# CPS296.1- Homework 2

Due on March 6, 2003

**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 assigment is about the Marr-Poggio stereo matcher and the joys, so to speak, of Matlab programming. The assignment may be a bit verbose, but it is important that you understand all the related implementation issues.

**1**. The file `stereodots.m` on the class web page contains Matlab code for creating a random-dot stereogram. The call `[left, right] = stereodots(30, 15, 2);` will create a stereo pair of $30 \times 30$ random-dot images. A $15 \times 15$ square in the middle of the images has a disparity of 2 pixels, so that when cross-fused (wiewed with your eyes properly crossed) the middle square appears to be floating in front of the background plane. You may want to try this. One way to see the images is to run `showpair(left, right);` where `showpair.m` is a Matlab file to be found on the class web page as well.

**(a)** (No credit) Stare at the pair until you see the square floating in mid air.

**(b)** Modify `stereodots.m` to make a file `stereodots2.m` with declaration

```
function [left, right] = stereodots2(imagesize, squaresize, squaresize2, disparity, disparity2)
```

that adds a second square of side `squaresize2` in the middle of the bigger square, at a disparity `disparity2`. Hand in your code and a print of the images obtained by the calls:

```
[left, right] = stereodots2(30, 15, 7, 2, -2);
showpair(left, right);
print -deps pair.eps;
```

(you may change the last call to suit your needs).

**(c)** Since the two "cameras" are assumed to be next to each other in the standard configuration (parallel optical axes, coplanar image sensors), it would seem that a positive disparity $d = j_R - j_L > 0$ (left-image pixel coordinates to the left of right-image coordinates of the corresponding pixels) makes no sense. (Here, $j_L$ and $j_R$ are assumed to point from left to right.) How come that you still get a meaningful three-dimensional percept, assuming that you are able to fuse the two images?

**2**. The file `marrPoggio.m` on the class web page implements most of the Marr-Poggio algorithm, but has a few missing pieces, marked by the comment `%%% missing code fragment` followed by a number. Your task is to fill in these missing parts. What follows is a description of some parts of `marrPoggio.m`, so you understand what the code does. The problems you are supposed to solve are listed at the end.

Each image in the stereo pair is of dimension `rows` $\times$ `cols`. The functions `stereodots.m` and `stereodots2.m` output square images, so in this particular case we have `rows` = `cols`. There are $n_{\text{disp}}$ possible integer disparities, collected in the vector

$$\delta = \left[\begin{array}{ccc} d_{\min} & \dots & d_{\max} \end{array}\right]$$

($n_{\text{disp}}$, $\delta$, $d_{\min}$, and $d_{\max}$ are stored in `ndisp`, `delta`, `mindisp`, and `maxdisp` in the code).

The most important data structure in your algorithm is the three-dimensional array `lmatch` which is initialized by the statement:

```
lmatch = zeros(rows, cols, ndisp);
```

This array contains only zero s and ones, and represents the match candidates for all pairs of image pixels that are within the disparity bounds $d_{\min}$ and $d_{\max}$. More specifically, entry $(i, j_L, n)$ of `lmatch` is equal to 1 if `marrPoggio.m` determines that the pixel in column $j_L$ of row $i$ in the left image matches the pixel in column $j_R = j_L + \delta(n)$ of row $i$ in the right image.

Thus, `lmatch` represents `rows` copies of the match matrix in figure 5.9, page 95, of Poggio's paper, one copy for each $i$. However, since the disparities are assumed to be between $d_{\min}$ and $d_{\max}$, only a central band of width $n_{\text{disp}}$ around the diagonal needs to be stored for each match matrix. This is a bit tricky, and figures 1(a) and1(b) may help.

