Cornell University
Computer Science 664

Computer Vision

Edge Detection

1 Low Level Vision

One of the primary goals of 'low level' vision is to extract geometric information from images. Low level vision operations include such things as edge detection and stereopsis. The process of edge detection is to identify locations in an image where the image intensity (brightness) changes quickly; in other words, to find the intensity boundaries in the image. The process of stereopsis is to synthesize a depth map from two or more intensity images; in other words, to determine the distance to objects given multiple intensity images. Edge detection thus recovers information about the two-dimensional geometry of the image, and stereopsis recovers information about the three-dimensional geometry of the scene. In both cases, we are extracting information about the geometry of the world from intensity images of the world; getting *from images to geometry*.

The images from most sensing devices are quantized both spatially and in terms of the values at each location in space. For example, a 'photograph' taken by a video camera and then digitized by a computer consists of a set of pixels (discrete locations), with an intensity value at each pixel. These intensities also take on discrete values in some range, generally 0-255. Common video images are of approximately 500×500 pixels. This is extremely low resolution compared to the human eye, or to many other sensors (such as scanners used for inputting documents, which have resolutions of between 300 and 600 pixels per inch). For instance, if you hold your hand at arms length and look at your thumbnail, the resolution of your fovea over this small area is about the same as the resolution of a video camera.

We will consider a number of different sensing devices, although our work will focus primarily on visual-range sensing devices (cameras). Many of these other kinds of sensors also produce 'grey level' images, in which the pixels take on a range of discrete values. For instance, there are sensors which measure the distance to the closest object – a 'depth map' (e.g., using time of reflection information). These sensors return an array of pixels where the value at each pixel indicates distance, rather than intensity. Most of the techniques that we discuss in this section of the course apply equally well to such non-visual spectrum sensing devices.

[1]Copyright © 1992, 1993, 1995 Daniel Huttenlocher

2 Edge Detection

We can express an idealized goal of the edge detection process: to produce a line drawing of a scene from an image of that scene. In its idealization, however, this goal is a gross oversimplification as we will see momentarily. Observe that, in general, the boundaries of objects in a scene tend to produce intensity boundaries in an image of that scene. For example, different objects are usually different colors or hues, which produces intensity changes in the image. In addition, different surfaces of an object receive different amounts of light, which again produces intensity changes. Thus geometric information, as might be conveyed in a line drawing, is encoded in the intensity changes in an image. Unfortunately, there are also a large number of intensity changes that are *not* due to geometry, such as surface markings, texture, and specular reflections. Moreover there are sometimes surface boundaries that do not produce very strong intensity changes. Therefore the intensity boundary information that we extract from an image will tend to indicate object boundaries, but not always. In some scenes it will be truly awful.

Figure 1a is an image of a simple scene, and Figure 1b shows the output of a particular edge detector on that scene. We can see that for this image the edge detector produces a relatively good 'line drawing' of the scene. However, say we change the lighting conditions in this scene, as illustrated in Figure 2a. Now the edge detector produces an output that is substantially less like a line drawing, as shown in Figure 2b. Our first lesson is thus that illumination conditions make a huge difference in the information that an edge detector extracts from an image.

While artists do use shadow information in producing line drawings, they somehow indicate which edges are due to shadows (for example by making blurry lines for shadow edges). Moreover, an artist tends to draw edges corresponding to object boundaries that are totally hidden in shadow. The automatically generated edges in Figure 2b miss many object edges that are hidden in the shadows, such as the bottom front corner of each of the blocks resting on the table.

Before considering some edge detection methods, we briefly summarize the goals of these methods and some of the difficulties with them. The primary goal of edge detection is to extract information about the two-dimensional projection of scene geometry for use in higher level processing, such as recognition, navigation, and hand/eye tasks. However, there are many types of physical events (in the scene) that cause intensity changes (or edges in the image). Only some of these physical events are geometric,

- object boundary – discontinuity in depth and/or surface color and texture
- surface boundary – discontinuity in surface orientation and/or color and texture
- occlusion boundary – discontinuity in depth and or surface color and texture

others do not directly reflect geometry (though we may be able to derive some geometric information from them, at least indirectly):

- specularity — direct reflection of light, such as a mirror

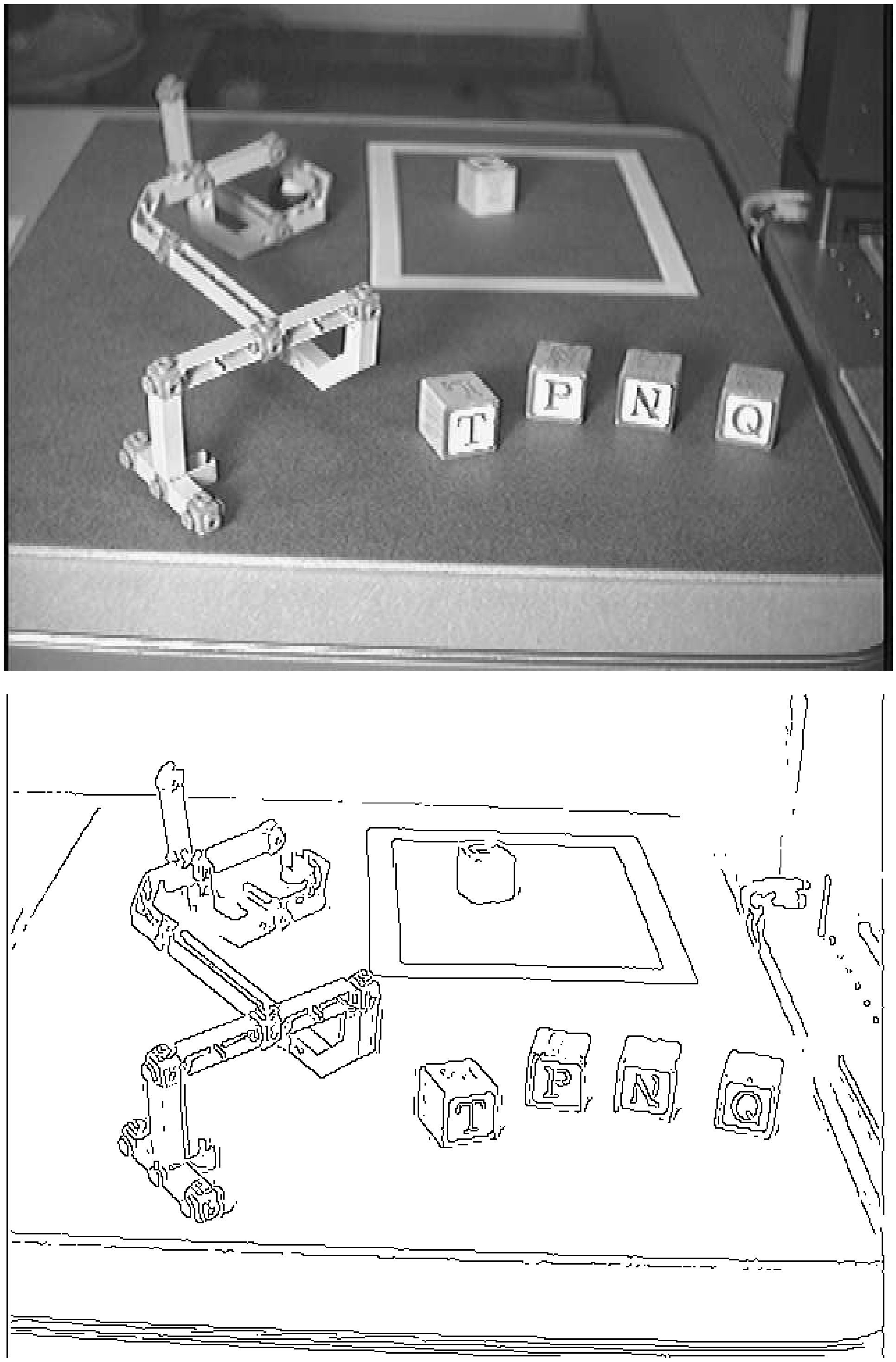

Figure 1: An image of a simple scene and intensity edges extracted from that image.

