TTK4255: Robotic Vision

Homework 1: Image processing

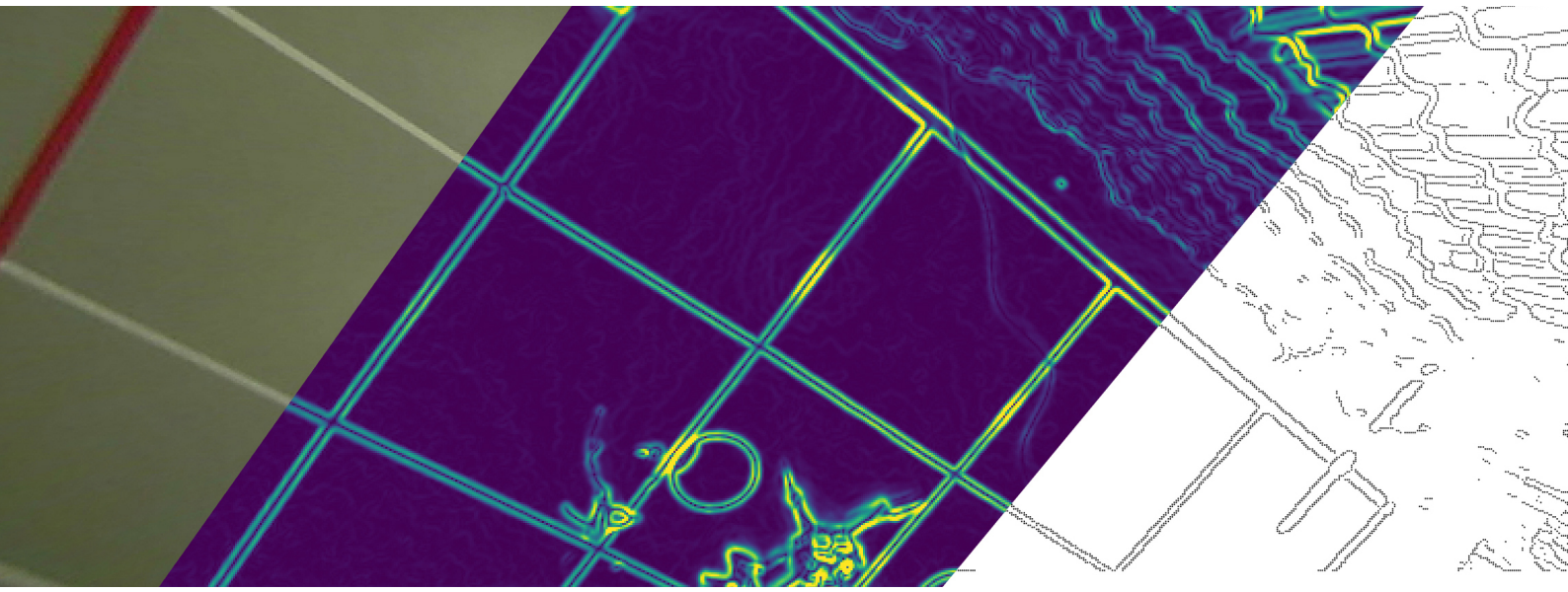

Figure 1: Intermediate stages of a simple edge detection algorithm you will implement.

Instructions

For more information about the course work and how to get help, see the `Course work/about.pdf` document on BB. To get your assignment approved, you need to complete 60% (sum of task weights). Upload the requested answers and figures as a single PDF. You don't need to submit your code. You may collaborate with other students and submit the same report, but you still need to upload it individually on Blackboard. Please write your collaborators' names on your report.

About the assignment

This assignment is intended to familiarize you with Python/Matlab, while teaching you some common image processing techniques and concepts. Leading into the next assignment (detecting lines), here you will implement a simple algorithm to detect "edges", which are locations in the image where the color abruptly changes.

Relevant reading

The image processing theory in this assignment is presented in Szeliski (2010): 3.1 (Point operators) and 3.2 (Linear filtering). Edge detection is discussed in great detail in Szeliski 4.2 (Edges), but it is not necessary to read the chapter to complete the tasks. In Part 2 we mention color spaces. This topic is presented in Szeliski 2.3.2 (Color), but is not required reading.

Getting started

The assignments are a mix of theory and programming problems. We recommend that you use Python or Matlab. We provide reference solutions, and often starter code, for these languages. The back of this document also has some tips, useful functions and common mistakes for Python and Matlab.

