Image Processing Practical 2

Colour Image Segmentation and Binary Images

Introduction

Image processing/computer vision has evolved a number of techniques that are quite distinct from signal processing for dealing with the special nature of digital images. One set of very successful, fast techniques for computer vision is based on *binary image processing* which we are going to tackle in this practical.

You are given an image `croppedpeasondesk.jpg` which is available on the `LaboratoryData` directory, or off the VLE. Load this into *Matlab* using `imread` and assign it to variable `X`; display it. Note the image type (i.e. paletted or RGB (full colour)); use `size` or `whos` to confirm this. It is important that you understand how the size and type of the image data – its represemtation in *Matlab*, relate to the display of the image. If you are not sure what this means, ask a demonstrator.

The aim of this exercise is to locate all the peas, and to create a "database" of pea images (seemingly rather pointless, but bear with me; this has extensions to object recognition and to methods that are used to statistically characterise images). **Task 3** also implicitly contains an exercise in image registration, although I don't describe it as such.

Task 1

First, let us look at the difference that colour can make to a segmentation problem, in order to appreciate why colour is very important for many everyday visual tasks. To do this use the command

```
>> G = rgb2gray(X);
>> imagesc(G); colormap(gray(256)); % imagesc is like
                  %image , but scales the values in X to
                  %the range of the colourmap;
```

Note that although the peas are easily visible to your eye (because of the prior knowledge you have about peas, and the sophisticated visual processing performed by your brain), there is a very strong change in the relative object/background contrast, because of the variation in illumination and the strong reflection of a light source off the background.

Let us look at the histogram of this image. To do this, use the syntax

```
>> G = double(G); % hist needs a double type to work
>> g = G(:);      % G(:) converts the 2D matrix into a 1D
                  % array;  also could use reshape command
>> hist(g) ;      % Note the number of bins, which is 10;
>> hist(g,100);   % Note the histogram now; what is the
                  % difference
```

Do you think that it is possible to set a threshold on this image to detect the peas ? try a few thresholds, and see if you can. Use the command sequence:

```
>>  gbin = (G > T);  % For whatever threshold T you decide on
>>  imagesc(gbin);
```

Try different values of `T` and observe the effect on the binary image produced as `T` is varied. Can you find a "best" `T` ?

- Think back to your first practical, in which you looked at the three channels of the 'lily' image.
- What colour space do you think would allow you to capture the difference in the hue between the background and the peas ? Try out your idea. Remember to be careful with the type of the pixel data that you use in any calculations. You will need to use the `double` command to do this.
- Once you have obtained the right conversion (you can use `imagesc` to check the result), check the histogram of each component of the new colour space. Note carefully the differences, and relate them to the image of the corresponding colour space component.

Now, by using the histogram of the pixels in your chosen colour space component, try setting a threshold, to obtain a binary image indicating where the peas are located. Remember that in *Matlab* if

```
>> A = [1, 2, 3, 4, 5, 6, 7]; % Creates a 1x7 array of
                              % real (double precision)
                              % values 1..7
```

then

```
>> B = A > 5;
```

creates a 1x 7 logical array, with values that are 0 where $A(n) \leq 0$, for each n and unity otherwise. Use `imagesc()` to display your binary image.

Task 2

Once you have obtained a satisfactory labelling of pea vs non-pea pixels, the next step will be to assign a label to each detected region, and determine whether or not it really is a pea. This allows you to refine the detection by removing possible erroneously detected pixels.

First, let us use *connected components labelling* to label each region. Use the `help bwlabel` command to learn how to use it. Once you have obtained the labelled image by connected components, display it. Note carefully the colourmap of the result, and that every coherent region will have a unique label.

The next command we are going to use is the `regionprops` command, which gets regions properties of a labelled image. Use the system `help` to learn how this works. You also need to know something about *Matlab* structures to use this:

```
>>K=struct('Name1','Brad','Name2','Pitt','Age',30);
>>disp(K)
>>K.Age = K.Age + 1;
>>disp(K)
```

Note that structures, such as `K` above, have fields, which have values assigned to them, so that `Name1` and `Name2` are of one type (string) and `Age` is of another type (double). Fields are addressed with the name of the structure, followed by a ".", followed by the field name. So

```
>> K.Name1
```

will be `Brad`.

Assign the output of the `regionprops` command to a *Matlab* variable, such as `Regions`, and note the size and type of the output. Can you figure out how to access the information contained within it ? Can you figure out what it all means ? *Hint*: look at the dimensions of the returned result, and note how many labels there are in the labelled image using the `max` operator. Use `max(Y(:))` to find the maximum value contained in a matrix, `Y`.

If you wish to copy the fields of all elements of a particular field of a structure array into one vector, you can write a loop to concatenate each element into a separate array, or use the command `cat`, which does not require a loop. For these to work, the field will have to have the same size for all elements of the structure, but some of the outputs of the `regionprops` command do have this property.