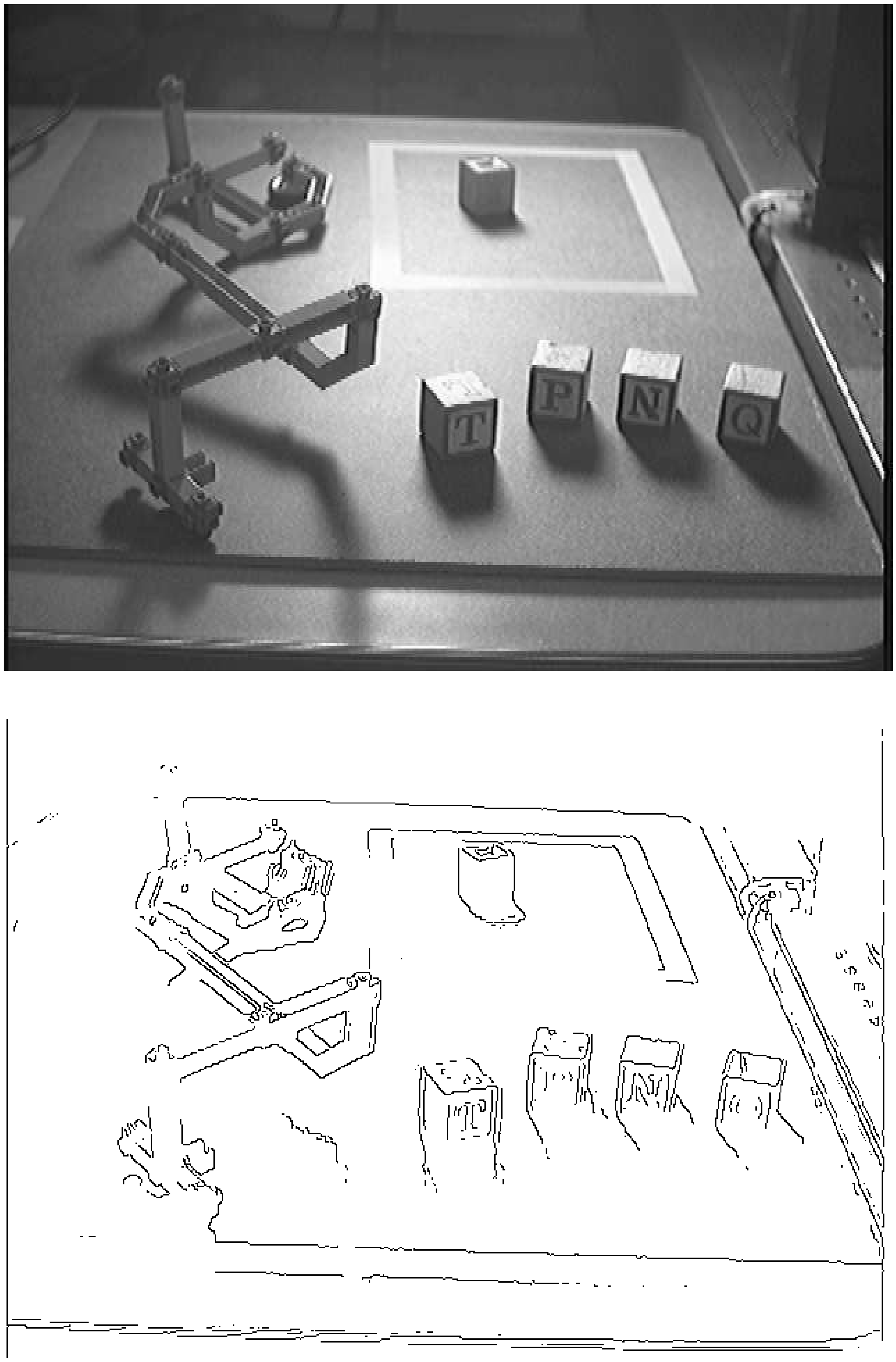

Figure 2: A less good image of the simple scene and intensity edges extracted from that image.

- shadows – from other objects, or part of the same object
- interreflections
- surface markings, texture, color changes

Note that much of the work in the field of computer graphics is concerned with the inverse of the problem described here — given the geometry of a scene, render a realistic looking image. In low level vision we are given an image of a scene, and wish to recover the geometry. Unfortunately, the image contains so many kinds of information, that it is considerably harder to get the geometry from the image than it is to render an image given the geometry (not that the graphics problem is easy either!).

2.1 Local Edge Operators

In this section we will look at some local operators for identifying edges, and consider some of their limitations. Throughout this section (and much of the course in general) we will refer to an image as $I(x, y)$, which denotes intensity as a function of some (arbitrary) coordinate system. In order to identify edges, we are interested in finding regions of the image where there is rapid change in the values of $I(x, y)$. Thus we consider local differential properties such as the *gradient* of the image intensity,

$$\nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right) .$$

This is a vector valued quantity which can be thought of intuitively as pointing in the direction of 'steepest descent' at each location (x, y) (or alternatively steepest ascent).

The magnitude of the gradient vector reflects how rapidly things are changing in this direction of steepest descent. Actually we usually use the squared gradient magnitude, to avoid computing the square root (moreover we often sloppily refer to the squared gradient magnitude as the gradient magnitude),

$$\|\nabla I\|^2 = \left(\frac{\partial I}{\partial x} \right)^2 + \left(\frac{\partial I}{\partial y} \right)^2 .$$

Simplistically speaking, where the squared gradient magnitude is large, there is an edge. That is, the image intensity is changing quickly and thus this is a boundary, or edge, in intensity. Of course, this begs the question of what is a 'large change'. We will return to this issue below (and never really find an answer that is totally satisfactory).

One nice property of the gradient magnitude is that it is rotationally symmetric, or *isotropic*. This means that it is insensitive to the orientation of the gradient vector, which is often desirable. In other words, an edge is an edge is an edge – regardless of the orientation of that edge with respect to the (arbitrary) Cartesian coordinate system. Two less desirable properties of the gradient magnitude are that it is a *nonlinear* operator, and it loses information about which side of an edge is brighter (due to the squaring which loses sign information). The 'problem' with nonlinear operators is that in general we don't understand their behavior as well as for linear systems. While a complete understanding of signal processing and linear shift-invariant systems is beyond the scope of this course, we will touch on some of the issues below (because without them you cannot really understand edge detectors).

