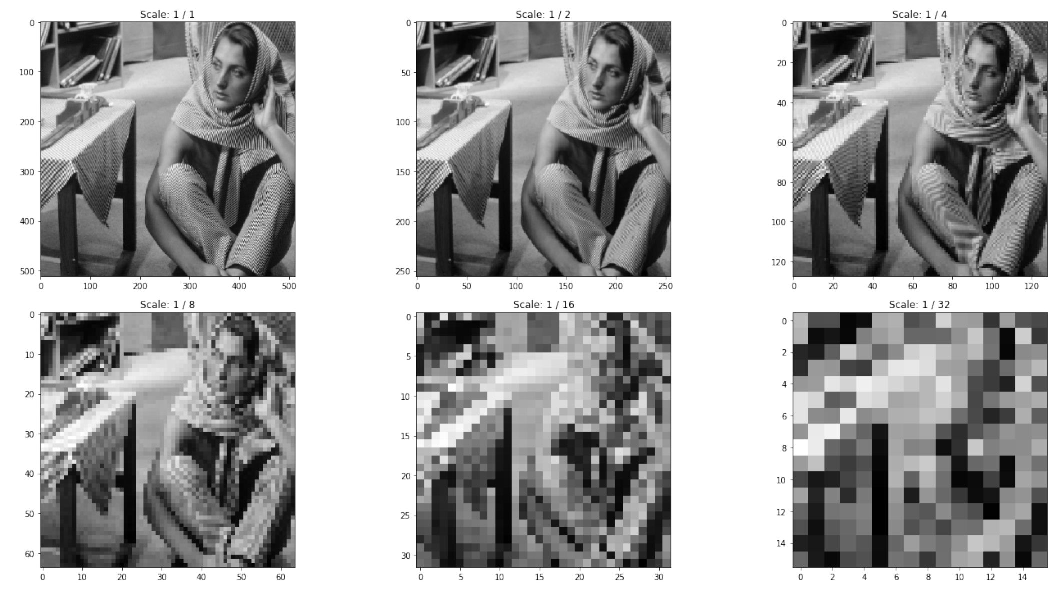


Figure 1: Effect on image quality of reducing the spatial resolution of the image, i.e., the number of samples (pixels) used to represent the underlying continuous intensity signal. This image, called "Barbara" is a standard one in image processing because it has many regions of large intensity variations ("high spatial frequencies"), such as edges, lines. It has also some "low spatial frequencies" regions, such as the skin and the floor. It is good for testing purposes. Feel free to run the script on other images that have same or different characteristics.

To select a subregion of an image (also called ROI - Region Of Interest) we may use Python slice notation. For example, to select the top-left 300 × 400-pixel region of `img` we type:

```
>> img_sub = img[ 0:300, 0:400 ]
```

There is another example in the Python script to select a central region of the image by "cropping" 10 pixels from the top and the bottom and 30 pixels from the sides. Please take a look.

2.4 Writing an image to disk

To save an image to disk, we use the OpenCV command imwrite:

```
>> cv2.imwrite('SavedImage.png', img)
```

This will create a file called SavedImage.png in the working folder.

2.5 Resolution

2.5.1 Spatial resolution

Given a well-sampled digital image, the effect of reducing the spatial resolution (while keeping the number of intensity levels constant) is shown in Fig. 1

The image quality degrades as it is represented with fewer number of pixels. The coarsening effect or "pixelization" is evident when an insufficient number of pixels is used; such number of samples cannot capture the fine details present in the scene.

The Python script has code to generate an figure like this one. You would need to know how to use the OpenCV command `cv2.resize` and the pyplot function `imshow` to complete the loop that generates an image like Fig. 1. The loop variable called scale_factor indicates the value by which the original image is downsized (in each dimension, x, y)) to get each image in the figure.

2.5.2 Intensity resolution

Next, we take a look at the effect of maintaining the spatial resolution but decreasing the intensity resolution. That is, we maintain the number of image pixels, but we use fewer bits to encode them.

