

Edge Detection in 1-D

1 a Compute a sigmoid function with `sigmf(x,[2 0])` over the domain `x=linspace(-5,5,n)`, where `n` is a size big enough to make the sigmoid look smooth. Then take the numerical derivative of it (use MATLAB's `gradient` function to get an output that is the same size as the input). Now plot both together in the same figure. Thinking of the sigmoid as an approximation of an edge in 1-D, what does its derivative tell you? When are its values high? When are they low?

— **b** From the derivative curve you computed above, localize its maximum using the following procedure: for each value (except for the first and the last), if it is greater than or equal to its left neighbor *and* greater than its right neighbor, output a 1; else, output a 0. Plot both the output and the original derivative arrays together. (Notice that the number of output values is 2 units less than the size of the derivative array, so it should be padded with a zero on each side in order to have the same size as before.)

— **c** Verify that this same rule can be implemented in a more “clever” way like this:

```
xs = 2:n-1; xpeaks = input(1,xs-1) <= input(1,xs) & input(1,xs) > input(1,xs+1);
```

where `inputs` is a row-vector containing your derivative values.

— **d** Now you will use a different strategy for localizing the 1-dimensional “edge” in the original curve. Compute its second derivative by calling `gradient` again on the output of part **a**. Plot it on top of the first derivative. What is special about the point where the second derivative crosses the zero value?

— **e** Devise a procedure to generate a vector similar to the one you obtained in **b** and **c**, this time using the second derivative result from **d**. Remember that the output should be a binary array with a single entry having value 1. Plot it on top of the second derivative and comment on the results.

Edge Detection in 2-D

2 a We would like to adopt a similar strategy in 2-D now, which hopefully will allow us to detect edges in real images. First, instead of the derivative we will need to find partial derivatives in the x and y directions. The gradient gives us exactly that. Recall the Sobel operator from section 16.3.4 in the lecture notes and implement it as two 3-by-3 matrices (make sure to normalize them by dividing each by 8). Then load Paolina (remember to use `im2double`) and compute its two “partial derivatives” by convolving it against

each operator separately (use `conv2(im,op,'same')`). Plot the resulting images side by side in `subplots` with `imshow(im,[])`—what information do they convey? How do these relate to the result from part 1-a?

— b Now combine the results of both operators by creating a new image containing the magnitudes of the gradient (using the same magnitude formula as in Homework 1). To “clean” up the result, apply a threshold of 2 times the average pixel intensity of the magnitude image (i.e., all pixels greater than twice the average pixel intensity of this magnitude image should be 1s, the other pixels should be 0s). Plot the result with `imshow(im)`. Hint: this can be done with two lines of code only—remember to use MATLAB’s elementwise and logical operations!

— c Adapt the code from 1-c to handle 2-D arrays. To do this, fill in the missing line of code:

```
[m n] = size(im);
xs = 2:n-1;
ys = 2:m-1;
xpeaks = im(ys,xs-1) <= im(ys,xs) & im(ys,xs) > im(ys,xs+1);
ypeaks = YOUR CODE HERE
im(ys,xs)= im(ys,xs) & ( xpeaks | ypeaks );
```

Now apply it to the final magnitude image you obtained in 2-b (i.e., have `im` be your cleaned up magnitude image). What do you see? What is the purpose of this post-processing? How does it compare to what you did in part 1-b,c? Why do we need this to get an edge map?

— d The analogous operator to the second derivative in 2-D is the Laplacian. Now, instead of implementing edge thinning for this filter yourself, use MATLAB’s `edge(im,'log')` to test the Laplacian of the Gaussian edge detector. Based on your results from 1-e, how do you expect this operator to be used for finding edges in images? Are the results better or worse than what you got using the Sobel detector? What is the role of composing a Gaussian with the Laplacian in this filter?

3 a Load `Circles.tiff` and use `im2double` to convert it to an image with intensities between 0 and 1. Then run the following edge detectors using MATLAB’s `edge` function (as always, check out `help edge`), choosing the best threshold you can (i.e., one that gives you all outlines in the image):

1. Sobel
2. Canny

— b Add noise to this image using `imnoise(im, 'gaussian')`. Repeat the edge detection. Choose a new threshold and try to get as close as you can to your result in part 3-a.

— **c** Blur the noisy image from **3-b** using a Gaussian blur, and repeat edge detection. Experiment a little bit until you find a good combination of threshold and σ parameter for the Gaussian blur. Does the pre-processing with blur improve or worsen the result?

If you don't want to use your blur code from Homework 2 (e.g., it had bugs), you can use the following code to do the blurring (replace variable names as appropriate):

```
blurred = imfilter(image, fspecial('gaussian', filtersize, sigma), 'replicate')
```

A handy rule of thumb is to use a `filtersize` that is (an integer) above 6σ .

Oriented Gabor Filters

4 a Flesh out the code in `gaborfilter.m` to create a function that builds an oriented Gabor filter (see the section below for details, and the discussion in Chapter 17 in the lecture notes). Use this function to build filters that enhance edges, as well as filters that enhance bright and dark lines. (Edge-enhancing filters will have phase offset 0; bright line enhancers will have phase offset $\pi/2$; dark line enhancers will have phase offset $3\pi/2$.)

