Edge Detection

We've discussed smoothing and diffusion as a way of getting rid of the effects of noise in an image. Now we're going to discuss the problem of finding the boundaries between piece-wise constant regions in the image, when these regions have been corrupted by noise. While edge detection is an ill-defined problem, we will consider edge detection with the following assumptions:

- An edge is a rapid change or discontinuity
- It is based on image properties (if a scene discontinuity fails to appear in the image, we won't find it)
- We only consider changes in intensity in gray-scale images (e.g., no color or texture)
- We only consider local image properties
- An edge may occur at different scales

As usual, we will begin by considering a 1D problem. But in the case of edge detection, we'll also have to discuss the 2D problem, because there are some issues that arise in 2D that don't show up in 1D.

1D Edge Detection

1D edge detection consists of three, key steps.

1) Reduce the effects of noise.
2) Measure the magnitude of change in the region.
3) Find peaks of change
    a. Non-maximum suppression
    b. Thresholding

The logic of this is that we of course need to avoid the effects of noise, and then we need to measure the amount of change in intensity in the image, so that we can find places where intensity is changing rapidly. We want to find peaks, because in the neighborhood of an edge, after smoothing, there may be many pixels where the image is changing rapidly, but we only want to identify one of them as the edge.

We have already discussed (1), reducing the effects of noise with smoothing. We also need to consider how to do convolution discretely. We use a discrete analog to the convolution equation:

$$h(t) = \int_{-\infty}^{\infty} f(t-u)g(u)du$$

becomes

$$h(k) = \sum_{j=-M}^{M} f(k-j)g(j)$$

One way to do this is to sample the filter (eg., sample the values of a Gaussian at some discrete points). One must be careful about two things:

1. Normalize the sampled Gaussian, so that it sums to 1.

2. Be sure to use a wide enough filter to capture the Gaussian shape. This is done heuristically, but is important. A good rule of thumb is to sample all values within 3σ of 0.

We mention that Canny has considered the question of finding the optimal way to smooth a noisy step edge in order to find edges, and has found that the optimal smoothing function is approximately a Gaussian. Canny defined optimality by defining some reasonable criteria, such as accurate localization and lack of false positives.

We also mention that it is particularly important to reduce the effects of noise before taking a derivative. One way to justify this is to note that white noise has a uniform power spectrum, while scene structure is usually more low frequency than high frequency. A derivative is a high-pass filter (the derivative of cos(kt) is –ksin(kt)). So the derivative preserves the noise much more than the structure. The way to avoid this is to low-pass filter the image first, which removes noise much more than it removes structure.

(2) in 1D, it is easy to measure the amount of change. The way that we measure change is by taking a derivative. Note that this can be done discretely, by convolution with a filter such as [1 -1] or [1/2 0 -1/2], which compute the Taylor series expansions to the derivative.

(3) Finding peaks is also simple. 1) We look for points where the magnitude of the derivative is bigger than at the two neighboring points. (We could also look for places where the second derivative is 0, although this is slightly trickier in the discrete case). This is called *non-maximum suppression* and is equivalent to finding the place where the first derivative is a maximum. 2) We also look for peaks where the magnitude of the first derivative is above some threshold. This eliminates spurious edges.

This provides a solution to the problem of finding piecewise-constant regions corrupted by noise. Some further comments:

- We saw that smoothing never introduces new extrema. It also doesn't introduce new extrema in the first derivative, that is, new edges. It does eliminate edges. As we smooth more and more, neighboring regions start to merge together.
- This means that there is a trade-off between our ability to eliminate noise, and our ability to locate small regions.
- It also means that smoothing doesn't just eliminate noise. It selects a scale at which we will detect edges.

2D Edge Detection

We perform Edge Detection by performing essentially the same steps. However, some of these steps will look a little different in 2D.

1) Reduce the effects of noise. This is exactly the same. We smooth with a Gaussian. The only difference is that we use a 2D Gaussian.

    a. One useful thing to note, though, is that we can decompose a 2D Gaussian into two 1D Gaussians, which improves efficiency.

$$G_0(x,y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\left(x^2+y^2\right) \middle/ 2\sigma^2 \right) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-x^2 \middle/ 2\sigma^2 \right) \exp\left(-y^2 \middle/ 2\sigma^2 \right)$$

2) Measure the magnitude of change in the region.
    a. In 1D we measure change with a derivative. In 2D, we measure change with a gradient. This is a 2d vector. We produce it by combining the partial derivative in the x and y directions. The direction of the gradient provides the direction of maximum change. The magnitude of the gradient tells us how fast the image is changing if we move in that direction.
    One way to think about this is that when we take the gradient, we are just looking at first order properties of the intensities. That means that we can approximate the intensities as locally linear. So think of them as lying in a plane. The gradient tells us the direction the plane is tilted, and how much it is tilted.

3) Find peaks of change
    a. Non-maximum suppression. This is a lot more complicated than in 1D.
        i. We don't just want to look at local maxima. First of all, that would be silly, because the boundaries of objects in 2D are 1D curves, whereas local maxima would be isolated points.
        Consider a simple case of a white square on a black background. The gradient magnitude will be constant along the edge, and will be decreasing in the direction orthogonal to the edge.
        So we look for points that are maxima in the direction of the gradient.
        ii. We must interpolate when the gradient doesn't point directly to a pixel.
    b. Thresholding -- An innovation of Canny was to use two thresholds (hysteresis).
        i. A high threshold. All maxima with gradient magnitudes above that are edges.
        ii. A low threshold. All maxima above this are edges if they are also connected to an edge. Note that this is a recursive definition.

Why is Canny so successful?
1. Hard to do better.
2. This might be about the best way to detect edges *locally*.
3. Improved over other approaches. Code distributed.

Corners

First, I want to point out that Canny can fail at corners. As an example, consider a square. What happens to the edge as we approach the corner? What are the gradients like? Next, how does this behavior vary depending on the angle at the corner?

So how do we find corners? This may not be too directly related to segmentation, but as long as we're on the topic…. One way to define a corner is to say that it's a region in which image gradients point in different directions. With the corner of a square, for example, the image gradients point in orthogonal directions at the corner. So, with that intuition, we can find corners by first taking a square region of the image and computing image gradients at every pixel in this square. Then we perform *Principal Component Analysis* (PCA) on the image gradients in this square. PCA tells us the dominant direction of the image gradients, and the magnitude of the image gradients orthogonal to this principal direction. If the second principal component is big, the image gradients have a strong dominant direction, but also strong components orthogonal to that direction. We do this by forming the matrix:

$$C = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

where we sum over all image gradients, *(Ix, Iy)*, in the window, and then finding the eigenvalues of this matrix. The first eigenvalue tells us the size of the first principal component, and the second eigenvalue gives the size of the second component. If both are big, we have a corner.