Another local differential operator is the Laplacian of $I(x, y)$,

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \ ,$$

This second derivative operator is rotationally symmetric and also preserves the *sign* of brightness difference across an edge. The zero crossings (sign changes) of $\nabla^2 I$ correspond to intensity edges in the image, and the sign indicates which side of an edge is brighter. The Laplacian is the lowest order linear combination of partial derivatives that is isotropic (rotationally symmetric). Recall that rotational symmetry is generally viewed as desirable, otherwise our edge detector is more sensitive to some edges than others. Later on, we will see that the visual systems of humans and other animals seem to have both isotropic and non-isotropic filters.

Finite Approximations

In machine vision we do not have continuous image functions $I(x, y)$ that we can differentiate in order to compute quantities such as the gradient magnitude and the Laplacian. An image is digitized both in space and in intensity, producing an array $I[j, k]$ of 'intensity values'. These intensities are generally integers in some range, and their magnitude usually reflects the brightness level (0 is darkest and the maximum value is brightest).

Thus in order to compute local differential operators, we use finite difference approximations to estimate the derivatives. For example, for a discrete one-dimensional sampled function, represented as a sequence of values (such as an array) $F[j]$, we know that

$$\frac{dF}{dx} \approx F[j+1] - F[j],$$

and

$$\frac{d^2F}{dx^2} \approx F[j-1] - 2F[j] + F[j+1].$$

Thus we can approximate the squared gradient magnitude $\|\nabla I\|^2 = (\partial I/\partial x)^2 + (\partial I/\partial y)^2$ at the center of a 2×2 grid of pixels as follows. Consider the 2×2 discrete array, $I[j, k]$, labelled as:

j, k	j+1, k
j,k+1	j+1, k+1

Note that we consider the j-axis to be increasing to the right, and the k-axis to be increasing downwards.

Directly estimating the gradient using the finite difference approximation shown above has the undesirable effect of causing the horizontal and vertical derivatives to be estimated at different points. Thus we estimate the derivatives using axes that are rotated 45 degrees ($\pi/2$). The value of the partial first derivatives at the center of the 2×2 grid (where the grid lines cross) are given by,

$$\frac{\partial I}{\partial \hat{x}} \approx I[j+1, k+1] - I[j, k],$$

and

$$\frac{\partial I}{\partial \hat{y}} \approx I[j, k+1] - I[j+1, k].$$

So the squared gradient magnitude is

$$\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2 \approx (I[j+1, k+1] - I[j, k])^2 + (I[j, k+1] - I[j+1, k])^2. \qquad (1)$$

Note that this computation *shifts the grid.* These expressions give a finite difference approximation to the value of the gradient, and its squared magnitude, for a grid that is centered at the boundaries between the cells of $I[j, k]$, rather than on the cells of $I[j, k]$. For many applications it is fine to ignore this small bias, and to simply compute the gradient magnitude using equation (1).

Now we can write a simple program to compute the squared gradient magnitude using this equation:

```
for i from 0 to imax − 1
  do for j from 0 to jmax − 1
   do M[j,k] = (I[j+1,k+1] − I[j,k])^2 + (I[j,k+1] − I[j+1,k])^2
```

This yields an array $M[j, k]$ with 'large' values where the intensity is changing quickly. (Note that we assume the arrays are zero-based, and that the array $M[x, y]$ is undefined for $i = imax$ or $j = jmax$, because otherwise we would access beyond the boundaries of $I[j, k]$.) We can *threshold* the array $M[j, k]$ to locate 'edges', because large values indicate an edge. We will see below that this does not yield a very effective edge detector.

To estimate the Laplacian, $\nabla^2 I$, we need a 3×3 grid of pixels because as we saw above the finite difference approximation to the second derivative is approximated using 3 values. Thus if we consider the 3×3 array $I[j, k]$ indexed as follows:

j-1, k+1	j, k+1	j+1, k+1
j-1, k	j,k	j+1, k
j-1, k-1	j, k-1	j+1, k-1

The second directional derivatives at the center of this array are

$$\frac{\partial^2 I}{\partial x^2} \approx I[j-1, k] - 2I[j, k] + I[j+1, k],$$

and

$$\frac{\partial^2 I}{\partial y^2} \approx I[j,k-1] - 2I[j,k] + I[j,k+1].$$

So the Laplacian, which is just their sum, is

$$\nabla^2 I \approx I[j-1,k] + I[j,k-1] + I[j+1,k] + I[j,k+1] - 4I[j,k] \tag{2}$$

We can view this just in terms of the coefficients $1, 1, 1, 1, -4$ of equation (2) and their relative locations on the grid. So using the same 3×3 grid as just above, this yields coefficients at each location of,

	1	
1	-4	1
	1	

These coefficients can be viewed as a 'mask' or 'stencil'. That is, in order to compute equation (2) at some location $I[j_0, k_0]$, this mask is placed centered at (j_0, k_0) on the grid $I[j,k]$, and the sum of the products of the mask with the corresponding values of $I[j,k]$ is computed. In other words, this mask is just another way of writing down equation (2).

To compute the Laplacian, $\nabla^2 I$, for the entire image $I[j,k]$, the mask is shifted to be centered at each location (j,k) and the sum of products is computed to yield the value of the Laplacian at that (j,k). This operation of placing a mask at each location in an image, and summing the product of the mask with the image is the discrete version of what is known as *convolution*. We will discuss convolution in more generality below, because it is necessary for understanding better (nonlocal) edge operators.

Note that on a rectangular grid its hard to come up with an approximation to ∇^2 that is rotationally symmetric (even though the continuous operator is symmetric). Certainly the computation specified by equation (2) is not rotationally symmetric — it depends critically on the orientation of the axes. For example, if we consider a 45° rotated coordinate system, then we get

1		1
	-4	
1		1

A particularly accurate approximation to the Laplacian is given by a weighted sum of the above two approximations, where the x-y-oriented term is weighted approximately twice as much as the diagonally oriented term. This results in a mask of

1	4	1
4	-20	4
1	4	1

(The exact reasons for these coefficients have to do with the grid sampling and can be found in [3].) Note that the sum of these coefficients is zero, which is required in order for the computation not to be biased towards positive or negative (i.e., towards increases or decreases in the image intensity).

The Laplacian is a second order differential operator. Thus large changes in the original image $I[k,l]$ will be reflected by zero crossings in the Laplacian (places where the value of the Laplacian changes sign from positive to negative and vice versa). Recall also from above that the sign of the Laplacian indicates which side of an edge is brighter or darker. Thus in order to identify edges in an image using this operator, we first compute $\nabla^2 I$ and then find the zero crossings (sign changes). We will not consider the computation of $\nabla^2 I$ in more detail here, because the more general discussion of convolution below will subsume the 'particular' mask that we have developed here for the Laplacian.

