Computational Photography HW1

contents:

- introduction
- filtering
    - part0 - convolution
    - part1 - gaussian filter
    - part2 - sharpen filter
    - part3 - median filter
- drawing with gradients
    - image gradients and the sobel filter
    - part4 - gradient orientations and magnitudes
    - mapping to color
    - conclusion
- references

1. introduction

In this homework assignment, we will be playing around with images, filters, and convolution in order to cement the concepts you learned in lecture.

We will begin by building a function that performs convolution. We will then experiment with constructing and applying a variety of filters. Finally, we will see one example of how these ideas can be used to create interesting effects in your photos, by finding and coloring edges in our images.

We expect that you already have python and required packages installed. If not, consult the ‘software installation’ link on the left hand side of the coursera page. This assignment will get into some more programming, but we still tried to keep things as accessible as possible. We don’t expect it to take you more than 5 hours.

2. filtering

The code for this part of the assignment is split up as follows:

part[0-3].py contain the functions you will be implementing. Running these files will also perform

unit tests on your code.

run.py will run unit tests, and if successful, apply your code to the images placed in the images/source subdirectory. Feel free to experiment with the code at the bottom of this file to apply different filters to your images.

submit.py will evaluate your code and give you credit on the coursera site.

part0 - convolution

This part of the assignment lives in the part0.py file. You are tasked with implementing the convolve function. The lecture and tutorial videos, and the documentation string in the file will give you more detail about what it should do.

Make sure you map the kernel with the right indices (recall the difference between correlation and convolution). You may find element-wise multiplication (the * operator for numpy arrays) and the numpy.sum function useful while implementing this part.

Once you are able to pass the unit test in part0.py, run.py will apply a box filter to the images in the images/source directory. The outputs will be saved to the images/filtered directory.

Even if you can't get this part of the assignment to work, the code in other parts of the assignment can use a built-in convolution function, so you can still move ahead and experiment with some filters before coming back to this part.

part1 - gaussian filter

In this part of the assignment, you will be implementing the make_gaussian function in the part1.py file. The documentation string in the file will give you more detail about what it needs to do.

You will be given a neighborhood size k, and a standard deviation std. The returned array should have shape (2*k+1, 2*k+1) and dtype float.

You can create a 1d gaussian array with this command:

```
from scipy import signal
gaussian1d = signal.gaussian(size, std)
```

Which is further discussed in the programming tutorials.

Each cell [i,j] of your 2d filter should be the product of the i'th and j'th elements of this 1d array. Once all of the elements are filled, the array has to be normalized so that the total sum of the

elements is 1.

part2 - sharpen filter

Next, you will get some practice combining filters by creating a sharpen filter in part2.py. As you saw in lecture, a sharpen filter is constructed by putting a 2 at the center of a kernel, and then subtracting a gaussian kernel from that.

Feel free to make use of the gaussian function you wrote in part1, and make sure to check out the results by using run.py!

part3 - median filter

One last thing we would like you to get a feel for is nonlinear filtering. So far, we have been doing everything by multiplying the input image pixels by various coefficients and summing the results together. A median filter works in a very different way, by choosing a single value from the surrounding patch in the image.

You will need to implement the filter_median function in part3.py, which works a lot like the convolve function from part0. The documentation string for the function will give you more detail about what it's supposed to do. You may find the function numpy.median useful when working on this part of the assignment.

As with previous parts, run.py will let you apply this function to the test (and your own) images.

3. drawing with gradients

At this point, you should be familiar with how to construct kernels and convolve them with your images, as well as have a handle of what nonlinear filters look like. Congrats!

One of our main objectives in this class is to give you the tools you need to start blending photography and computation together in order to achieve interesting artistic results. We created the second part of this assignment with this in mind. For some parts of the assignment, I will simply describe what the code is doing. In other parts, you will be asked to fill in the missing functionality.

A facet of this assignment is that you will get experience reading and modifying other people's code, as well as exploring documentation and experimenting with objects and functions of new libraries. These skills lie at the core of becoming a capable programmer.

At a high level, we will be finding the intensity and orientation of the image gradient, and then

visualizing this information in an aesthetically pleasing way. In the remainder of this document, I will walk you through the process by which we can transition from the image on the left to the image on the right.