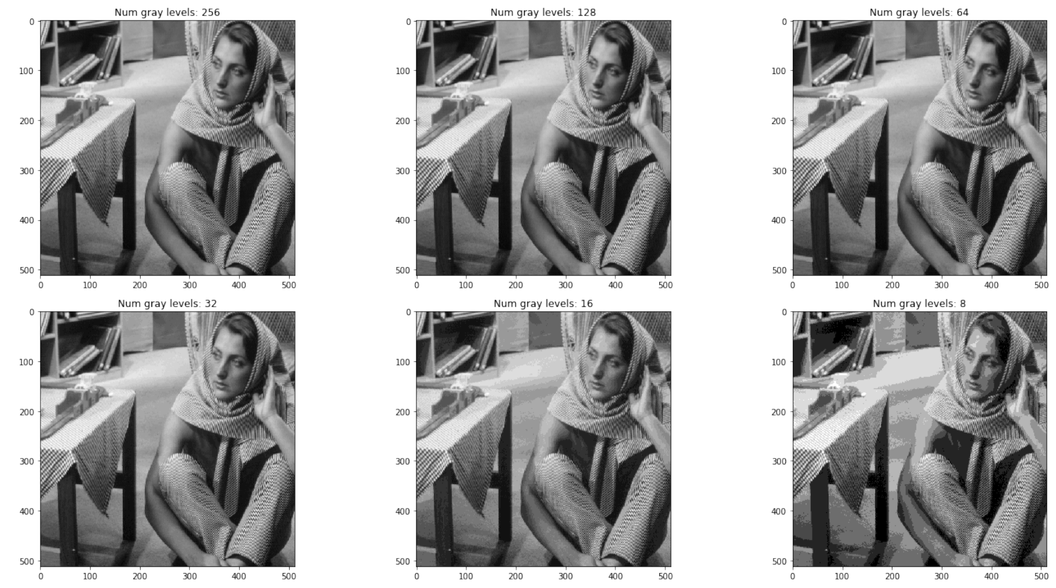


Figure 2: Effect on image quality of reducing the intensity resolution of the image, i.e., the number of gray levels used to represent the underlying continuous intensity signal.

The output produced on a given image with 256 intensity levels is depicted in Fig. 2.

Can you discern the differences between the resulting images?

How many intensity levels does the human visual system need in order to represent the scene faithfully?

Observe that, as the number of levels decreases, *false contours* appear in smooth intensity regions. In the bottom right image of Fig. 2, can you discern the 8 intensity levels used to represent the image? Which test image would you use to clearly see the number of reduced intensity levels?

Think of it and then take a look at Fig. 3.

For this part of the script, you would need to remember that division of integer-type variables produces integers and therefore discards fractional information, effectively producing the desired quantization effect. Feel free to use the code provided in the script or design simpler ways to generate a figure like Fig. 2 by processing the original image (in the top left of the figure).

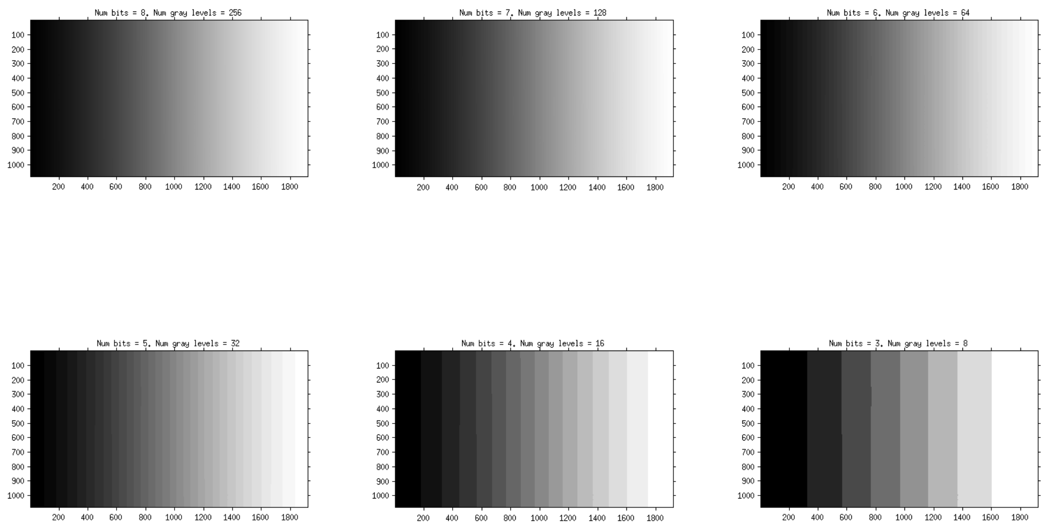


Figure 3: Effect on image quality of reducing the intensity resolution of the image, i.e., the number of gray levels used to represent the underlying continuous intensity signal.