Now we've seen two local (2×2 or 3×3 pixels) operators for detecting edges – the gradient magnitude and Laplacian (which are first and second order differential operators, respectively). Unfortunately these operators are almost always terrible in practice. This is the first of many "sounds good but doesn't work" stories in computer vision. The central problem with these local operators is that there is substantial local variation in a digitized image, and much of this variation is 'noise' rather than information regarding the scene. While a discussion of these noise sources is beyond the scope of this course, they are due to factors such as the Poisson nature of the individual photo sensors in many sensing devices and errors in the analog to digital conversion. As a result, there are many local changes in an image that are not due to the scene at all.

In practice this local variability in an image causes 'edges' to appear nearly everywhere. For example, Figure 3 shows the result of running a local gradient magnitude edge detector (similar to the simple program given above) on the image shown in Figure 1a. Contrast this with the edges from Figure 1b to see that we can do much better (with some nonlocal, or at least less local, edge operators).

Computer Vision Research

This brief introduction to local edge operators actually reflects a larger methodology, or approach, to research in computer vision. We start with a statement of a problem that we want to solve, such as find the edges, and derive a mathematical formulation of that problem. Then we implement some method based on the mathematical formulation, often a discrete approximation to some continuous mathematics. The implementation is then tested on some data (either from real sensors or synthetically generated data, though real data is nearly always better because of the difficulty of synthesizing 'realistic' data). Finally, based on the performance we derive a refined statement of the problem, and begin again. This process iterates until a desired level of performance on the problem has been attained, or until we run out of ideas about how to refine/improve the methods.

Thus one paradigm for computer vision research is:

1. state the problem and derive a mathematical formulation
2. implement a method based on the formal problem description
3. test the method (using synthetic and/or real data)

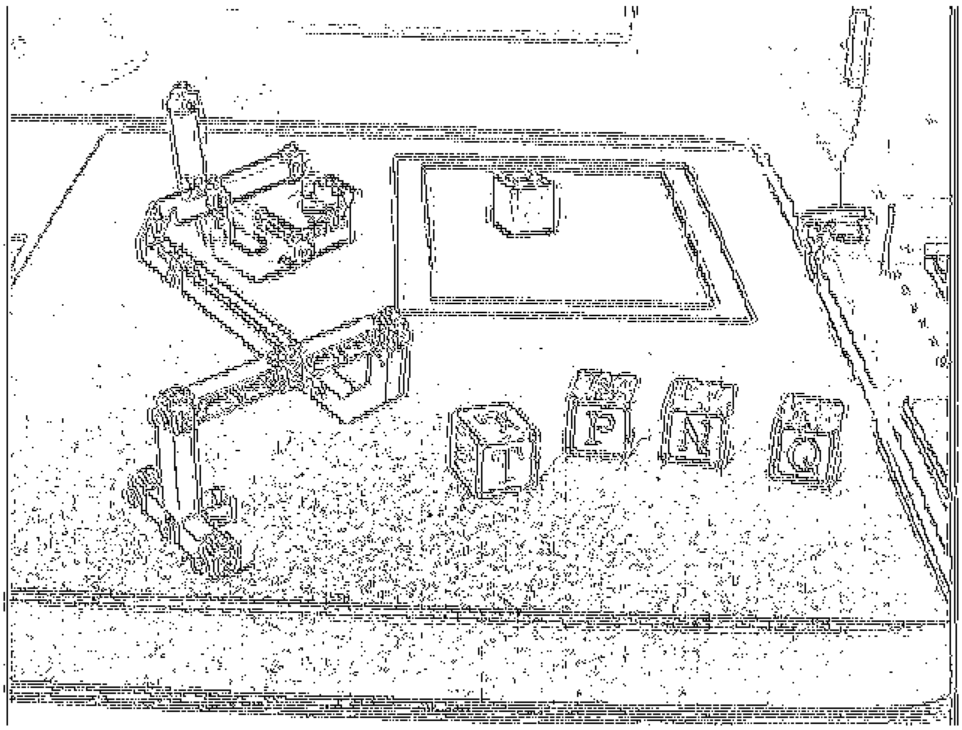

Figure 3: Local edges extracted from the 'good' image of the simple scene.

4. evaluate the results and refine the method (or reject it)

One of the interesting things about this paradigm is that in order to be successfully applied it requires a broad range of skills. The mathematical formulation of a problem can involve many different branches of mathematics (as we will see during the semester). Some major areas that we will touch on are differential equations, linear algebra, topology, group theory, combinatorial geometry, and differential geometry. The implementation of a method requires good algorithmic skills and often also requires a good understanding of numerical properties of different methods (as well as good systems and coding skills), thus spanning a wide range of areas in computer science as well. Finally, the ability to test and refine a method requires experimental skills as one generally finds in biology, physics, and similar fields. This makes computer vision a lot of fun, because we can combine skills from a number of quite different fields.

Returning to the problem at hand, of refining our edge detection methods, we will consider some changes to the Laplacian and gradient magnitude operators to yield methods with larger "areas of support". That is, we will develop techniques that consider a larger neighborhood of the image, in order to cancel out (or smooth out) small local changes. This will also involve a small detour into linear systems and convolutions, in order to enable us to understand these new methods. We will end up with two different resulting edge detectors. One, based on the smoothed gradient magnitude, was originally developed by Canny [2]. The other, based on the Laplacian of Gaussian, or $\nabla^2 G$, was originally developed by Marr and Hildreth [5].

Historical Notes

Over the last 25 years there have been a number of other local edge operators developed, which can mainly be understood in terms of the directional first and second derivative operators that we have just discussed (although they were not always presented that way in their original development). A more detailed discussion of some of these operators can be found in Ballard and Brown's book [1]. As one example, the Sobel operator is a first derivative operator in which the approximation to the directional first derivative is $F[j+1] - F[j-1]$, as opposed to $F[j+1] - F[j]$ as used above. The Sobel operator also uses a simple form of local weighting (which we will consider in more detail in the following section, when we discuss smoothing). The mask, or template, for this (directional) operator is

-1	0	1
-2	0	2
-1	0	1

for the derivative with respect to x. Note that since this edge operator is not isotropic (is sensitive to the edge orientation) a number of different edge operators must be used, sensitive to edges in various directions. This contrasts with the Laplacian and the gradient magnitude methods described above. For many biological vision systems, however, it appears there are orientation-selective filters and thus it may be that such filters are used in edge-detection type operations in these systems (there are also isotropic filters in biological vision systems so the evidence is not conclusive one way or the other).

These local edge operators (and 4×4 or 5×5 versions) work slightly better in practice than the edge operators we have discussed so far. The main reason is that they do some local averaging (or weighted smoothing) of the image as part of the processing. We now turn to a discussion of local smoothing operations, and then we put together the local smoothing with the Laplacian and gradient magnitude edge detectors. The resulting methods work better than local methods such as Sobel.

3 Convolution and Smoothing

We saw in the previous section that to detect edges in an image, we can use first and second order spatial derivatives. In particular we considered the Laplacian

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

and the squared gradient magnitude

$$\|\nabla I\|^2 = \left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2 .$$

We also saw how to approximate these quantities on a discrete grid. One remaining problem, however, is that sampled images contain a lot of high-frequency 'noise'. We saw the effects of this in Figure 3, where the application of a local gradient magnitude edge detector yielded many small edges.

