An Introduction to Computer Vision

David Fridovich-Keil

Humans are visual creatures — we understand things best by seeing them. We have two eyes, each of which captures a two dimensional version of our surroundings, and somehow our brains can stitch those images together into a three dimensional worldview. But what if we replace the two eyes with digital cameras, and the brain with a computer: can we teach a computer to "see" the world around it? This week we will see just how some tasks in computer vision are quite easy, and how others are impossibly difficult, through demonstrations that you can take back home, replicate, and build upon.

First off: what is computer vision? As I mentioned before, computer vision has to do with teaching computers to recognize images, to interpret them in some way. Maybe that means identifying objects as tables and chairs and cars and people, or maybe that means dividing the image into foreground and background, or maybe that means classifying textured regions as either grass or cobblestone or brick. It could be anything; literally, anything that we do with our own eyes.

Today we're going to look at a bunch of different examples of different things we can do with computer vision. I should point out that, as you might expect with a subject this broad, the colossal amount of work that's been done in computer vision already does not even begin to compare with the breath of potential applications. You can definitely imagine tasks that we humans do with relative ease (how about estimating an object's depth from the camera?) that no one has really managed to do well with computers. It's an open area of research — that's what makes it so cool!

Ok. Down to business. The demos I show today are all written in Python, which is a freely available programming language intended for quick and easy development. I highly recommend that you check it out — it's basically a super easy-to-learn language that is both reasonably fast and extraordinarily powerful due to the wide variety of free function libraries that are available for it. In fact, all the demos today rely on the scikit-image library, which in turn relies on the scipy ("scientific Python") and numpy ("numerical Python") libraries. Other libraries you might want to check out are opencv-python ("open-source computer vision") and mahotas (another random computer vision library). I only wrote one demo myself (the one on Fourier analysis), and the rest I literally grabbed off the scikit-image website (http://scikit-image.org/docs/dev/auto_examples/). If you want to follow along, or check out the code yourself later on, just download Python (at https://www.python.org/download) and the associated libraries (lots of ways to do this — Google is probably your best bet for finding them), and run the code!

Demo time! I have 10 demos prepared. Rather than go through them all in detail here, I will just link to each one's documentation on the scikit-image site, since they do a great job explaining them. This is really intended for a reference — you can see most of the cool stuff by eye, just by running the demo. During my talk, I will try to talk through how some of the simpler algorithms work, and for those ones I'll summarize the algorithm below. Let's get started!

1. **Edge operators** are probably some of the most fundamental tools in computer vision. As the name suggests, the goal of these operators (we call them "filters") is to determine where the boundaries are in the image. Where does one object end, and another begin? This is critical for a lot of different tasks, like image segmentation (dividing the image up into sub-images that represent different regions of interest) and object recognition. There are a couple different ways to do edge detection, but probably the most common approach is to look at some approximation of a gradient filter, which measures the change in intensity value around a pixel. Areas with lots of changes in intensity are probably edges. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_edge_filter.html#example-plot-edge-filter-py.
2. **Canny edge detection** is a special flavor of edge detection that is actually an adaptation of the normal gradient-based approach. Basically the way it works is it has two thresholds for gradient magnitude, and traces regions where gradient is above the lower threshold, then discarding (i.e. treating as not an edge) if there are no pixels with gradient above the upper threshold. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_canny.html#example-plot-canny-py.
3. **Entropy** is a term that comes from chemistry, where it is used to describe the disorder of a sample. The higher the disorder, the greater the entropy. Actually, the second law of thermodynamics can be stated as "the universe tends towards increasing entropy" — that's why if I never make an effort to clean up my room, it just keeps on getting messier. Anyway, we can define a similar notion in digital information theory: entropy is the amount of information stored in a sequence of bits. The more information stored, the more complicated the patterns in the bits, and hence the more random. In computer vision, we often use local entropy — measured in some neighborhood of a pixel — to measure the texture in that area. Higher entropy means more texture. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_entropy.html#example-plot-entropy-py.
4. **Histogram equalization** is a class of algorithms that seek to improve the contrast of an image. There are many different algorithms that do this, and the example I present here shows three of the most common techniques. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_equalize.html#example-plot-equalize-py.
5. **Adaptive thresholding** is a technique for separating the pixels in every small region of the image into two classes, white and black, depending on their intensity. Compare adaptive to global thresholding — notice that global thresholding is not robust to gradual shifts in background pixel intensity (e.g. due to lighting changes or shadows), while adaptive thresholding is more sensitive to local variations in intensity. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_threshold_adaptive.html#example-plot-threshold-adaptive-py.
6. **Tinting a grayscale image** is a common filter you can apply in photo editing suites like Photoshop. It may seem complicated — it's changing the entire color of the image after all — but it turns out to be super simple. Basically, you can transform the traditional RGB (red, green, blue) color representation space into what is called HSV (hue, saturation, value/luminance). Hue and saturation are like the angle and radius on a color wheel, and value/luminance is exactly the grayscale pixel intensity. Clearly then, all we need to do to tint a grayscale image is set value equal to grayscale intensity, then select appropriate, constant hue and saturation values depending on what color tint we want. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_tinting_grayscale_images.html#example-plot-tinting-grayscale-images-py.
7. **Image denoising** is one of the classic applications of computer vision. Often times images are what we call "noisy." Noisy images look sort of static-y, in that they have random

fluctuations that make the image look like it's corrupted. There are a lot of different denoising algorithms out there, but basically all of them try to identify pixels that don't look like their surroundings, then interpolate some value for those pixels based on the surrounding pixels. One way to do this is just average all the pixels in a small region (say 3x3 pixels) and then set the center pixel to that value, and repeat. One can also do more complicated things, like replace the averaging operation with a median operation, or something like that. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_denoise.html#example-plot-denoise-py.