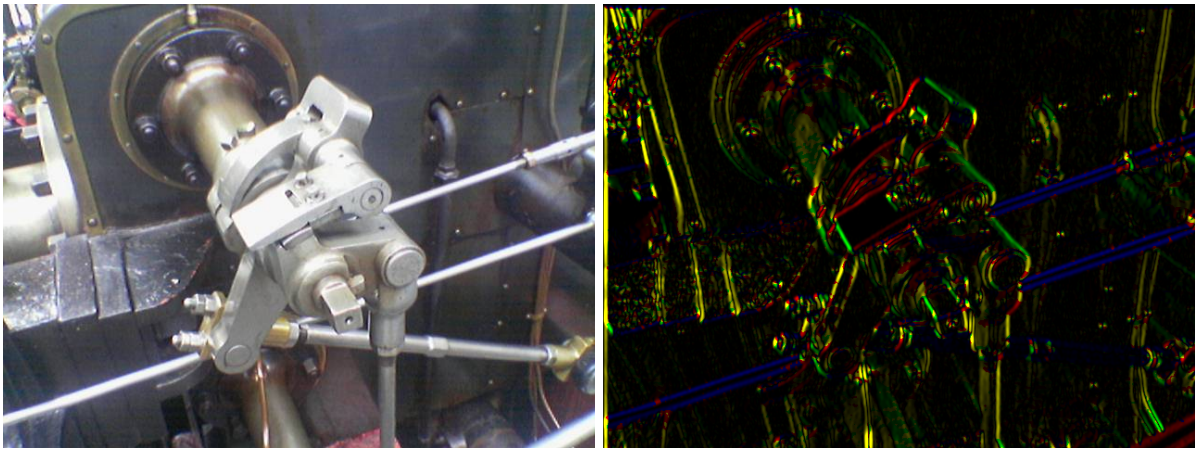


image source: http://en.wikipedia.org/wiki/File:Valve_original_(1).PNG

For more detail on this process, and some ideas for how you may extend it, consult the following page: http://en.wikipedia.org/wiki/Canny_edge_detector

The code for this part of the assignment is split up as follows:

part4.py contains the implementation, including the two functions that are given for you to complete. Running this file will test your code, and if successful, apply the pipeline to the images in the images/source directory, saving to the images/filtered directory.
submit.py will evaluate your code and give you credit on the coursera site.

image gradients and the sobel filter

At the bottom of part4.py you will find code that searches the images/source folder and applies run_edges function to each one. This run_edges function orchestrates the transformation for each image. The first thing it does is find the gradient of the image.

A gradient is a fancy way of saying “rate of change”. There are a variety of ways for finding it, and one standard is through the use of Sobel filters, which have the following kernels:

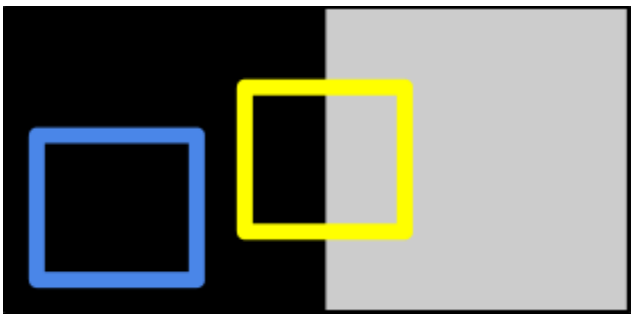
in the x direction:

-1	0	1
-2	0	2
-1	0	1

and in the y direction:

-1	-2	-1
0	0	0
1	2	1

If you think about the correlation between the x sobel filter and a strong vertical edge:



Considering the yellow location, the values on the right side of the kernel get mapped to brighter, and thus larger, values. The values on the left side of the kernel get mapped to darker pixels which are close to zero. The response in this position will be large and positive.

Compare this with the correlation of the kernel to a relatively flat area, like the blue location. The values on both sides are about equal, so we end up with a response that is close to zero.