This problem can be abstracted with the following example. Consider an array of numerical values,

5	6	7	6
6	5	6	5
5	5	6	7
7	6	5	6

This array has local 'boundaries' all over the place – nearly none of the neighboring values are the same. Yet, the changes are very small, and there are not 'regions' of similar value. What we would like is to produce a different array, such as

6	6	6	6
6	6	6	6
6	6	6	6
6	6	6	6

from the original array. This process of removing spatial high frequencies (small local changes) is known as *lowpass filtering*. We will discuss a particular method of lowpass filtering, which is done by *convolving* an array (in this case the image) with a lowpass filter function (generally a Gaussian operator, or normal distribution).

Note that this case should be distinguished from an array of values such as

5	5	5	6
5	5	6	6
5	5	6	6
5	6	6	6

where the differences are still relatively small, but there are two distinct regions (of value 5 and value 6). We don't want the filtering operation to remove these sorts of distinctions, and in fact we will be able to do this by appropriately selecting a *scale* or spatial extent for the lowpass filtering operation.

So we'll spend most of this section on a 'detour', looking at convolution and now to implement it on a grid (or array). As in the previous section, we start with the continuous case because it is simpler to express, and then we look at finite approximations on a grid.

Consider the following function $g(x, y)$, defined in terms of $f(x, y)$ and $h(x, y)$,

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x - \xi, y - \eta) h(\xi, \eta) d\xi d\eta \; . \tag{3}$$

We say that g is the convolution of f and h, which is written as

$$g = f \otimes h \; .$$

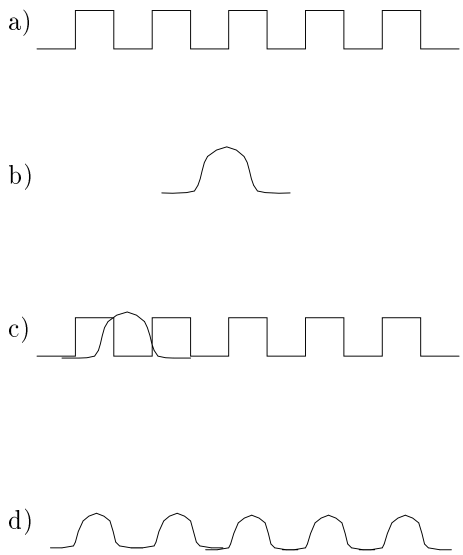

Figure 4: Illustration of one-dimensional convolution (see the text).

Convolution is *commutative*, which can be seen by a simple substitution, $\alpha = x - \xi$, $\beta = y - \eta$ then rename α to ξ and β to η,

$$a \otimes b = b \otimes a \ .$$

Convolution is also associative

$$(a \otimes b) \otimes c = a \otimes (b \otimes c) \ .$$

These two properties are very useful because they allow us to rearrange computations in whatever fashion is most convenient (or efficient).

The double integral in equation (3) may not immediately leap out at you as being intuitive — every point of the output is the sum of the product of h with a shifted version of f. This is a bit easier to illustrate for a one-dimensional function,

$$g(x) = \int_{\infty}^{\infty} f(x - \xi) h(\xi) d\xi \ .$$

Consider f being the square wave shown in Figure 4a, and h being the 'bell shaped' curve shown in Figure 4b. The value of $g(x_0)$ is then obtained by integrating (summing) the product of h with f shifted by x_0. In other words, to compute $g(x_0)$ we can view h as being superimposed on f at 'location' x_0, as illustrated in Figure 4c. Because of bell shape of this particular function h, which is nearly zero-valued for most of its range, it is mainly the nearby values of $f(x_0)$ that determine the value of $g(x_0)$. To compute $g(x)$ for all x, f is shifted to each position with respect to h, resulting in the output shown in Figure 4d.

Convolution is considerably harder to visualize at first than operations such as multiplication, because the value of $f \otimes h$ at any point is the result of *all* the points of f and

h, not just single points. It should be noted that convolution in the spatial domain is equivalent to multiplication in frequency domain, something that we will return to later. It is thus easier to understand convolution in terms of its frequency domain effects.

Convolutions are equivalent to linear shift invariant systems (LSI) — a topic central to much of signal processing which we will only touch on here. Say you are given a black box h, such that when the function f_1 is input to the box the function g_1 is output, and when the function f_2 is input, the function g_2 is output,

$$f_1 \longrightarrow \boxed{\text{h}} \longrightarrow g_1$$

$$f_2 \longrightarrow \boxed{\text{h}} \longrightarrow g_2$$

We say that h is linear shift invariant (or LSI) when it obeys linearity,

$$\alpha f_1 + \beta f_2 \longrightarrow \boxed{\text{h}} \longrightarrow \alpha g_1 + \beta g_2 \text{ for any } \alpha, \beta$$

and it is shift invariant

$$f_1(x-a, y-b) \longrightarrow \boxed{\text{h}} \longrightarrow g_1(x-a, y-b) \text{ for any } a, b\ .$$

In practice most physical systems only exhibit linear shift invariant behavior over some range (if at all). For example linearity is violated due to saturation effects — most systems cannot handle arbitrarily large (or negative) inputs. Similarly, most systems are only shift invariant over some finite extent. Any linear shift invariant system can be implemented with a convolution (where the input f is convolved with the function h computed by the black box, yielding the output g). Conversely, any convolution is a linear shift invariant system.

3.1 Discrete Convolution

In the discrete case, assume we have a square $n \times n$ array (sampled function) $h[i,j]$, $0 \le i < n$, $0 \le j < n$, and $f[i,j]$ is at least as large as h (at least $n \times n$). We define

$$g[k,y] = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f[x - \lfloor n/2 \rfloor + i, y - \lfloor n/2 \rfloor + j] h[i,j]$$

to be the discrete convolution of f and h (where $\lfloor x \rfloor$ is the integer part of x).

That is, each point x_0, y_0 of $g[x,y]$ is obtained by 'placing' h centered at x_0, y_0 over f, and summing the products of the individual entries of f and h. It is trivial to write a set of nested loops to do this computation.

```
for x from xmin to xmax
  do for y from ymin to ymax
    do sum = 0
```

```
for i from 0 to n-1
  do for j from 0 to n-1
    do sum = sum + f[x - ⌊n/2⌋ + i, y - ⌊n/2⌋ + j]h[i,j]
g[x,y] = sum
```

Note that there is an issue of how to choose the minimum and maximum values of the iteration variables x and y. Clearly these depend on the extent of $f[i,j]$ $(m \times m)$ and $h[i,j]$ $(n \times n)$, but they also depend on the method that is used to determine values 'outside' of the extent of f. For example, if x and y range over just the locations that place all of h inside of f, then $xmin$ and $ymin$ are $\lfloor n/2 \rfloor$, and $xmax$ and $ymax$ are $m - \lfloor n/2 \rfloor$. However, it is also possible to 'pad' outside the array f, to obtain an array g that is the same size as f. This amounts to handling the situations where h is positioned partially outside of f as special cases. In any implementation, these boundary issues are very important.