Python: While the assignments don't require a specific version, our reference solutions are written for Python 3.7. If you want to be able to run these without significant modifications, use a version from 3.0 or above. We recommend that you install Numpy (link) and Matplotlib (link), for linear algebra and plotting. You may need to install PIL (link) to read some image formats. The Python Numpy Tutorial (link) by Justin Johnson is a good reference for various functionality we'll use.

Matlab: You can use any version of Matlab. However, our reference solutions make use of local function definitions inside script files—a feature added in 2016b. If you have an earlier version, you may want to upgrade. The assignments will require some functions in the Image Processing Toolbox (link). If you don't have it, you can install new toolboxes to an existing Matlab installation by rerunning the installer (link). While you're at it, you may want to install the Computer Vision Toolbox (link), which will be useful for the projects later in the course. The first eight pages of the Matlab Primer (link) from ETHZ is a good reference for Matlab functionality we'll use.

Part 1 Theory questions

Task 1.1: (4%) Which of the following image transformations are *point operators*?

(a) Convolution

(b) Median filtering

(c) Brightness and contrast adjustment

(d) Rotating the image 90 degrees

Task 1.2: (8%) What is the effect of convolving an image with this kernel?

$$\begin{bmatrix} -0.5 & -1.0 & -0.5 \\ -1.0 & +7.0 & -1.0 \\ -0.5 & -1.0 & -0.5 \end{bmatrix}$$

Tip: Work out the result of the convolution along a straight line in an image with (a) uniform pixel values and (b) a vertical or horizontal step edge (e.g. a transition from 0 to 1).

Task 1.3: (8%) Several filter kernels in signal processing are based on continuous functions due to the ease of analyzing theoretical properties in the continuous domain. For example, a commonly-used function is the one-dimensional Gaussian,

$$g(x) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot \exp\left(\frac{-x^2}{2\sigma^2}\right), \tag{1}$$

where σ is called the standard deviation. A corresponding discrete filter kernel can be created by evaluating the continuous function within a finite-sized window, e.g. at integer locations:

$$[g(-h), \cdots, g(-1), g(0), g(1), \cdots, g(h)]. \tag{2}$$

Because the Gaussian has infinite support, a finite-sized window will cause a truncation error. Thus, when creating a Gaussian filter kernel, you need to choose both the standard deviation (reflecting the desired filtering properties) and the kernel size (constrained by your computational budget).

(a) For a given error threshold ϵ and standard deviation σ, derive an expression for the half-width h such that the value of g outside the window is less than ϵ.

(b) Suppose the desired threshold is $\epsilon = 1/256$ and $\sigma = 3$, how big should the kernel be?

Part 2 Basics

This part will familiarize you with loading, plotting and basic image operations. There is no provided starter code, so you will have to write the code to produce the requested outputs yourself.

Task 2.1: (4%) An image named `grass.jpg` is in the provided data directory. The image is from a dataset of a sugar beet field (link). Load the image and print its width and height.

Task 2.2: (4%) The image has three channels: red, green and blue. A single-channel image can be extracted (in Python) using `img[:,:,i]`, $i \in 0, 1, 2$ or (in Matlab) using `img(:,:,i)`, $i \in 1, 2, 3$. Can you find out which i corresponds to the green channel by plotting the three single-channel images?

Task 2.3: (4%) Thresholding is a simple technique for isolating parts of interest in an image (a problem known as segmentation). In Python and Matlab, thresholding is a one-liner operation using the syntax `img > threshold`. The result is a binary image of 1's and 0's, where a value of 1 is assigned to each element that satisfies the expression. Using only the green channel, can you find a threshold that isolates all the pixels belonging to the sugar beet leaves? Tip: Zoom in to ensure you are not simply segmenting their shadows.

Task 2.4: (4%) The issue you will encounter above is that the G component of RGB confuses the hue and luminance of the color and cannot be used in isolation to determine the "green-ness" of something. For example, pure white has a large G component, but doesn't look green. Usually we would represent the image in a different color space which disentangles the aspects of interest of the perceived color. A simple solution is to use normalized rgb coordinates:

$$r = \frac{R}{R+G+B}, \quad g = \frac{G}{R+G+B}, \quad b = \frac{B}{R+G+B}, \tag{3}$$

which sum up to 1. Normalized rgb coordinates are invariant to a scalar multiplicative factor, which can be somewhat useful to make an algorithm sensitive only to the hue of a color. Convert the RGB image to normalized rgb and include a figure showing the three single-channel images (r, g, b).

