DUCKIETOWN

💻 04 - Image filtering

Images contain a tremendous amount of data. A 640×480 grayscale image contains 307,200 pixels, each of which can take on one of 256 different values. Reasoning over raw images as a means of estimating the structure of a robot's environment is intractable, particularly when we want to process images in real-time as they are streamed at 30 frames-per-second from a robot's camera.

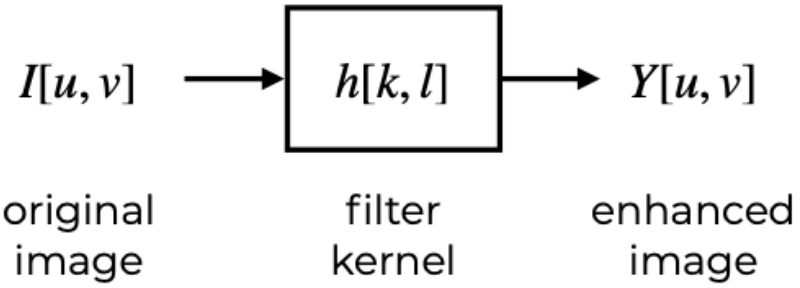


Image filtering performs local operations on an image using a kernel matrix as a means of enhancing its content.

Image filtering provides a means of enhancing an image by amplifying certain desireable aspects of the image while supressing others. The result is a transformation of the original raw image into a more concise collection of salient information.

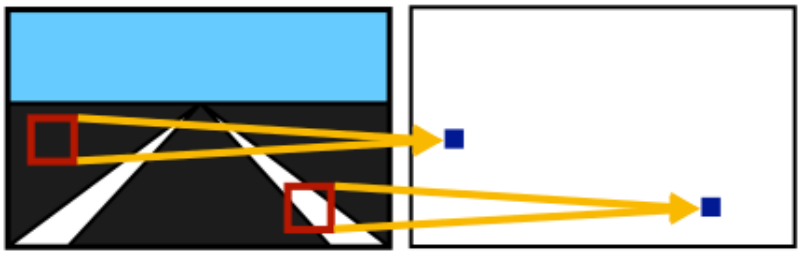


Image filtering performs local operations on an image using a kernel matrix as a means of enhancing its content.

In image filtering, each pixel in the output image is computed from a weighted combination of the neighboring pixels in the input image. This weighting is is referred to as the *filter kernel* and takes the form of a small (relative to the image) matrix h. Importantly, the kernel is the same everywhere in the image.

Mathematically, we can represent image filtering as the process of convolving the image with the kernel filter

$$Y = I * h$$

which, for each pixel u, v in the output image becomes

$$Y[u,v] = \sum_{k,l} h[k,l] I[u-k, v-l]$$

Notice that the $I[u-k, v-l]$ inside the sum has the effect of flipping the kernel h left-to-right and top-to-bottom.

The kernel determines which content is amplified and which is supressed, and so we design the kernel based upon how we want to enhance the image.

Example: Image blurring

In this example, we are given a noisy image and want to design a filter that removes as much of the noise as possible, while not losing too much of the actual detail. A candidate for such a filter is the *box filter*, a normalized matrix of ones.

$$h = \frac{1}{N} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$

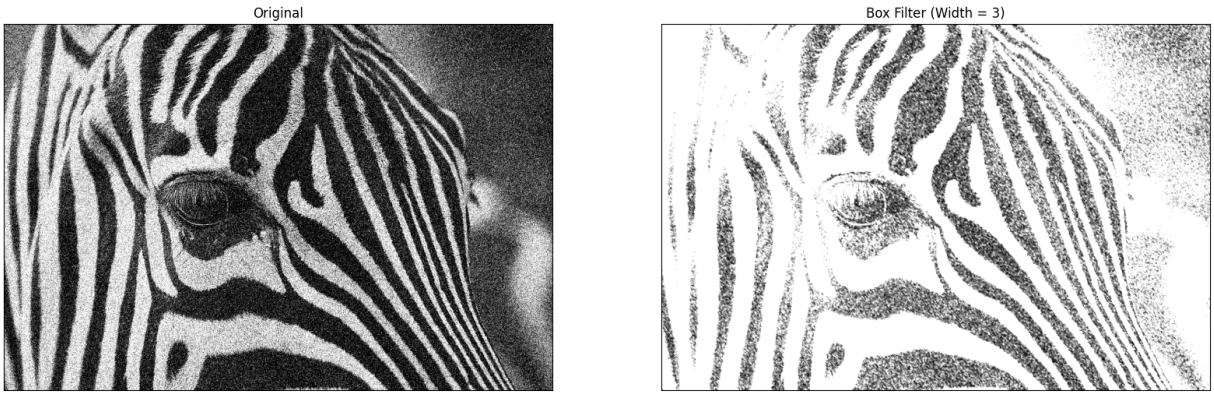
where N is the number of elements in the matrix. In designing a box filter, we have freedom to decide on the width and height of the kernel, though we will assume that the matrix is square. Increasing the size of the kernel means that a greater number of neighboring pixels will be used in determining the average intensity at each output location. Intuitively, this will improve noise reduction at the expense of attenuating actual detail in the image.

