CPS296.1- Homework 2

Due on March 9, 2006

Questions may continue on the back. Please write clearly. What I cannot read, I will not grade. Typed homework is preferable. A good compromise is to type the words and write the math by hand.

This assignment explores both edge detection and tracking by inviting you to build an edge tracker. Figure 1 shows the image of a hand (this should look familiar) and the result of running the MATLAB version of Canny's edge detector on it. Both images are available from the class web page.

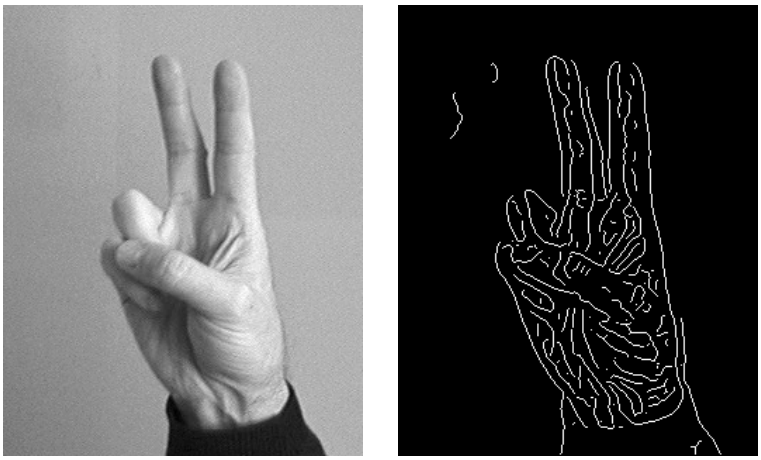

Figure 1 The image of a hand and the edges Canny's detector found with default thresholds and a smoothing parameter $\sigma = 1.5$.

The image on the right in Figure 1 is a bit map, that is, an image with binary pixel values. In some applications, this output is not sufficient, as one may need to connect the dots, that is, to output edges as properly ordered chains of pixels, one chain per edge. An algorithm that links edge pixels ("edgels" for short) into chains is called an edge linker, or an edge tracker. Ideally, a good edge tracker would produce the outline of the hand as one chain, and perhaps other details as separate chains. However, even a cursory examination of the right part of Figure 1 should convince you that linking edges can be tricky.

One approach to edge tracking may be to skip edge detection altogether, and instead follow edges in the original gray-level image by some modification of a point-feature tracker such as the one based on Sums of Squared Differences (SSD) discussed in class, and described in a handout. After all, points *along* an edge ought to be more similar to each other than points *across* the same edge, and a point-feature tracker is designed to look for similar windows. While a point-feature tracker is usually employed on a pair of images at a time, in this assignment we will use it on a single image.

To avoid issues of numerical convergence, we use a "brute-force" version of the tracker, which computes the SSD in a neighborhood of the current pixel and finds the minimum by exhaustive search. The function `brute_force` distributed with this assignment does this.

(a) For this question, assume that you only have access to the edge map, not the original image. Assuming that you want to find the outline of the hand, find at least two different types of problems that you would encounter when linking the edges in the right image of Figure 1. For each type of problem, give image coordinates of a point in the image where the problem occurs. Coordinates can be found by browsing the image with `Irfanview` (on Windows), or `xv` (on Unix). Both software packages are available for free download from the Internet. Please specify coordinates as (row, column) pairs, even if the browsers above give coordinates in the opposite order. Loosely speaking, a "problem" is something that causes undesirable results. However, exactly what constitutes a "type of problem" is up to you. There are many acceptable ways of answering this question.

(b) Let us now take the tracking approach mentioned above, and work instead on the original, gray-level input image. Where would you end up if you ran `brute_force`, with no modification, starting at pixel **p** with row coordinate 265 and column coordinate 202 (or anywhere else, for that matter)? In other words, what results from the following call, where `img` is the image on the left in Figure 1 and `p` is `[265 202]`?

```
[u, v] = brute_force(img, img, p);
```

Explain.

(c) Show a plot of the SSD function that the previous call minimized. To do this, simply call `brute_force` with two more arguments:

```
[u, v, s, drange] = brute_force(img, img, p)
```

and then type

```
mesh(drange, drange, s)
```

You can then print the figure with the `print` command.

(d) To convince the tracker to move, we add a term to the SSD function that makes it "dislike" a zero displacement. For instance, we could add a function of `dr` and `dc` (these are variables used in `brute_force.m`; read the code to understand what they mean) that decreases with increasing value of `norm([dr, dc])`. Try this. More specifically, state (in plain English and/or math) how you modify `brute_force.m` and why, show a plot of your modified SSD function at $\mathbf{p} = (265, 202)$, and show the new point found by the tracker starting at $\mathbf{p}$. There are many acceptable answers. Something smooth works better than something sharp, and scaling matters. Make sure that the term you add makes the tracker move by at least a couple of pixels away from $\mathbf{p}$, so the tracker does not get stuck tracking image noise.

(e) What is the displacement (u, v) resulting from the following call with your modified function?

```
[u, v] = brute_force(img, img, p)
```

Give the numerical values of `u` and `v`, and print the figure produced by the following command:

```
showStep(img, [p; p + [u v]])
```

(the function `showStep` is provided with this assignment). You may have to improve your answer to the previous question if the new point is nowhere near where you would expect it to be.

(f) Why does the tracker move upwards from $\mathbf{p}$, rather than downwards?

(g) Suppose that you now want to move one step further by calling the function `brute_force` on the new point `q = p + [u, v]`. It is possible that instead of going forward one more step you in fact fall back towards `p`, since `p` may look more similar to `q` than the desired "third point" `r` along the boundary does. Use the same trick you used before to prevent this from happening. Again, explain how you modify `brute_force`, show the plot of the modified SSD function for the first step (from `p` to `q`), and run showStep with a call equivalent to the following to produce a plot of the first two steps:

```
showStep(img, [p; p + [u1 v1]; p + [u2 v2]])
```

where `u1`, `v1` and `u2`, `v2` are the outputs from two runs of the twice-modified `brute_force`, one starting at `p`, the other at `q = p + [u1 v1]`. Of course, this point `q` may be somewhat different from what you obtained with your first modification. Also show what values you obtained for `u1`, `v1` and `u2`, `v2`.

(h) Starting at the same point $\mathbf{p}$, run your modified version of `brute_force` until the tracker gets lost away from the hand. Show the path you obtained by running the command

```
showStep(img, P, 1)
```

where `P` is an $n \times 2$ matrix that collects the n points traversed by your algorithm, and the third argument '1' tells `showStep` to display the full image rather than only part of it. Print and hand in the picture. Do *not* hand in the point coordinates.

(i) If your tracker made it all around the hand, skip this question. If not, do not worry. Probably nobody else got there either. On the other hand, if you did not make it anywhere close to the tip of the index finger, you may want to revisit your earlier answers and improve your code. Then, try to explain why the tracker failed where it did. Is there any relation between where your tracker failed and where Canny's edge detector failed (see Figure 1)? Briefly propose ways to fix this problem, if you think this is possible, or explain why you think it is not. You need not implement your proposals. I am not looking for a specific answer, just trying to see how you reason about these issues.