Task 2.5: (4%) Repeat the thresholding task using the normalized rgb image. Include a figure showing your segmentation result (i.e. the binary image). Are you able to achieve a result similar to that below? (You don't have to reproduce the rightmost figure.)

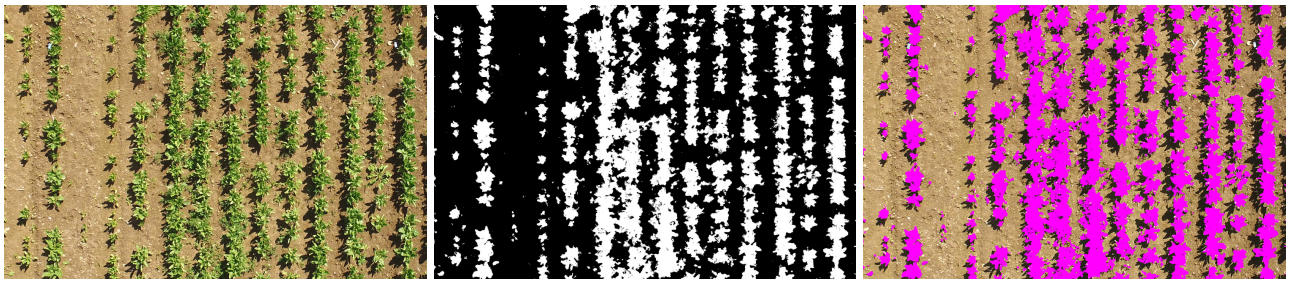

Figure 2: Segmentation obtained by thresholding normalized rgb. Left: Input. Middle: Binary image. Right: Above-threshold pixels replaced by magenta.

Part 3 Edge detection

In this part you will use filtering and thresholding to implement a simple edge detection algorithm. In the tasks below, you should finish the implementation of stub functions declared in the provided template code. After each task, you can run the `task3` script to generate a figure visualizing each output. You only need to include this figure once, after completing the last task. If you do not complete all the tasks, you can run the script anyway to get a figure showing your partial results.

Task 3.1: (4%) Color images have three channels, but it can sometimes be more practical to work with "grayscale" images with a single channel. There is no unique color-to-grayscale conversion. Here, we just want something that preserves gradients between objects and textures of interest. A fast and simple conversion is the average of the three channels. Implement the function `rgb_to_gray` that converts a color image to a grayscale image by averaging.

Task 3.2: (14%) To detect edges, we will look for pixels that have an abrupt change in pixel value. This can be measured by using the image gradient. Implement the function `central_difference` that takes a grayscale image I and computes the horizontal and vertical gradient by convolving with the 1-D central difference kernel

$$\begin{bmatrix} +\frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \tag{4}$$

Return the two gradient images and the gradient magnitude. To compute the horizontal gradient image I_x, convolve the kernel with each row of the input. For the vertical gradient image I_y, convolve with each column. You can use `conv` (Matlab) or `numpy.convolve` (Python) for 1-D convolution. The gradient magnitude can be computed as $I_m = \sqrt{I_x^2 + I_y^2}$.

Ensure that the outputs have the same size as the input. To do this you need to handle border effects (Szeliski 3.2). One solution is to pad the image row or column with zeros, which you can do easily by specifying the `'same'` option in `conv` and `numpy.convolve`. However, zero-padding may produce darkening near the sides. You can alternatively pad by replicating the end element on either side.

Task 3.3: (14%) Implement `gaussian` which convolves a grayscale image with the 2-D Gaussian

$$\frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{5}$$

taking the standard deviation parameter σ (sigma) as the blur strength. The 2-D Gaussian is separable, which means that the 2-D convolution can be computed by convolving the 1-D Gaussian with each row of I, giving an intermediate image I', followed by convolving with each column of I'. Again, ensure that the result image has the same size as the input using one of the two padding strategies above.

Task 3.4: (14%) Implement the function `extract_edges` that identifies edges by thresholding the gradient magnitude. The function should return a list of pixel coordinates (x, y) and orientations θ. The orientation should be computed as the angle of the image gradient vector: $\theta = \text{atan2}(I_y(y, x), I_x(y, x))$.

Tip: Use the functions `np.nonzero` (Python) or `find` (Matlab) to extract the coordinates of above-threshold pixels. In Matlab, `sub2ind` may be useful to obtain the I_x, I_y values for computing the angle.

Task 3.5: (14%) Test your functions on the `grid.jpg` image and include the figure showing the blurred image, gradient images and the extracted edges. Try adjusting the blurring parameter σ and the edge threshold. What effect does blurring have on the detected edges?