In [ ]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

In [ ]:
```
### Run this cell to import relevant modules
from matplotlib import pyplot as plt
import numpy as np
import cv2
```

In [ ]:
```
# TODO: Create a 3x3 box filter and compare the result of convolving the noi
#       Can you find the filter width above which too much detail is lost?
width = 3
hbox = np.random.rand(width,width) # CHANGE ME

# load the image
img = cv2.imread('../../assets/images/image_filtering/zebra-noisy.jpg', 0)
```

```
# filter the source image
img_box_filter = cv2.filter2D(img,-1,hbox)

# Visualize the filtered image alongside the original image.
fig = plt.figure(figsize = (20,20))
ax1 = fig.add_subplot(1,2,1)
ax1.imshow(img,cmap = 'gray')
ax1.set_title('Original'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,2,2)
ax2.imshow(img_box_filter,cmap = 'gray')
ax2.set_title('Box Filter (Width = ' + str(width) +')'), ax2.set_xticks([]),
```

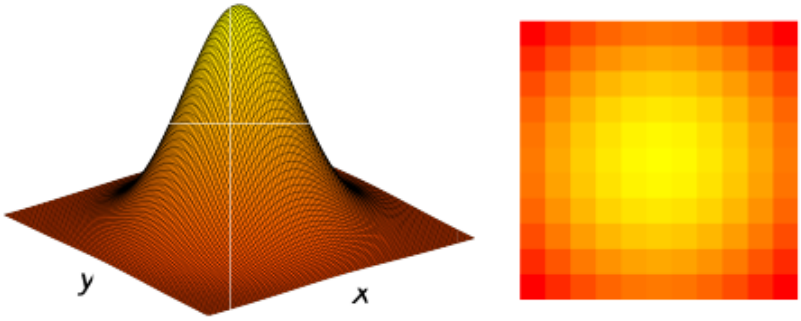


Gaussian Blurring

While the box filter is intuitive, using it to smooth an image prior to computing numerical derivatives can result in artefacts. In practice, people commonly use a kernel that approximates a zero-mean (i.e., centered at the origin) isotropic (i.e., cross-sections are circles; the distribution spreads out evenly in all directions) Gaussian

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where σ is the standard deviation. The advantage of using a Gaussian kernel over a box filter is that it places more weight on nearby neighbors than it does on those that are farther away.



A 2D Gaussian (left) and a Gaussian kernel (right).

The standard deviation σ determines how quickly the Gaussian decays. For large values of σ, the Gaussian will decay slowly and, in turn, neighboring pixels will be weighed

similarly. On the other hand, small values for σ result in a more peaked distribution, with the weights dropping off quickly.

While Gaussians have infinite support, we are interested in kernels that are finite (and typically small). We will typically choose the size of the kernel based on the support of the distribution (i.e., how spread out it is), which is determined by the standard deviation. When the standard deviation is small and the distribution is peaked, we can use a small kernel since weights outside the kernel will be very small. However, when the standard deviation is larger and the distribution is more spread out, we tend to use a larger kernel.

Example: Gaussian blurring

The Gaussian kernel is defined by its size and standard deviation σ. Since the standard deviation determines the rate at which the weights of the kernel decay, it is common to choose the size of the kernel (which will be square for isotropic Gaussians) based upon the standard deviation. Many existing libraries, including the implementation in OpenCV, support this functionality.