In these figures, the `match` array is simply `rows` copies of Poggio's array. The central band is more compactly represented in figure 1(b). Notice that horizontally aligned entries in `lmatch` represent pairs of pixels where the left-image pixel is fixed and the right-image pixel varies. For instance, in the solid oval, the left-image column subscript $j_L$ is constantly equal to 3, while the right-image column subscript $j_R$ varies. This is convenient if you want to use the built-in Matlab function `conv2()`

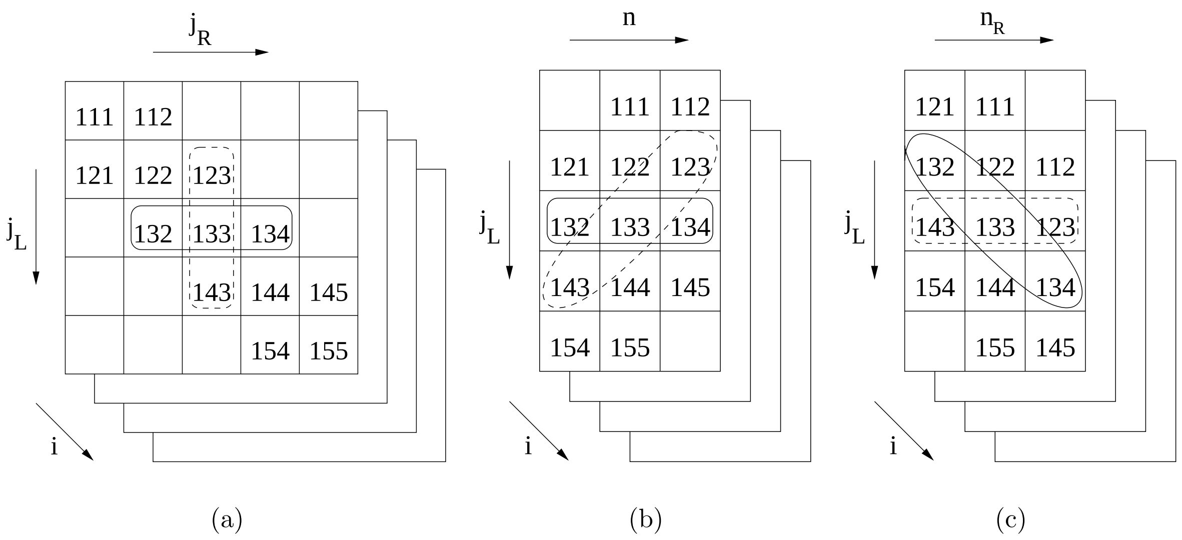

Figure 1 Different representations of the match array for `rows` $= 4$, `cols` $= 5$, $d_{\min} = -1$, $d_{\max} = 1$
(a) The array contains only zeros and ones. The numbers in all the figures are *subscripts* $i, j_L, j_R$, which refer to this first, "natural" representation of the array (in these figures, the subscript $i$ numbers what Matlab calls "rows;" $j_L$ numbers "columns;" and $j_R, n, n_R$ number "pages"). This representation is analogous to figure 5.9 of Poggio's paper. Let us call this representation the `match` array. Subscripts of positions that are always zero are not shown. (b) The lower-left figure shows how `lmatch` is actually stored in the code. (c) The lower-right figure is a third type of storage, call it `rmatch`, which you may or may not need to compute in your code. See text for details.

to quickly add values with constant $j_L$ (type `help conv2`). If you wanted to add values with constant $j_R$ instead, this would be harder, since these values are stored along NE-to-SW diagonals in `lmatch` (dashed oval in figure 1(b)). If you ever had to add those values, you could either write your own function to replace `conv2()` (which would be very slow because you would probably use `for` loops), or use `conv2()` on yet another representation, `rmatch`, shown in figure 1(c). Here, horizontally aligned entries correspond to pixel pairs where $j_R$ is constant. Fortunately, `lmatch` can be easily transformed to `rmatch`:

```
n = 1;
rmatch = zeros(size(lmatch));
for d = mindisp : maxdisp,
  rs = max(1, 1 + d);
  re = min(cols, cols + d);
  ls = max(1, 1 - d);
  le = min(cols, cols - d);
  rmatch(:, rs : re, n) = lmatch(:, ls : le, n);
  n = n + 1;
end;
```

The reverse transformation is:

```
n = 1;
lmatch = zeros(size(rmatch));
for d = mindisp : maxdisp,
  rs = max(1, 1 + d);
  re = min(cols, cols + d);
  ls = max(1, 1 - d);
  le = min(cols, cols - d);
  lmatch(:, ls : le, n) = rmatch(:, rs : re, n);
  n = n + 1;
end;
```

You may or may not need these transformations, depending on how you implement your ideas.