Task 3.6: (Optional self-study task - 0%) Convolution commutes with differentiation. Instead of convolving with the Gaussian and the central difference kernel separately, modify your program to directly convolve with the partial derivatives of the 2-D Gaussian. Note that partial derivatives of the 2-D Gaussian are also separable: to compute the horizontal gradient image, convolve each row of the input with the horizontal derivative of the 1-D Gaussian, followed by convolving each column of the result with the vertical Gaussian (not its derivative). Vice versa for the vertical gradient image.

Task 3.7: (Optional self-study task - 0%) Looking closely, you may observe that the extracted edges are "thick". A slightly more complicated algorithm that produces thin single-pixel edges is to filter with the Laplacian of Gaussian (LoG) and extract zero crossings of the result, as described in Szeliski 4.2.1. Modify your program to use this algorithm as well and compare the results.

Note: Tasks with the label "Optional self-study task" do not count toward the approval of your assignment. They may still be relevant to the exam.

Python tips (see next page for Matlab)

`import matplotlib.pyplot as plt`	Import a Matlab-like interface to Matplotlib.
`import numpy as np`	Common way to import Numpy.
`from numpy import *`	Make all functions accessible in global namespace.
`help(<name>)`	Print information about `<name>`.
`np.array([x,y,z])`	Create a vector.
`np.array([[x,y,z]])`	Create a row vector.
`np.array([[x],[y],[z]])`	Create a column vector.
`np.array([[a,b],[c,d]])`	Create a 2×2 matrix.
`np.linalg.norm(p)`	Calculate 2-norm of vector.
`h,w = img.shape[0:2]`	Retrieve the height and width of an image.
`plt.imread`	Load an image.
`plt.imshow`	Draw an image on the current plot.
`plt.imsave`	Save an image to disk.
`plt.show`	Make your plots actually show up on the screen.
`plt.savefig('figure.png')`	Save displayed figure to disk. (Call before `plt.show`)
`plt.plot([x0,x1], [y0,y1])`	Draw a single line from (x_0, y_0) to (x_1, y_1).
`plt.scatter(x, y)`	Draw a marker at (x, y).
`plt.scatter(p[:,0], p[:,1])`	Draw a set of points stored as an $n \times 2$ array.

Unlike Matlab, the `*` operator in Numpy is element-wise multiplication, not matrix multiplication. The `dot` function or `@` operator can be used instead to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices: e.g. `A.dot(b)` and `A @ b` both perform standard matrix multiplication between A and b. (The `np.matrix` class does define `*` to be matrix multiplication, but it is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future).

`plt.imread` may store the loaded image using 8-bit unsigned integers (values 0 to 255 in each channel). Using this format later can lead to unexpected behavior or mysterious error messages due to overflow or type mismatch. You can convert an 8-bit image to floating-point by dividing it by 255. This will scale the RGB components to be in the range [0, 1], so keep that in mind when e.g. choosing thresholds.

It's very easy to accidentally swap x and y coordinates. Remember, images expressed as NumPy arrays are accessed like a matrix: e.g. `image[row, column]` or `image[y, x]`

Matlab tips (see previous page for Python)

`help <function>`	Print information about a function.
`lookfor <keyword>`	Get a list of functions related to a keyword.
`doc <function>`	Documentation for a function.
`doc images`	Documentation for the Image Processing Toolbox.
`imread`	Load an image.
`imshow`	Draw an image on the current plot.
`imwrite`	Save an image to disk.
`print('myfigure', '-dpng')`	Save displayed figure to disk.
`plot(x,y)`	Draw lines.
`scatter(x,y)`	Draw points.
`im2double`	Convert image in 8-bit unsigned format (values 0 to 255 in each channel) to floating-point (values 0-1).
`norm(v)`	Calculate the 2-norm of vector v.
`vecnorm(A,p,dim)`	If A is a matrix, calculate the p-norm of each vector along dimension dim.

`imread` may store the loaded image using 8-bit unsigned integers (values 0 to 255 in each channel). Using this format later can lead to unexpected behavior or mysterious error messages due to overflow or type mismatch. It can be useful to convert the image to floating-point using `im2double`. This will scale the RGB components to be in the range [0, 1], so keep that in mind when e.g. choosing thresholds.

It's very easy to accidentally swap x and y coordinates. Remember, images expressed as Matlab arrays are accessed like a matrix: e.g. `image(row, column)` or `image(y, x)`.