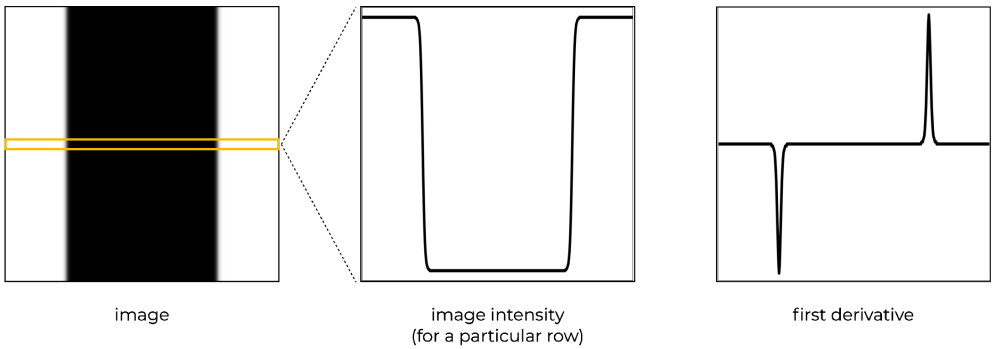
In this exercise, you will experiment with different choices of the standard deviation and compare them to the result of using the box filter that you identified above.

Try and find a setting for the standard deviation that reduces noise without removing too much content from the image.

In [ ]:
```
# TODO: Experiment with different settings for the standard deviation of the
#       Identify a setting for the standard deviation that removes noise whi
sigma = 2

# Smooth the image using a Gaussian kernel
img_gaussian_filter = cv2.GaussianBlur(img,(0,0), sigma)

# Visualize the filtered image alongside the original image.
fig = plt.figure(figsize = (20,20))
ax1 = fig.add_subplot(1,3,1)
ax1.imshow(img,cmap = 'gray')
ax1.set_title('Original'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,3,2)
ax2.imshow(img_gaussian_filter,cmap = 'gray')
ax2.set_title('Gaussian Filter (Sigma = ' + str(sigma) +')'), ax2.set_xticks
ax3 = fig.add_subplot(1,3,3)
ax3.imshow(img_box_filter,cmap = 'gray')
ax3.set_title('Box Filter'), ax3.set_xticks([]), ax3.set_yticks([]);
```

Enhancing Edges

Edges are a feature of an image that are used for a variety of tasks, including segmentation, object detection, shape estimation, and scene reconstruction. In the case of Duckietown and other self-driving domains, access to edge information is also very useful for detecting lane boundaries.



The change in intensity associated with an edge where an image transitions from white to black and black to white.

Edges correspond to significant changes in local intensity. This suggests that we can identify edges by looking for regions in the image where the intensity gradient ∇I is large. Consider the left-most image above, which might be a zoomed in view of one of the zebra's vertical stripes. Moving left-to-right along a single row of the image, we see that the intensity rapidly drops as the pixels transition from being white (i.e., close to 255) to black (i.e., closs to 0), and then stay small until they jump back up as the pixels go from black to white. The edges are clearly visible as spikes in the x-component of the intensity gradient.

In general, we are interested in the two-dimensional image gradient, which includes both horizontal as well as vertical changes in intensity:

$$\nabla I[u,v] = \begin{bmatrix} \frac{\partial I[u,v]}{\partial x} \\ \frac{\partial I[u,v]}{\partial y} \end{bmatrix}$$

The magnitude of the gradient $|\nabla I[u,v]|$ provides a measure of the strength of the edge, while it points in the direction of greatest change

$$|\nabla I[u,v]| = \sqrt{\left(\frac{\partial I[u,v]}{\partial x}\right)^2 + \left(\frac{\partial I[u,v]}{\partial y}\right)^2} \qquad \theta[u,v] = \text{atan2}\left(\frac{\partial I[u,v]}{\partial y}, \frac{\partial I[u,v]}{\partial x}\right)$$