In `marrPoggio.m`, the array `lmatch` is initialized by the lines:

```
lmatch = zeros(rows, cols, ndisp);
n = 1;
```

```
for d = mindisp : maxdisp,
  ms = max(1, 1 - d);
  me = min(cols, cols - d);
  rs = max(1, 1 + d);
  re = min(cols, cols + d);
  ls = ms;
  le = me;
  lmatch(:, ms : me, n) = 1 - xor(l(:, ls : le), r(:, rs : re));
  n = n + 1;
end;
```

This code implements the subscript mapping in figure 1(b), and places a `1` in `lmatch` if and only if the two corresponding pixels have equal values (i.e., both black or both white).

The heart of the `marrPoggio.m` algorithm is the following code (search for it with your favorite editor):

```
support = excitation - inhibition - epsilon * mismatch;

% Update the lmatch array.
oldmatch = lmatch;
if iter == 1,
  lmatch = (support >= sigma1);
else
  lmatch = (support >= sigma);
end;
```

In code fragment 1, you will define an array `mismatch` that has a `1` if and only if the two corresponding pixels have *different* values. This is a one-liner.

The excitation for `match`$(i, j_L, j_R)$ is some measure of how many other pixels in adjacent rows and columns in, say, the left image have matches at the same disparity $d = j_R - j_L$. It is jour job to write code that computes the `excitation` array (code fragment 2).

Inhibition penalizes multiple matches for the same pixel in either image. Your code fragment 3 will compute the `inhibition` array.

Everything else in the algorithm should stay unchanged, except possibly for adding storage definitions if your code needs these.

Notice that two different thresholds `sigma` and `sigma1` are used in order to decide if `support` is sufficient for turning on a match. This is a minor point, and has the purpose of speeding up convergence. You may ignore this issue (but leave the code unaltered). If you design your `mismatch`, `excitation`, and `inhibition` so that `sigma` $= 0$ is the appropriate threshold, everything will work fine.

Finally, notice the last piece of code:

```
delta = (mindisp : maxdisp);
for i = 1 : rows,
  for j = 1 : cols,
    d = find(squeeze(lmatch(i, j, :)));
    if isempty(d),
      disparity(i, j) = NaN;
    else
      disparity(i, j) = median(delta(d));
    end;
  end;
end;
```

Since the iterative part of the algorithm does not rule out multiple matches for the same pixel (it only discourages them), the "true" disparity is taken to be the median of the disparities found for each pixel in the left image. Unmatched pixels are assigned disparity `NaN` ("Not a Number"). Most of the time this computation of the median does nothing of interest, since matches are usually unique. This is very inefficient Matlab code, because of the two nested `for` loops. Be patient!

**(a)** Write the missing code fragment 1 that creates the `mismatch` array as described above. This should be one simple instruction.

**(b)** Write the missing code fragment 2 that computes the `excitation` array. There are several possible answers here. It would be best if `excitation` were on average approximately zero for a completely wrong match. Explain the rationale for your code in detail, and hand in the fragment.

**(c)** Write the missing code fragment 3 that computes the `inhibition` array. Again, several answers are possible. It is important to quantitatively balance `inhibition` and `excitation` so that they operate correctly together. Explain the rationale for your code in detail, and hand in the fragment.

**(d)** Run `[d, lmatch] = marrPoggio(l, r);` where `l, r` is the stereo pair produced by the `stereodots2()` call from the previous problem (with those parameters). Hand in the response to the command `squeeze(lmatch(15, :, :))` (no semicolon!). This displays a $30 \times 7$ matrix of zeros and ones that refers to the pair of scanlines at row 15 in the input stereo pair, as explained above. This is just to check intermediate results in your code.

**(e)** Hand in a print of your disparity map displayed with:

```
clf;

imagesc(d);

colormap(gray);

axis square;

axis off;
```

**(f)** Do you see any problems with your disparity maps? If so, can you propose ways to fix them? If your map is roughly OK, you need not implement your fixes. If you map is perfect (unlikely), answering "no" to this question is fine.

**(g)** Why would your algorithm not work too well if the squares in the stereo pair were slanted (i.e., not parallel to the the image plane)? How could you fix this? Again, no implementation is needed here.