8. **Template matching** is a key element of object recognition. The idea is to look for a particular object in an image, given that you know what it should look like (i.e. you have a template) beforehand. One way to do this is just to slide the template across the image and look for when most of the pixels match up, but of course that is not necessarily robust to things like rescaling or lighting changes. Documentation: http://scikit-image.org/docs/dev/auto_examples/plot_denoise.html#example-plot-denoise-py.
9. **Geometric transformations** are probably the coolest thing we've seen so far. Suppose you're looking at an image of a hallway, and you want to see what it would look like from a different perspective. If we were literally standing there with the camera, of course we would just move to a new location and take a new picture, but it turns out that you can computationally change the perspective as well, without taking a new image. This is called "warping." It relies on some simple linear algebra, which is fancy language for matrix multiplication. If you're interested in the details, I highly recommend that you check out the Wikipedia page on "homography," which is the fancy term for perspective transformation. Documentation: http://scikit-image.org/docs/dev/auto_examples/applications/plot_geometric.html#example-applications-plot-geometric-py.
10. **Fourier analysis** is incredibly useful for understanding some of the deeper, hidden information content of images. Basically, Fourier analysis allows us to decompose an image (or any function, in any number of dimensions) into a set of oscillating functions called sinusoids, like sin(2*pi*k*x). These sinusoids are oriented in different directions and have different characteristic frequencies (that's "k" in the equation from the last sentence), and carry a lot of useful information. For example, looking in the frequency domain can reveal underlying periodic structure in an image. One thing that I should mention here is that taking the Fourier transform involves complex numbers — the coefficients that multiply each sinusoid are complex, with both magnitude and phase. Often times people think that the magnitude is more useful, since most of the time phase appears random. However, as I show in this demo, phase is actually much more important than magnitude. See the appendix for source code.

I hope you've enjoyed these demos! If you have any questions about computer vision, or anything else kind of related to electrical engineering or computer science, feel free to shoot me an email at dfridovi@princeton.edu any time. If this is something you'd like to hear more about next semester, I'd also really appreciate it if you emailed me to let me know.

One final comment: if this stuff gets you excited, I can't recommend the Python programming environment strongly enough. I taught myself most of this stuff just in a summer; it's free to download and incredibly easy to use. There's great online documentation and a very active forum-based user community to which you can always post questions, or just browse for fun.

Appendix

Pasted below is the source code for my Fourier analysis example. Enjoy!

```
import numpy as np
from scipy import misc
import cmath
import matplotlib.pyplot as plt

from skimage import data

# load data
pic1 = data.clock()
pic2 = data.camera()

# resize to (300, 300)
pic1 = misc.imresize(pic1, (300, 300))
pic2 = misc.imresize(pic2, (300, 300))

# compute Fourier transforms
pic1_fft = np.fft.rfft2(pic1)
pic2_fft = np.fft.rfft2(pic2)

# separate into magnitude and phase
pic1_mag = np.absolute(pic1_fft)
pic1_ph = np.angle(pic1_fft)

pic2_mag = np.absolute(pic2_fft)
pic2_ph = np.angle(pic2_fft)

# swap phases and reconstruct
cexp = np.vectorize(cmath.exp)
pic1_reconstructed = np.fft.irfft2(np.multiply(pic1_mag,
                                               cexp(1j * pic2_ph)))
pic2_reconstructed = np.fft.irfft2(np.multiply(pic2_mag,
                                              cexp(1j * pic1_ph)))

# display results
fig, ax = plt.subplots(nrows=2, ncols=4, figsize=(8,5))
plt.gray()

ax[0, 0].imshow(pic1)
ax[0, 0].axis('off')
ax[0, 0].set_title('Pic1, original')

ax[0, 1].imshow(np.fft.fftshift(np.log(pic1_mag)))
ax[0, 1].axis('off')
ax[0, 1].set_title('Pic1, magnitude')
```

```
ax[0, 2].imshow(np.fft.fftshift(pic1_ph))
ax[0, 2].axis('off')
ax[0, 2].set_title('Pic1, phase')

ax[0, 3].imshow(pic1_reconstructed)
ax[0, 3].axis('off')
ax[0, 3].set_title('Pic1, reconstructed')

ax[1, 0].imshow(pic2)
ax[1, 0].axis('off')
ax[1, 0].set_title('Pic2, original')

ax[1, 1].imshow(np.fft.fftshift(np.log(pic2_mag)))
ax[1, 1].axis('off')
ax[1, 1].set_title('Pic2, magnitude')

ax[1, 2].imshow(np.fft.fftshift(pic2_ph))
ax[1, 2].axis('off')
ax[1, 2].set_title('Pic2, phase')

ax[1, 3].imshow(pic2_reconstructed)
ax[1, 3].axis('off')
ax[1, 3].set_title('Pic2, reconstructed')

plt.show()
```