2.6 Filtering

2.6.1 1-D (Linear) Filtering. Convolution

The convolution of two sequences of numbers ("signals") $a[n]$ and $b[n]$, $n \in \mathbb{Z}$, is symbolized by $c = a \star b$ and calculated according to

$$c[n] = \sum_{k=-\infty}^{\infty} a[k]b[n-k]. \tag{1}$$

The convolution is commutative ($a \star b = b \star a$), so a and b can be swapped in the previous formula. In practice, we use sequences of finite length, so the summation in (1) is carried over a finite number of products.

A demonstration of the convolution is illustrated in Fig. 4, where two signals a, b are defined and their convolution is computed. In the folder accompanying this script there is also a movie of the way that each coefficient of the output sequence, c, is computed.

Linear filtering is implemented by linear shift-invariant systems, whose output consists of the convolution of the input signal (a) and the impulse response of the filter (b), i.e., $c = a \star b$. For images, we will be using the 2-D convolution implemented in OpenCV's command `cv2.filter2D`.

2.6.2 2-D (Linear) Filtering.

The convolution operation can be extend to two-dimensional discrete signals, i.e., monochrome images. The convolution of two digital images h, f is written as $g = h \star f$ and calculated by

$$g[i,j] = \sum_u \sum_v h[u,v] f[i-u, j-v]. \tag{2}$$

Often, h is the filter (called "kernel" or "mask") and f is the input image. Then, due to the commutativity of the convolution, $g = h \star f = f \star h$, it is standard to think of g as computed by reversing and shifting h (the kernel) rather than by reversing and shifting the input signal f in (2).

Each output pixel is computed by a weighted sum of its neighbors in the input image, and if h is a filter with a kernel of size $n \times n$ pixels, this is the size of the neighborhood, and it implies n^2 multiplications. For example, if h is a filter with Gaussian shape (samples of a Gaussian function on an $n \times n$ grid), each output pixel is computed as a Gaussian weighted average of the input pixels in the neighborhood. The output g will be smoother than the input f since high frequencies have been attenuated by the filter h. You could see this by plotting the Fourier Transform of h, but this

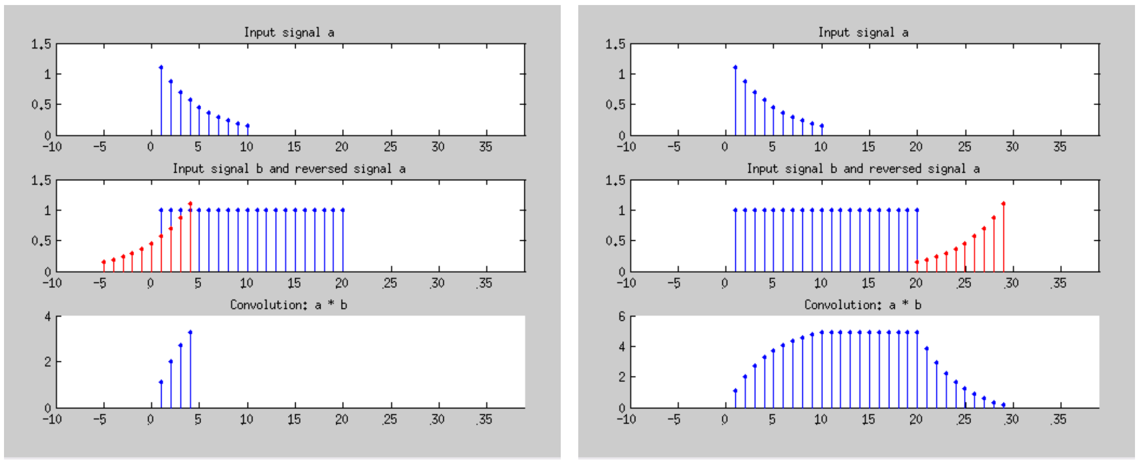


Figure 4: Example of 1-D convolution computation. One signal (middle plots, in blue) is multiplied by a reversed and shifted version (middle, red) of the other signal (top); the sum of the product gives one sample of the output signal (bottom).

is optional for you to show.

Separable filters are of special importance since they save a lot of computations. A filter is separable if its kernel can be written as the product of two independent 1-D kernels, i.e., $h[u,v] = p[u]q[v]$. That is, the matrix $h[u,v]$ can be written as the (exterior) product of two vectors $p[u]$ and $q[v]$. This reduces the cost of convolution; the filtered image can be obtained as a cascade of the two 1-D convolutions, with $2n$ multiplications per pixel instead of the n^2 count in the general, non-separable case. This is internally handled by OpenCV in popular filters, and there is a specific function for that in case you are curious about it: `cv2.sepFilter2D`.