Since pixels are discrete, we can't compute the derivatives analytically. Instead, we estimate them numerically via a finite difference approximation. We can compute these estimates by convolving the image with appropriately chosen kernels h_x and h_y that compute finite derivatives in the x and y directions, respectively:

$$\nabla I[u,v] = \begin{bmatrix} G_x[u,v] \\ G_y[u,v] \end{bmatrix}$$

where

$$G_x = h_x * I \tag{1}$$
$$G_y = h_y * I \tag{2}$$

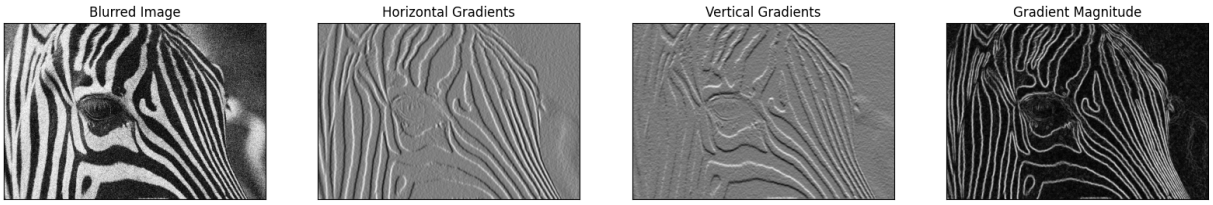
Example: Image gradients

In this example, you will design different filter kernels, h_x and h_y, to estimate the horizontal and vertical gradients of an image. In doing so, remember that the filter will be flipped left-to-right and top-to-bottom when performing convolution.

In [ ]:
```
# TODO: Design two kernels hx and hy to estimate the horizontal and vertical
#       These filters will be applied to the blurred image that you generate
#       kernels, compare the output for different amounts of blurring (i.e.,
#       compare the gradients and their magnitude with little (i.e., sigma=1

hx = np.array([[-1, 0, 1]], dtype=np.float32) # CHANGE ME, BUT KEEP dtype=np
hy = np.array([[-1],[0],[1]], dtype=np.float32) # CHANGE ME, BUT KEEP dtype=

print(hx)

# Apply the filters to the blurred image
Gx = cv2.filter2D(img_gaussian_filter.astype(np.float32),-1,hx)
Gy = cv2.filter2D(img_gaussian_filter.astype(np.float32),-1,hy)


# Compute the magnitude of the gradients
Gmag = np.sqrt(Gx*Gx + Gy*Gy)

# Visualize the filtered image alongside the original image.
fig = plt.figure(figsize = (20,20))
ax1 = fig.add_subplot(1,4,1)
ax1.imshow(img,cmap = 'gray')
ax1.set_title('Blurred Image'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,4,2)
ax2.imshow(Gx,cmap = 'gray')
ax2.set_title('Horizontal Gradients'), ax2.set_xticks([]), ax2.set_yticks([]
ax3 = fig.add_subplot(1,4,3)
ax3.imshow(Gy,cmap = 'gray')
```

```
ax3.set_title('Vertical Gradients'), ax3.set_xticks([]), ax3.set_yticks([]);
ax4 = fig.add_subplot(1,4,4)
ax4.imshow(np.uint8(Gmag),cmap = 'gray')
ax4.set_title('Gradient Magnitude'), ax4.set_xticks([]), ax4.set_yticks([]);
```

```
[[-1.  0.  1.]]
```



Sobel Operator

A popular set of kernels used the emphasize edges in an image is the *Sobel operator*, which consists of two 3×3 kernels:

$$h_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad h_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

These kernels correspond to simple averaging-based blurring followed by a simple finite derivative:

$$h_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} +1 & 0 & -1 \end{bmatrix} \qquad h_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix}$$