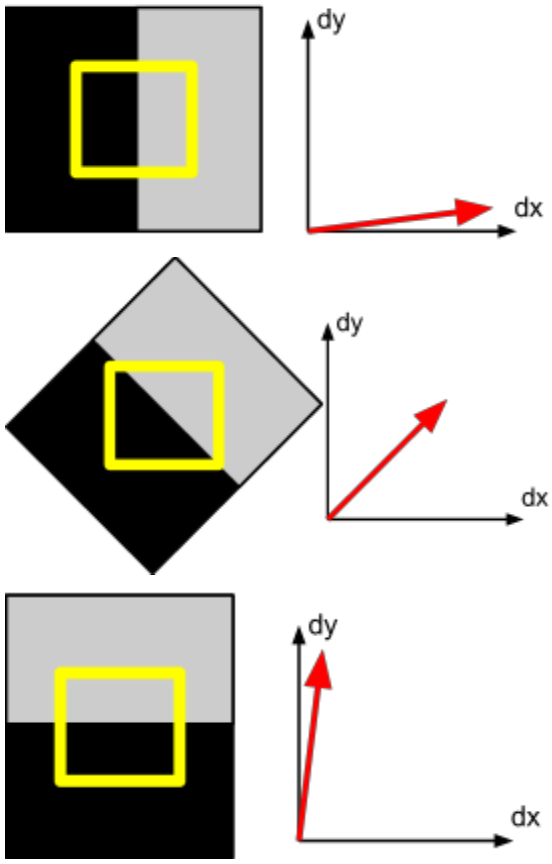
Thus, the x-direction sobel filter gives a strong response for vertical edges. Similarly, the y-direction sobel filter gives a strong response for horizontal edges. Both filters give responses close to zero in flat areas.

The first few lines of run_edges convert the image to grayscale, and then apply the two sobel filters to it. You may notice that we also blur the image with a gaussian kernel prior to applying this filter. The purpose of this is to remove noise from the image, so that we find responses only to significant edges, instead of small local changes that might be caused by our sensor or other factors.

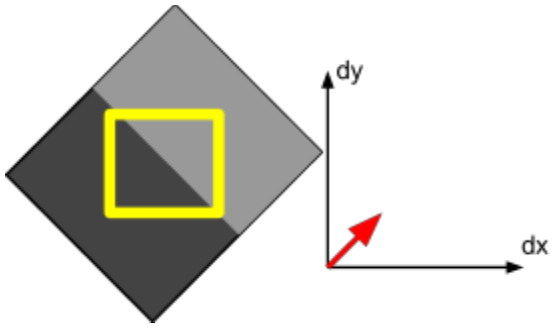
part4 - edge orientations and magnitudes

Now we have the rate at which the image is changing in the x and y directions, but it makes more sense to talk about images in terms of edges, and their orientations and intensities. As we will see, we can use some trigonometry to transform between these two representations.

First, let's see how our sobel filters respond to edges at different orientations:



The red arrow on the right shows the relative intensities of the response from the x and y sobel filter. We see that as we rotate the edge, the intensity of the response slowly shifts from the x to the y direction. We can also consider what would happen if the edge got less intense:



The response from both filters gets less intense. The orientation of the edge corresponds to the angle of the red arrow, and the intensity of the edge corresponds to its magnitude. Based on that, we create the following transformations:

$$angle = arctan(dx/dy)$$
$$magnitude = \sqrt{dx^2 + dy^2}$$

Your task is to implement these transformations in the transform_xy_theta and

transform_xy_mag functions. You will find further details in the documentation strings for these functions.

mapping to color

We now have the magnitude and orientation of the edge present at every pixel. However, this is still not really in a form that we can examine visually. The angle varies from -pi/2 to pi/2, and we don't know the scale of the magnitude. What we have to do is to figure out a way to transform this information to some sort of color values to place in each pixel.

There are many ways of doing this, and you are welcome to get creative with this and other parts of the implementation.

Currently, the edge orientations are pooled into four bins - edges that are roughly horizontal, edges that are roughly vertical, and edges at 45 degrees to the left and to the right. Each of these bins are assigned a color (vertical is yellow, etc). Then the magnitude is used to dictate the intensity of the edge. For example, a roughly vertical edge of moderate intensity would be set to a medium yellow color, or the value (0, 100, 100) (remember, opencv stores colors in BGR order). The function get_color handles this transformation.

conclusion

That's it! By being clever about how we create kernels, apply convolution and trigonometry, we were able to obtain a novel representation of an image as a collection of gradient orientations and intensities. We then came up with a way to transform this information back into an image, and created an interesting-looking result.

At this point you should be able to apply this pipeline to your own images, and tweak the different parameters to make it fit appropriately (the size and standard deviation of the gaussian kernel to account for more noise, for example).

Hopefully, you also have some ideas about how you can modify and expand this pipeline to make something unique and interesting.

Regardless of what you end up doing, please share your experiences and results with us by uploading your image to the special Image Share forum on coursera!

4. references

python - link
numpy and scipy - link

numpy user guide - link
opencv - link
image gradient - link
canny edge detection - link
sobel filter - link