2.6.3 Applications: Blurring and Noise Reduction

Let us show some standard (2-D) image filtering. First, we load an image and pass it through a low-pass filter, for example, a box kernel or a Gaussian kernel. A box kernel, such as

$$h[u,v] = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

that is,

$$h[u,v] = \frac{1}{(2W+1)^2}, \quad u,v \in \{-W,\dots,W\}$$

is a square one (where W is half of the kernel size in each dimension) with the same value in all pixels (and they sum up to one), so that when used in a convolution it computes the average of the input pixel values (in a $(2W+1)\times(2W+1)$ neighborhood).

In OpenCV, the box filter is implemented in the function `cv2.blur`. In the script there is sample code of this function. Check out the input arguments of such a function to understand their meaning.

A Gaussian kernel $h_\sigma[u,v]$ is an approximation of the (2-D) Gaussian function:

$$\begin{aligned} \mathbb{G}(x,y;\sigma) &= \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right), \qquad (3) \\ h_\sigma[u,v] &= \mathbb{G}(u,v;\sigma), \quad u,v \in \{-W,\dots,W\} \end{aligned}$$

Is this filter separable? Justify your answer.

The OpenCV function `cv2.GaussianBlur` implements this filtering operation. Check out the documentation to understand their input arguments and complete the code. If the kernel size is not specified, as written by the tuple (0,0), then the size is automatically computed from the value of the blur parameter(s) σ.

```
# %% Smoothing by Gaussian filtering
s = 2. # Parameter that controls the amount of smoothing
if s > 0:
    img_gauss = cv2.GaussianBlur(img, ???)

fig = plt.figure()
fig.suptitle('Gaussian smoothing')
plt.figimage(img_gauss, cmap='gray')
plt.show()
```

Comparing the original and the filtered image, can you describe the process that took place? Where edges smoothed equally in all directions (i.e., is the filter "isotropic")?

Try other parameters of the filter (kernel size and σ), and see the corresponding output. For example, try $\sigma = 5$ or a large kernel size.

Optional: Add Gaussian noise to the original image (i.e., variations in intensity drawn from a Gaussian normal distribution and filter the noisy image with a Gaussian kernel, as before. Try, for example, a standard deviation of noise $\sigma_n = 15$ intensity levels, and try to vary the parameters of the filter to remove noise while preserving the details of the underlying clean image. Sample images before and after Gaussian noise removal are given in Fig. 5.

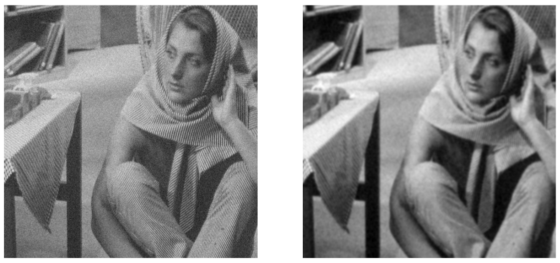


Figure 5: Left: noisy image; Right: filtered ("de-noised") image.

Also optional: Add "salt & pepper noise" (random occurrences of black and white pixels) to the original clean image. Then, use a linear filter (Gaussian) and a non-linear filter (the median filter) to try to remove the noise, as shown in Fig. 6. Observe that linear smoothing filters do not alleviate salt and pepper noise whereas the median filter does an excellent job.