This discrete convolution is *exactly* what we were doing with the templates for computing the Laplacian in the previous section. That is, in order to compute the discrete approximation to the Laplacian, we set the array $h[i,j]$ to be the mask

1	4	1
4	-20	4
1	4	1

from the previous section, and make the array $f[i,j]$ be the image, $I(x,y)$. Then the convolution $g[i,j]$ is (the discrete approximation of) the Laplacian of the image, $\nabla^2 I$,

$$\nabla^2 I \approx I[i,j] \otimes h[i,j] \ .$$

In the previous section we derived this mask as an approximation to the Laplacian ∇^2, thus we have in a roundabout sort of way shown that discrete convolution can be used to compute finite differences (approximations to derivative operators)! That is, derivatives are just linear shift invariant functions, and thus convolutions can be used to compute them. In fact, this is true in the continuous case as well (but we need to introduce δ functions). It is worth remembering that differentiation is a type of convolution; that is, differentiation is a *linear operator* (LSI system). We return to this further below.

3.2 Smoothing Using Convolution

Now we want to use the convolution operator to smooth, or lowpass filter, an image in order to handle problem of high-frequency variation (sampling differences from one pixel to the next). This was the problem that motivated us to consider convolution in the first place. A simple-looking way to do this sort of smoothing is to simply average together neighboring values. This can be accomplished with an $n \times n$ mask that has the value $1/n^2$ at each location. For example, a four by four version of such a mask would be

1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16

Convolving this mask, $h[i,j]$, with an image $f[i,j]$ computes the average value over a 4×4 neighborhood for each resulting location of $g[i,j]$ (by simply summing $1/16$ of each of 16 values). However, the sudden truncation of this mask has unfortunate frequency domain effects — it causes high-frequency noise (which is known as Gibbs phenomenon, or ringing). While a formal derivation of this fact is beyond the scope of this course, intuitively one can see why this is true. As the mask $h[i,j]$ is shifted across the image $f[i,j]$, noise at the edges of a given mask position will produce significant noise in the output – because the noisy value will be part of the sum at one position of the mask, and then completely absent from the sum at the next position of the mask. One way to avoid this problem is to weight the contributions of values farther from the center of mask by less. Then as the mask is shifted along, there is only a very small change at the boundaries where values are 'no longer selected' by the mask.

We use a Gaussian to do this weighting (other functions work too). In one dimension, the Gaussian is given by

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-x^2}{2\sigma^2}} .$$

This is the canonical 'bell-shaped' or 'normal' distribution as used in statistics. The maximum value is attained at $G_\sigma(0)$, the function is symmetric about 0, and $\int G_\sigma(x)dx = 1$ (the area under the function is 1). The parameter σ controls the 'width' of the curve – the larger the value of σ the slower the function approaches 0 as $x \to \infty$ (and $x \to -\infty$).

In two dimensions, the Gaussian can be defined as

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

or equivalently

$$G_\sigma(r) = \frac{1}{2\pi\sigma^2} e^{\frac{-r^2}{2\sigma^2}}$$

where r is the distance from the origin, rather than explicit (x,y) coordinates.

Note again that $\int\int G_\sigma(x,y) = 1$. One effect of this is that $\int\int G_\sigma(x,y) \otimes f(x,y)dxdy = \int\int f(x,y)dxdy$. In other words, convolving a Gaussian with another function will preserve the area under the original function.

In the discrete case we wish to form some sort of mask, analogous to the 'averaging' mask considered above that had values of $1/n^2$ at each location. While the Gaussian is defined across the entire domain $[-\infty, \infty]$, in practice we can truncate the function after some point, because the value of G_σ becomes very small beyond some value. In general we use $[-4\sigma, 4\sigma]$, because the value of $G_\sigma(4\sigma)$ is very close to zero. Note that if we truncate the Gaussian too quickly (say at $\pm 2\sigma$), then we get the same effect that occurs when we simply average – the values near the edge of the mask are relatively large and thus a good deal of high frequency noise is produced when the mask is shifted from one position to the next.

The discretely sampled function, or mask, should sum to 1 just like in the continuous case (where the integral over the entire domain is 1). The simplest way to do this is to compute the un-normalized Gaussian

$$\mathcal{G}_\sigma[i,j] = e^{-\frac{(i^2+j^2)}{2\sigma^2}}$$

for each integer grid point (i,j), where i and j range over $\pm 4\sigma$ (i.e., the origin $(0,0)$ is at the center of the mask). Then each element is divided by the sum of all the elements, in order to yield a normalized (sampled) Gaussian which sums to 1. That is, $G_\sigma[i,j] = \mathcal{G}_\sigma[i,j]/S$, where $S = \sum_i \sum_j \mathcal{G}_\sigma[i,j]$.

For example, if $\sigma = 0.5$ then we obtain a 5×5 mask (with center at $(0,0)$ and out to $\pm 4\sigma = \pm 2$). The values sum to 1 and the entries are symmetric about the origin:

6.96E-8	2.80E-5	2.07E-4	2.80E-5	6.96E-8
2.80E-5	0.0113	0.0837	0.0113	2.80E-5
2.07E-4	0.0837	0.618	0.0837	2.07E-4
2.80E-5	0.0113	0.0837	0.0113	2.80E-5
6.96E-8	2.80E-5	2.07E-4	2.80E-5	6.96E-8

If we assume that this mask $G_\sigma[i,j]$ is indexed from -4σ to 4σ (i.e., the origin of the mask is at its center), which in this case is -2 to 2, then the convolution of the mask with the image is given by

$$I_s[x,y] = \sum_{i=-4\sigma}^{4\sigma} \sum_{j=-4\sigma}^{4\sigma} I[x+i, y+j] G_\sigma[i,j] \ . \tag{4}$$

This resulting array $I_s[x,y]$ is a lowpass filtered (or smoothed) version of the of the original image I. Figure 5 shows a smoothed version of the image from Figure 1, where the image has been convolved with a Gaussian of $\sigma = 4$. Compare it with the original image.

There is a tradeoff between the size σ of G_σ and the ability to spatially locate an event. If σ is large, we no longer really 'know' where some event (such as a change in intensity) happened, because that event has been smoothed out by a factor related to σ. This issue will become more apparent when we talk about using smoothed images for edge detection, in the following section.

Efficiently computing $G_\sigma \otimes I$

A 'direct' implementation of discrete convolution, as shown in equation (4) requires $O(m^2 n^2)$ operations for an $m \times m$ mask and an $n \times n$ image. The mask is positioned at each of the n^2 image locations and m^2 multiply and add operations are done at each position. In the case of Gaussians, the operator is *separable* and we can use this fact to speedup the convolution (for this restricted set of separable operators) to $O(mn^2)$. This is a significant savings, both theoretically and in practice, over the direct implementation. Any smoothing method (or edge detector) that uses separable filtering operators should be implemented in this manner.

g_y.

Figure 5: Lowpass filtering (smoothing) an image with a Gaussian filter.