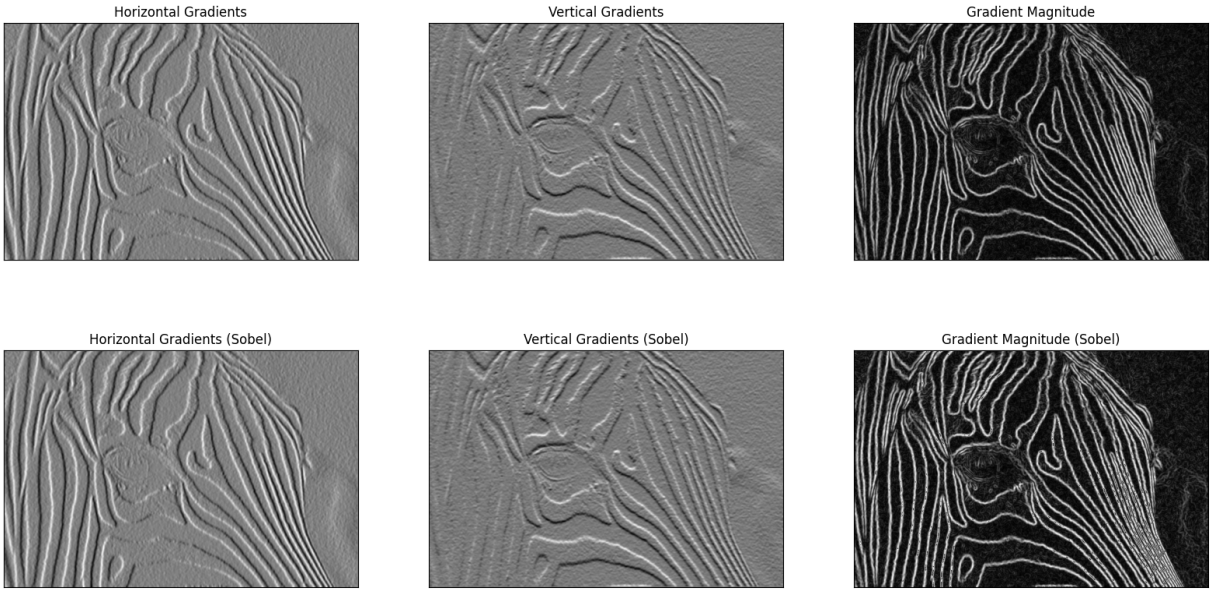
Example: Sobel Operator

In this example, we will compare the horizontal and vertical gradients identified with the filter that we designed with those identified using the Sobel operator. In doing so, explore how the filter that you designed above and the Sobel operator respond differently to different amounts of smoothing.

Note: To be fair, the Sobel operator has some smoothing by default, which the filter that you designed above may not.

In [ ]:
```
sobelx = cv2.Sobel(img_gaussian_filter,cv2.CV_64F,1,0)
sobely = cv2.Sobel(img_gaussian_filter,cv2.CV_64F,0,1)

# Compute the magnitude of the Sobel-based gradients
sobelmag = np.sqrt(sobelx*sobelx + sobely*sobely)

fig = plt.figure(figsize = (20,10))
ax1 = fig.add_subplot(2,3,1)
ax1.imshow(Gx,cmap = 'gray')
ax1.set_title('Horizontal Gradients'), ax1.set_xticks([]), ax1.set_yticks([]
ax2 = fig.add_subplot(2,3,2)
```

```
ax2.imshow(Gy,cmap = 'gray')
ax2.set_title('Vertical Gradients'), ax2.set_xticks([]), ax2.set_yticks([]);
ax3 = fig.add_subplot(2,3,3)
ax3.imshow(np.uint8(Gmag),cmap = 'gray')
ax3.set_title('Gradient Magnitude'), ax3.set_xticks([]), ax3.set_yticks([]);
ax4 = fig.add_subplot(2,3,4)
ax4.imshow(sobelx,cmap = 'gray')
ax4.set_title('Horizontal Gradients (Sobel)'), ax4.set_xticks([]), ax4.set_y
ax5 = fig.add_subplot(2,3,5)
ax5.imshow(sobely,cmap = 'gray')
ax5.set_title('Vertical Gradients (Sobel)'), ax5.set_xticks([]), ax5.set_yti
ax6 = fig.add_subplot(2,3,6)
ax6.imshow(np.uint8(sobelmag),cmap = 'gray')
ax6.set_title('Gradient Magnitude (Sobel)'), ax6.set_xticks([]), ax6.set_yti
```



Wrap-Up

In this exercise, we have explored a few filters that can be used to enhance an image, whether by removing noise or other unwanted detail or by emphasizing the presence of specific content, such as edges. These and other filters are an integral step to extracting information from images, whether it is to identify the location of the lane markings and, in turn, the geometry of the robot's lane, or detecting the location of pedestrians (i.e., duckies). These are just a few examples of filters that are commonly used for image processing. Operating on the same principle as filters designed for edge detection, corner detection filters look for locations in the image where there are rapid changes in intensity in both the x- and y-directions (e.g., like you might see at corners where two lines intersect). These and related filters are often employed to detect "interest points", salient points in the image that are used to identify correspondences between multiple images of the same scene (i.e., pixels that correspond to the same object in the world).

You can now move to the visual servoing tutorial.