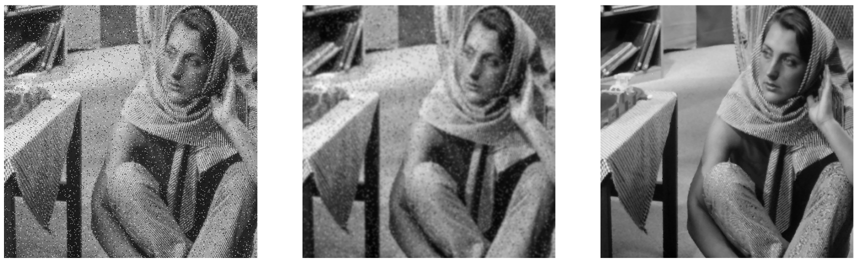
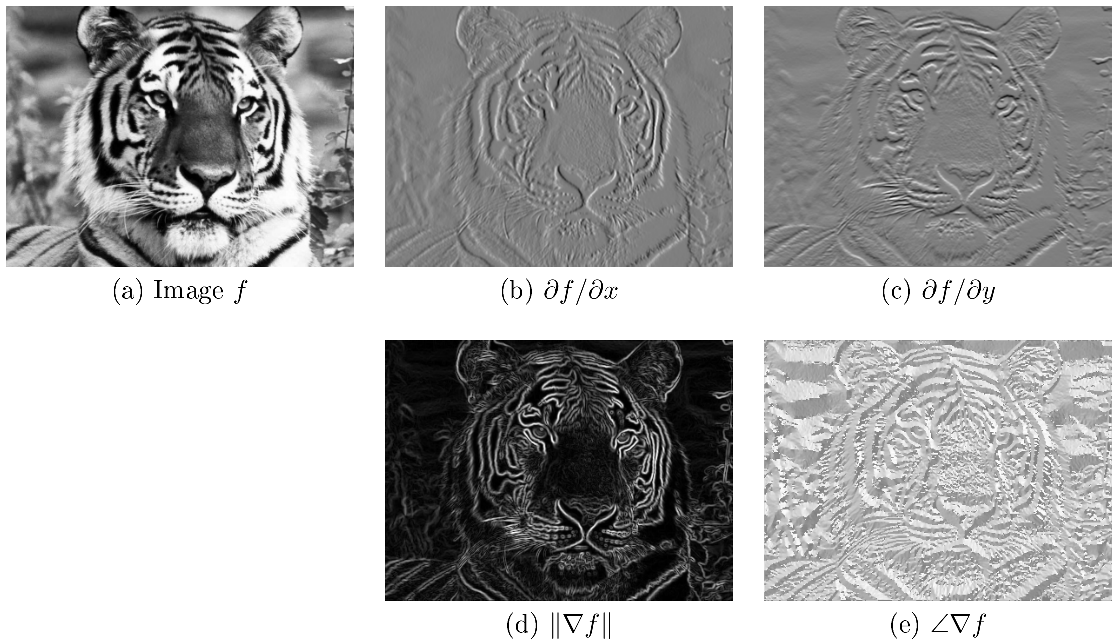


Figure 6: Salt & pepper noise (left): Gaussian (center) vs. median filter (right) noise removal.


(a) Image f (b) $\partial f/\partial x$ (c) $\partial f/\partial y$

(d) $\|\nabla f\|$ (e) $\angle \nabla f$

Figure 7: Edge detection via linear filtering. (a) original image f; (b) approximation to $\partial f/\partial x$ (which detects transitions along x, i.e., vertical edges); (c) approximation to $\partial f/\partial y$ (which detects transitions along x, i.e., horizontal edges); (d) gradient magnitude $\|\nabla f\|$; (e) gradient angle. (d) and (e) is the same information as (b) and (c) but in "polar coordinates".

2.6.4 Application: Computing Spatio-temporal Derivatives.

Spatial derivative calculation. Sobel filter. Linear filtering can be used to compute numerical approximations to the partial derivatives of an image. This is usually accomplished by filters kernels such as Sobel. Note that the kernel to compute the partial derivative with respect to y is just the transpose of the kernel to compute the partial derivative with respect to x.

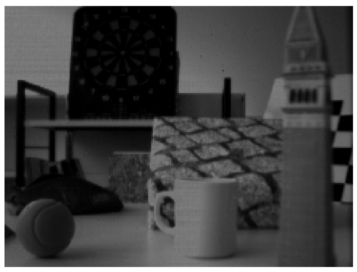
In OpenCV, there is already a function that implements these standard kernels: `cv2.Sobel`
Are Sobel kernels separable?
What is the interpretation of the Sobel kernels?
Can you relate the kernel to the central difference approximation formula for the first derivative?

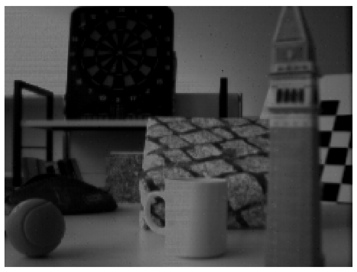
The following sample code computes the gradient images ($f_x \doteq \partial f/\partial x$ and $f_y \doteq \partial f/\partial y$, shown in Fig. 7).