The edge filters should have 8 orientations, equally spaced in the range $[0, \frac{7\pi}{4}]$ (in radians). The line filters should each have 4 orientations equally spaced in the range $[0, \frac{3\pi}{4}]$. You should only have 4 bright line filters and 4 dark line filters, since line-enhancing filters have even symmetry (so rotating them by 180° doesn't change them), while edge enhancers have odd symmetry. This results in 16 different filters: 8 edge filters, 4 bright line filters, and 4 dark line filters.

Your writeup should consist of images of all 16 filters, as well as `Paolina.tiff` convolved with each filter. Alternatively, you can get the same results (same 16 filtered Paolina images) using half as many filters (and performing half as many `filter2` calls). If you say how, you only need to include the 8 filters you use. Use `imagesc` to show the filters, and `imshow` for the filtered images.

Be sure to include your code in the submission.

Creating a Gabor Filter

Assuming your coordinates x and y are centered at 0, define x' and y' as follows:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

In creating a matrix containing the filter values for a Gabor with orientation θ , the value at (row, col) position (i, j) is then given by

$$G_{ij} = G(x_{ij}, y_{ij}) = \sin\left(\frac{2\pi}{\omega}y'_{ij} + \phi\right) \exp\left(-\frac{x'^2_{ij}/\alpha^2 + y'^2_{ij}}{2\sigma^2}\right)$$

where ω is the wavelength of the Gabor filter in pixels (distance between peaks of the sinusoid), and ϕ is the phase. You should choose α (the aspect ratio) so that the filter is somewhat elongated in the x' direction (say what you pick and why). Choose ω and σ (the bandwidth parameter of the Gaussian envelope) in a coordinated way based on the size of the edge or line features you want to emphasize (say what you pick and why)—a good rule of thumb is to have ω be approximately 4σ . You should be able to clearly see at least one full period of the sinusoidal component of the Gabor when you display the filter, and not more than two (though it's fun to experiment with what happens when you filter an image as you change the relationship between σ and ω).

Normalize the filter so that the sum of its entries is 0, and the sum of the squares of its entries is 1.

MATLAB hint: Elementwise operations will greatly simplify your code. Also, make sure you understand what the output of `meshgrid` in the code means!

- **b** Compare the effects of an edge enhancer, a bright line enhancer, and a dark line enhancer for a given orientation. Feel free to load any other images of your choice! (Optionally, try to combine multiple orientations of a given type of filter into a single image.)
- **c** How do Gabor filters relate to receptive fields in primary visual cortex? What do you think are the advantages of using multiple orientations and edges *vs.* lines when representing a visual scene?
- **d** In the space domain, a Gabor filter is the pointwise product of a Gaussian function and a sinusoid. Recall that a Gaussian in the space domain transforms to another Gaussian in the frequency domain (having a bandwidth parameter inversely related to the the bandwidth parameter of the Gaussian in the space domain), while a sinusoid in the space domain transforms to a (pair of) delta functions in the frequency domain. Describe the frequency domain representation of a Gabor filter in terms of the frequency domain representations of its Gaussian and sinusoidal components. Plot its 2-D Fourier transform to confirm your results (remember to use `fftshift`!).

Stereo

Build a simple stereo algorithm, finishing the code in `disparity.m`. Your algorithm should take a pair of rectified stereo images that are the same size. Being rectified in this context means corresponding points in

the two images will lie on the same row.

For a patch surrounding each pixel in the left image, it should find the most similar image patch in the right image. You should restrict your search along the same row, and only a few pixels to the right and left of the patch in the left image. The measure of similarity between your patches $p_L(x_0, y_0)$ and $p_R(x_0 + \delta x, y_0)$ should be

$$\cos(\theta) = \frac{\langle p_L(x_0, y_0), p_R(x_0 + \delta x, y_0) \rangle}{\|p_L(x_0, y_0)\| \|p_R(x_0 + \delta x, y_0)\|},$$

where

$$\langle p_L(x_0, y_0), p_R(x_0 + \delta x, y_0) \rangle = \sum_{x=-M}^M \sum_{y=-M}^M I_L(x + x_0, y + y_0) I_R(x + \delta x + x_0, y + y_0),$$

$$\|p_L(x_0, y_0)\| = \sqrt{\sum_{x=-M}^M \sum_{y=-M}^M I_L(x + x_0, y + y_0)^2},$$

$$\|p_R(x_0 + \delta x, y_0)\| = \sqrt{\sum_{x=-M}^M \sum_{y=-M}^M I_R(x + \delta x + x_0, y + y_0)^2}.$$

$I_L(x, y)$ is the left image, $I_R(x, y)$ is the right image, $2M + 1$ is your patch width and height, and δx is the horizontal disparity. Your horizontal disparity δx should probably range from $\pm M$, but you can experiment with different values for different images.

Your algorithm should return a matrix of disparities for each x_0, y_0 such that the similarity between the left patch and the displaced right patch is maximized. Use `imagesc` to view the disparity matrix.

You should turn in copies of your disparity images (save the figures as image files) for the two example rectified pairs provided (`*-Left.tif` and `*-Right.tif`), as well as your code. For each pair of images, comment on the quality of the results, and whether or not the frontal parallel plane approximation is roughly valid.

MATLAB hint: Note that while using a pair of `for` loops to evaluate the sums in the above equations will work, your code will run significantly faster if you make use of MATLAB's powerful indexing capabilities to turn the image patches into vectors. You can then compute the norms and dot products in a "vectorized" way (using `sum`, `.*`, `.^`). If you have questions about this, ask one of us.