Once you have figured out how to interpret the output of `regionprops`, answer the following:

(a) how many objects have been found ?
(b) what is the estimated mean value for pea area (in pixels) ?
(c) what is the estimated standard deviation of pea area ?
(d) how can you find those areas that are not likely to be single peas (i.e. that are clumps of peas or are small regions of supra-threshold noise) ?
(e) Interpret the `BoundingBox` field by sketching a region consisting of a simple closed shape on paper; show the lab demonstrator what you think it represents, and get confirmation.

Use the `rectangle` command to draw boxes around your detected peas by using the information from `regionprops`. Also, display the centre of each pea using the `plot(x,y,'x')` syntax. To personalize your rectangles (!), you can use the following sequence:

```
h = rectangle('Position',[lowerx lowery width height]);
set(h,'EdgeColor',[0 1 0]);  % Hint: use you colourmap
                             % knowledge o figure this
                             % out
h=text(x,y,'P');             % Add a text label to the
                             % detected region
set(h,'Color','g');          % set the color
```

Have fun! But do be careful with the mix of coordinate systems with the commands above and with the *row, column* addressing of matrices – this causes no end of confusion, but you eventually get used to it.

Task 3 The "average" pea.

In order to make explicit the idea of what the concept of an "average" pea is, we can do the following:

- First, eliminate all regions that are not likely to be peas, because of their size, and create a vector, `ValidPeaIndices`, containing the indices of only the valid regions in `Regions`.
- In order to average across pea-space, **we need to place all the pea regions into one array, such that all their centres are aligned**. Since the bounding boxes around the peas will be of different sizes, we need a bit of work to place all pea regions into a four-dimensional array by making sure each pea patch is of the same size. Why is this necessary? From a programming point of view, all slices of the matrix should be of the same number of elements, so you need the largest practical patch size that contains all peas which can then be used as the "common" dimensions for all images in the pea array. So….
- ….find the **largest bounding box** amongst the peas that you have detected. The aim here is to find a common size that can be used to generate the four dimensional pea array. This should not be too large compared to the size of most peas, so be careful in estimating this. Once you have a rough idea of what size the common bounding box size around each centroid should be, you would then extract, from the original image, the pixels within the bounding box, and stack them into a 4D array.
- In stacking the peas into a 4D array, we want to do this so that all of the pea centres are aligned in *x,y* space, or aligned with the centre of each "slice" of the 4D array as closely as possible.

You could, for example, use this syntax (**do not** try typing this into *Matlab* directly without reading a bit more !):

```
for i = 1:numberTruePeas
   imagestack(:,:,:,i)=getRegionPatch(X,Regions,. . .
       ValidPeaIndices,i,W,H); % Note the "continue line"
                               % syntax, (. . .)
end
```

where there is an **implied** function called `getRegionPatch` which returns a 3D array of size defined by `W` and `H` and the number of colour planes (3) from the i^{th} *valid* pea region. Be careful with this; there are some important considerations, which have nothing to do with *Matlab*, but have to do with largely practical issues. You do not have to write such a function, but it is a recommended way of going about creating the required 4D array. Notes:

- The code fragment I have given above is an example in which I have assumed that there is a function called `getRegionPatch`. Such a function does not exist in *Matlab*, so you would have to write your own, or find something that could replace it. **This is part of the exercise**. You may prefer to ignore the code fragment above, and use paper and pencil to work out exactly what it is I am asking you to do.
- To ensure that the pea centroids are aligned in the 4D array, you will not be able to directly use the `BoundingBox` supplied by `regionprops`, but, as alluded to above, must calculate the dimensions of a box, size `WxH`, suitable for use around each pea centroid. To do this, you will need to calculate your own locations for the corners of the bounding box using information on the centre of each pea and the maximal size of bounding box to be used, and this must be in pixel coordinates. Be careful if *x,y* and *row, column* coordinates are both being used.

If you **do** successfully manage to grab each colour pea-patch out of the image, `imagestack` will be a 4D array, and you can display all the peas using something like the following:

```
[m,n,o,p] = size(imagestack);
Plotcols = 10;  % This is kind of set arbitrarily
Plotrows = ceil(p/Plotcols); % Relative to # of peas
for i = 1:p  % 'o', not zero
      subplot(Plotrows,Plotcols,i);
      imagesc(imagestack(:,:,:,i));
      axis off;
end;
```

The montage command can also be used to do this, in which case all the images contained within the `imagestack` will be displayed in one window.

- By carefully reading the help on the `mean` command, compute and display the average pea. How does it look ?

Comment – is this alignment and averaging of peas of any use whatsoever ? Well, peas, maybe not. But for everything else – "Yes". Examples of software packages (in this case, for fMRI) where this approach is used are to be found at:

http://www.loni.ucla.edu/~thompson/disease_atlases.html

If you were to dig into the wildly popular and heavily used SPM (http://www.fil.ion.ucl.ac.uk/spm/), you would find that it contains normalisation of new scans to a standard "brain space". This includes spatial registration. See also http://imaging.mrc-cbu.cam.ac.uk/imaging/MniTalairach.

Also, for computer graphics and video games, one has http://www.faceresearch.org/demos/average . Finally, for computer vision, please see http://en.wikipedia.org/wiki/Eigenface which introduces you not only to the average face, but eigenfaces. The next piece of this practical will be an extension to computing eigenpeas.