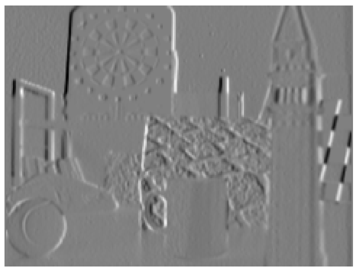
```
grad_x = cv2.Sobel(img, cv2.CV_64F, 1,0)
grad_y = cv2.Sobel(img, cv2.CV_64F, 0,1)
```

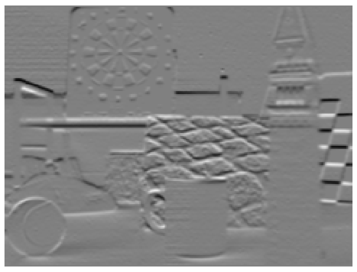
Since kernels to compute derivatives have signed values, the result of the convolution will also be signed (values at a pixel can be positive or negative). Because the pyplot functions `figimage` or `imshow` display the minimum value as black and the maximum as white, an image with positive and negative values will display intensities around zero using a gray color. Note that the Python script includes a specific function, `imwriteSymmetricImage`, to be able to save to disk images that have negative values. For this, you need to write code to convert and image with values in $[-b, b]$, to an image with values in $[0, 255]$ before using `cv2.imwrite`.

At each location (x, y), the gradient of the image is given by a 2-vector $\nabla f \doteq (f_x, f_y)$. Thus, two matrices/images are needed to describe the gradient for all pixels. To detect edges, we look for peaks in the gradient magnitude $\|\nabla f\| \doteq \sqrt{f_x^2 + f_y^2}$, also called the *edge strength* (see Fig. 7d). In numpy, this square root formula is implemented in the function `hypot` (hypotenuse). The information about the *edge orientation* is given by the angle of the gradient $\angle \nabla f \doteq \arctan(f_y/f_x)$, as shown in Fig. 7e.


(a) Frame $f(x, y, t)$


(b) Frame $f(x, y, t + \Delta t)$


(c) $\partial f/\partial x$


(d) $\partial f/\partial y$

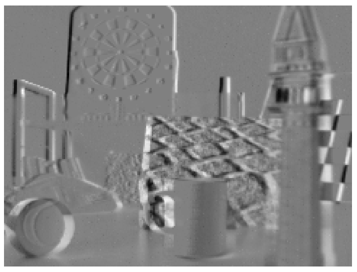

(e) $\partial f/\partial t$

Figure 8: Derivatives in x, y, t directions of a video signal $f(x, y, t)$ acquired by a DAVIS240C camera.

In numpy this can be computed with the standard function `arctan2`.

In OpenCV, the function `cv2.cartToPolar` converts the "Cartesian coordinates" $\nabla f = (f_x, f_y)$ into "polar coordinates" $(\|\nabla f\|, \angle \nabla f)$. Thus, it does two operations in one: `hypot` and `arctan2`. This is the code used in the script.

Temporal derivative calculation. Forward difference formula. So far, we have seen how to compute the spatial derivatives of an image. If we add one more dimension (time) then we could think of having a 3D volume of intensities (x, y, t variables) and compute the derivative along all three dimensions. The same kernels used for spatial derivatives could be applied to computer temporal derivatives if applied along the new dimension. In practice, it is common to use th forward difference formula, which just requires to temporal samples, that is, two images:

$$\frac{\partial f}{\partial t} \approx \frac{f(x, y, t + \Delta t) - f(x, y, t)}{\Delta t}.$$

Fig. 8 shows two consecutive images (also called "frames" in video) of a sequence recorded with a DAVIS240C sensor (240×180 pixels), from the Event Camera Dataset (Mueggler et al. IJRR 2017). The approximation to the temporal derivative consist of using the above formula. The scale factor $1/\Delta t$ is typically omitted since it does not affect the "structure" of the image. Take a look at the corresponding code in the Python script.

In this example, the camera is moving horizontally to the right, and so the temporal derivative captures the vertical edges ($\partial f/\partial x$). Note, however, that the approximation to $\partial f/\partial t$ differs from $\partial f/\partial x$: both (c) and (e) look similar in the darts and objects far away from the camera because the apparent motion on the image plane is small at such depths, but (c) and (e) look very different in the objects closer to the camera (tennis ball, mug, toy tower) because there is a large motion between the two consecutive frames. The larger the time interval and the smaller the depth of the object with respect to the camera, the larger the apparent motion in the image plane, and therefore the less accurate the above first-order approximation formula.