We note that

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)} = G_x(x,y)\, G_y(x,y)$$

where

$$G_x(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-1}{2}\left(\frac{x^2}{\sigma^2}\right)}$$

and

$$G_y(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-1}{2}\left(\frac{y^2}{\sigma^2}\right)} .$$

In other words, the rotationally symmetric Gaussian, G, is just the product $G_x G_y$ of two orthogonal one-dimensional Gaussians. In the discrete case if we have two column vectors g_x and g_y that contain values approximating G_x and G_y respectively, then $g_x g_y^T = g$ (where g approximates the two-dimensional Gaussian G),

$$\left[\begin{array}{c} \\ \\ g_x \\ \\ \\ \end{array}\right] \left[\begin{array}{ccc} & g_y & \end{array}\right] = \left[\begin{array}{ccc} & & \\ & & \\ & g & \\ & & \\ & & \end{array}\right].$$

That is, each entry $[i,j]$ of g is the product of the i-th entry of g_x with the j-th entry of g_y.

How do we use this fact to speed up the computation? We further note that $G(x,y) = G_x^*(x,y) \otimes G_y^*(x,y)$, where

$$G_x^*(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-1}{2}\left(\frac{x^2}{\sigma^2}\right)} \delta(y)$$

and

$$G_y^*(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-1}{2}\left(\frac{y^2}{\sigma^2}\right)} \delta(x)$$

and δ is the unit impulse (or Dirac delta function). Intuitively speaking, the unit impulse can be thought of as being zero everywhere except at the origin where it is 'infinite'. In particular it obeys the property that $\int_a^b \delta(x)dx$ equals 1 over any interval $[a,b]$ containing the origin, and equals zero over any other interval. The unit impulse is not a function in the classical sense, because we do not specify it by giving a value of $\delta(x)$ for each x. One way to think of the unit impulse is in terms of its integral, which defines the unit step function,

$$u(x) = \int_{-\infty}^{x} \delta(t)dt,$$

where $u(x) = 1$ for $x > 0$ and $u(x) = 0$ for $x < 0$. Conversely, the derivative of $u(x)$ is $\delta(x)$.

In effect we are using the delta function as a technical 'trick' to turn the product $G = G_x G_y$ into the convolution $G = G_x^* \otimes G_y^*$. The delta function acts to 'select' just one value in computing the convolution – all the other values are zero. Recall from above that convolution sums the product of all the elements of two functions in order to produce one element of the output function, so the delta function can be thought of as selecting a single element (by multiplying all the others by zero).

Thus,

$$G \otimes I = (G_x^* \otimes G_y^*) \otimes I = G_x^* \otimes (G_y^* \otimes I),$$

by the fact that $G = G_x^* \otimes G_y^*$ and the associativity of the convolution operator. This means that in order to compute the convolution of G with I, we can instead first convolve G_x^* with I and then G_y^* with the result (note we could alternatively first convolve G_y^* with I and then G_x^* with the result, by the commutativity of convolution). Why is this an advantage? Because most of the entries of G_x^* and G_y^* are zero (due to the delta function in the definition of these functions), many of the multiplications and additions do not need to be performed. It is easiest to see this in the discrete case, so we now turn to that.

When g is an $m \times m$ mask forming a discrete sampling of G, the arrays g_x^* and g_y^* (corresponding to the functions G_x^* and G_y^*) only have m nonzero entries each. The array g_x^* has only one row of nonzero entries, and g_y^* has only one column. The nonzero row of g_x^* is just the one-dimensional vector g_x from above, and the nonzero column of g_y^* is the one-dimensional vector g_y. Diagrammatically,

$$\left[\begin{array}{c} 0 \\ [\quad g_x \quad] \\ 0 \end{array}\right] \otimes \left[\begin{array}{ccc} 0 & \left[\begin{array}{c} \\ g_y \\ \\ \end{array}\right] & 0 \end{array}\right] = \left[\begin{array}{c} \\ \quad g \quad \\ \\ \end{array}\right].$$

Since all of the zero entries in the convolution have no effect on the output, we can simply replace g_x^* with g_x, and g_y^* with g_y. Thus, the convolution $G \otimes I$ can be done by first convolving I with a column vector g_y and then convolving the result with a row vector g_x. The two-dimensional convolution $G \otimes I$ is replaced by two one-dimensional convolutions. Each of the one-dimensional convolutions involves $O(mn^2)$ operations, and thus the overall time to compute the convolution has been reduced from $O(m^2n^2)$ by the direct method to $O(mn^2)$ by this method.

In practice, the method of smoothing by two successive convolutions with one-dimensional masks is much faster than convolution with a two-dimensional mask. For example, when $\sigma = 1$ the two-dimensional mask (out to $\pm 4\sigma$) is 9×9, and thus the savings is a factor of about 5 (two 1×9 masks as opposed to one 9×9 mask). For $\sigma = 2$, it is about 10 times faster. Recall, however, that this method only works for functions that can be decomposed into the product of two one-dimensional functions (such as the Gaussian).

Approximating $G_\sigma \otimes I$

Gaussian smoothing can be approximated quite accurately in an even more efficient manner. The central idea for the speedup is based on the fact that the sum of the pixels in a $w \times h$ region around each pixel of an image can be computed in 4 operations per pixel, using dynamic programming. That is, the computation is independent of w and h, the dimensions of the window over which the summation is done. This can easily be seen in the one-dimensional case, where the sum in a window of width w can be updated from one sample to the next by simply adding in one value, and subtracting out another. This is independent of w. Using the above decomposition of symmetric functions, it is straightforward to see that the two-dimensional case can simply be expressed as two one-dimensional convolutions.

A Gaussian can be approximated as the convolution of such sums over windows (sometimes called box filters, because they add all the values in a box). The details of this are covered in the paper by Wells.

4 Marr-Hildreth Edge Detector

The Marr-Hildreth edge detector [5] is based on computing the zero crossings of the Laplacian of the Gaussian smoothed image,

$$\nabla^2(G_\sigma \otimes I)\ .$$

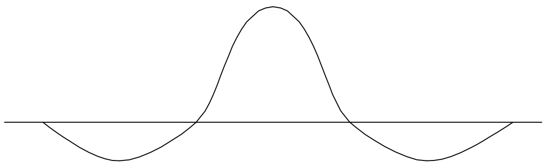

Figure 6: The Laplacian of Gaussian.

We know from the previous section that both the Gaussian smoothed image and the Laplacian can be implemented by convolution, and are hence linear operators. Thus, by associativity we can equivalently express the computation as,

$$(\nabla^2 G_\sigma) \otimes I \ .$$

The Marr-Hildreth edge detector is thus often referred to as a Laplacian of Gaussian operator, because $\nabla^2 G_\sigma$ (the Laplacian of Gaussian) is convolved with the image, I.

The one-dimensional Laplacian of Gaussian operator is illustrated in Figure 6. The two-dimensional operator is rotationally symmetric about the origin, and is often called the 'Mexican hat' operator because of its appearance.

The Laplacian of Gaussian can be approximated by a difference of two Gaussians. In the one-dimensional case this is

$$DOG(\sigma_e, \sigma_i) = \frac{1}{\sqrt{2\pi}\sigma_e} e^{\frac{-x^2}{2\sigma e^2}} - \frac{1}{\sqrt{2\pi}\sigma_i} e^{\frac{-x^2}{2\sigma i^2}} \ ,$$

which approximates a second derivative operation where $\sigma_e < \sigma_i$ (generally $\sigma_i/\sigma_e \approx 1.6$).

