Writing Computer Vision Programs

For this computer vision assignment, I decided to keep things simple, efficient, and convenient. Since this is not a software engineering class, we will not develop complete, user-friendly applications but just implement some computer vision algorithms and play around with them. And we will do this in the plain, old C programming language. There are several reasons for this: (1) C compilers are freely available for any platform and operating system, and the tiny image reading/writing library I wrote for you should be compatible with all of them; (2) you most likely know C already or can quickly learn it because of its similarity to Java; (3) the compiled code will execute extremely fast, which will be very helpful for some transformations and machine learning approaches; and (4) programming in a low-level language without specific libraries will force you to think about every elementary step in the algorithms and really understand how they work.

Regarding the digital images, we are going to work with the simplest file format, which is called Netpbm. These are the files with extensions “.PBM” (black-and-white, 1 bit per pixel), “.PGM” (grayscale, 8 bits per pixel), and “.PPM” (RGB color, 24 bits per pixel). You can find out all about them here:

http://en.wikipedia.org/wiki/Netpbm_format

I wrote a very small C library for reading, writing, creating, and deleting images in these formats, which should be easy to understand and use. You can download its “netpbm.c” and “netpbm.h” files, plus a sample program named “netpbm_test.c” and a sample image “sample.ppm,” from the course homepage, next to the April 14 slides. You should put all files into the same directory and include the C files into a project for your compiler. When you compile and run the sample program, it should load the sample image, produce several variants of it, and write them to the same directory. This program should be very easy to understand and will show you how to implement your own image processing and computer vision algorithms. Notice that this library only works with the binary versions of the Netpbm formats. There are also ASCII versions of these formats, which are horribly inefficient and should be avoided. While the Netpbm format is not as common as, for instance, the JPEG or PNG formats, there is free software available for all common operating systems that can read, write, and convert Netpbm files. I recommend
the cross-platform image editor GIMP (http://www.gimp.org), and for Windows also the image viewer IrfanView (http://www.irfanview.com/). For our purpose, the advantage of the Netpbm format is that it is easy to understand and only contains the information we need. If you have any questions about the code or the format, please do not hesitate to ask me by e-mail or in person.

Question 1: Implement the Sobel Edge Detectors and Gaussian Filters

In a first step, write a function that performs a convolution on a given image and filter. Both inputs and the output are of the Matrix type, which means that you first have to convert the input image into a matrix. This seems cumbersome but is more efficient in the long run, because sometimes we want to perform multiple successive convolutions on an image, and we would also like to be able to use real-valued convolution inputs. The function should thus have the following signature:

Matrix convolve(Matrix m1, Matrix m2)

As you know, m1 and m2 are in principle exchangeable, but only if they are padded with zeroes outside of their borders so that all overlaps of elements from m1 and m2 are considered. For our computer vision applications, however, we expect the filter to be smaller than the input image, and during convolution we never want any elements of the filter to be outside of the image. Therefore, we require that m1 contains the values of the input image, and m2 those of the filter. The resulting matrix is always of the same size as the input image, with zeroes in those positions that could not be reached by the center element of the filter. For example, if the input image has 7 by 7 entries and the filter has 3 by 3 entries, then the output is a 7 by 7 matrix with zeroes in its first and last rows and its first and last columns. The function assumes that we use the center of the filter as the indicator of where to store the output value. If the height or width of the filter is an even number of elements, the center value is rounded down. For example, if m2 is a 2 by 2 filter, we use its upper-left position as its “anchor” for storing output values.

Having implemented the convolve function, it is now easy to write a sobel function that takes an input image (only considering the intensity, i.e., grayscale component) and outputs a grayscale image of the same size, indicating the edge strength at each position in the input image. The weakest edge pixel in the image has intensity 0 (black), and the strongest one intensity 255 (white), which is accomplished by linearly scaling the Sobel filter edge strength output. Use the following signature:

Image sobel(Image img)

Now implement a gauss function that applies an $n \times n$ Gaussian filter to a given grayscale image. It has the following signature:

Image gauss(Image img, int size)
Here, size means the length and width of the filter, e.g., size 5 indicates a 5×5 filter. Only odd sizes are allowed so that the filter always has a well-defined center. Note that you can either build an \( n \times n \) filter or just create an \( n \times 1 \) Gaussian filter, apply it to the image, and then apply the transpose of the filter (1×\( n \)) to the result. It turns out that this has the same effect.

Write a program file filtering.c that reads the sample.ppm file from the same directory, applies the Sobel detector, a 3×3 Gaussian filter, and a 9×9 Gaussian filter to it and writes the result to grayscale files named “sobel.pgm”, “gauss3.pgm”, and “gauss9.pgm”.

**Question 2: A Maximum Filter?**

We have seen that a median filter is good at removing salt-and-pepper noise. What effect would a maximum filter have, i.e., a filter whose output, instead of the median, is the maximum intensity within its \( n \times n \) pixel area? What kind of noise could this filter remove best? What would be its side effect?

**Question 3: FAST and BRIEF**

(a) What is the effect of increasing the threshold in the FAST algorithm?

(b) What would happen if instead of 12-pixel runs we would only require 8-pixel runs for detecting a corner with FAST? Would we still detect corners?

(c) In the BRIEF method, we compare intensity at point pairs \((a, b)\), and if it is greater for \(a\) than for \(b\), we add a “1” to our descriptor and otherwise a “0”. What would be the advantages and disadvantages of using a vector of real numbers as the descriptor instead, each indicating the intensity difference between \(a\) and \(b\) for one of the pairs \((a, b)\)?

(d) We found that the original BRIEF method is slightly rotation invariant, i.e., if it sees the same keypoint at a slightly different orientation, it will still recognize it. What is the reason for it? It always looks at exactly the same points relative to the keypoint, so if the local information is rotated, it will compare completely different pixels, so should it not fail for even the slightest rotation?

**Question 4 (required for CS670, bonus for CS470): A Hough Transform Variant**

For some mysterious application, you need to build an application that detects horizontal lines in a picture. These lines are precisely horizontal, but your client does not only want to know in which rows of the image they are located but also where they start and end, i.e., what their leftmost and rightmost pixels are.
How would you set up a Hough transform to accomplish this task? What would be the dimensions of the output space, and what would be the equation for placing votes?