A number of biological vision systems appear to compute difference of Gaussians in their low-level visual processing. This was the original motivation for the Marr-Hildreth edge operator – the fact that a biological mechanism had been identified which would support a similar type of processing to the $\nabla^2 G$ operator. Computer vision research has to varying degrees been motivated by 'wetware' (or biological systems) both through neurophysiological results and psychophysical results. While it is generally difficult to formulate a precise computational model of a biological system, the existence of particular physiological methods can serve as a guide for potential algorithms. Some vision researchers argue for a tight coupling of computational vision methods to biological systems; in this course we will view such systems as providing interesting related methods, but will not try to draw close parallels between artificial and natural vision systems.

The $\nabla^2 G_\sigma$ operator is a second derivative, and thus the portions of the original image, I, where there are rapid intensity changes will show up as zero crossings of $\nabla^2 G_\sigma \otimes I$ (changes in sign). These zero crossings are then the 'edges' of the image. In the discrete case, the processing steps for this edge detection method are,

1. Smooth the image by convolution with G_σ (use the one-dimensional decomposition from Section 3).

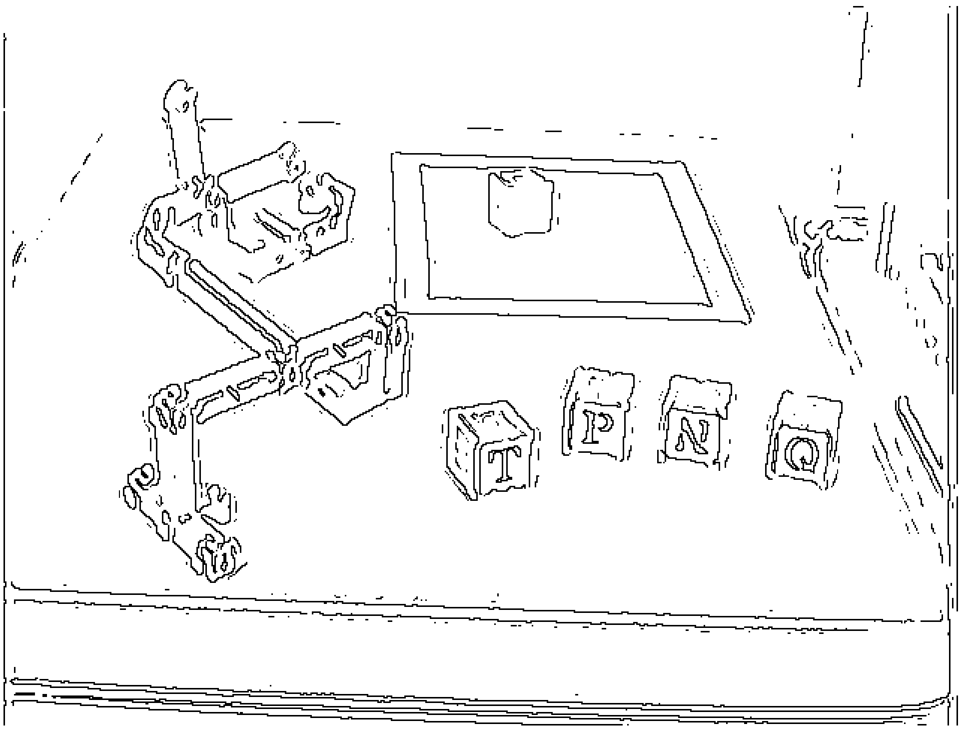

Figure 7: Laplacian of Gaussian edges for an image.

2. Apply the Laplacian to the result of the previous step (using convolution with the mask from Section 2).

3. Identify edge pixels at the boundaries of regions of constant sign in the result of the previous step.

The final step, of identifying edge pixels, can be done by viewing the sign of the Laplacian of the Gaussian smoothed image as defining a binary image (e.g., with value one where the sign is positive and zero where it is negative). Then the zero crossings are simply the boundaries of the non-zero regions of this binary array. These boundaries can be found by identifying any non-zero pixel in the binary array that has an immediate neighbor which is zero. Thus for each pixel, if it is non-zero we consider its eight neighbors. If any neighboring pixel is zero, then we classify the pixel as an edge. Figure 7 illustrates the resulting edges for the image from Figure 1.

Note one issue with a $\nabla^2 G$ edge detector is that it always forms closed edge chains, because an edge is the boundary of a region that has positive or negative sign in $\nabla^2 G \otimes I$.

5 Canny Edge Operator

The Canny [2] edge detector is based on computing the squared gradient magnitude. Local maxima of the gradient magnitude that are above some threshold are then identified as edges. This thresholded local peak detection method is called *non-maximum suppression*, or NMS. The motivation for Canny's edge operator was to derive an 'optimal' operator in the sense that it,

- Minimizes the probability of multiply detecting an edge.
- Minimizes the probability of failing to detect an edge.
- Minimizes the distance of the reported edge from the true edge

The first two of these criteria address the issue of *detection*, that is, given that an edge is present will the edge detector find that edge (and no other edges). The third criterion addresses the issue of *localization*, that is how accurately the position of an edge is reported. There is a tradeoff between detection and localization – the more accurate the detector the less accurate the localization and vice versa.

Canny considers first derivative operators. First recall that computing a derivative is a linear operation. In particular the first derivative $f'(x)$ can be computed by convolving $f(x)$ with a special function, $\Delta(x)$, the unit doublet. This function is a positive unit impulse and a negative unit impulse spaced ε apart (recall the definition of the unit impulse $\delta(x)$ from Section 3 is a function that is zero everywhere except at the origin). The computation of a derivative by convolution with a unit doublet can be thought of as corresponding to the finite difference approximation to a derivative (which becomes more accurate as ϵ gets smaller),

$$\frac{df(x)}{dx} \approx f(x) - f(x+\varepsilon) = f(x) \otimes \Delta(x)$$

as $\varepsilon \rightarrow 0$.

In order to illustrate the tradeoff between localization and detection, consider the problem of detecting a one-dimensional unit step-edge, $u(x)$. Recall from Section 3 that $du(x)/dx = \delta(x)$; the derivative of the unit step edge is the unit impulse (which also equals $u(x) \otimes \Delta(x)$). One possible method of detecting a step edge would be to differentiate $u(x)$, yielding $\delta(x)$, and then to consider $\int_a^b \delta(x)dx$ for small intervals $[a, b]$. Those intervals where the integral is nonzero (in fact equal to one) contain the step edge. However, this is analogous to smoothing the (differentiated) function by adding together neighboring values, which as we saw in Section 3 is not a very good smoothing method.

A better smoothing method is to convolve the original step edge with a Gaussian (or other lowpass filter function) and then take the derivative, $e(x) = d(G_\sigma(x) \otimes u(x))/dx$. This function will be nonzero near the location of the step edge. Given that differentiation is a linear operator (can be implemented via convolution), and that convolution is associative, we can express this edge operator $e(x)$ in two equivalent forms,

$$(G_\sigma(x) \otimes u(x))' = G'_\sigma(x) \otimes u(x),$$

(where we use the prime notation to denote taking the derivative).

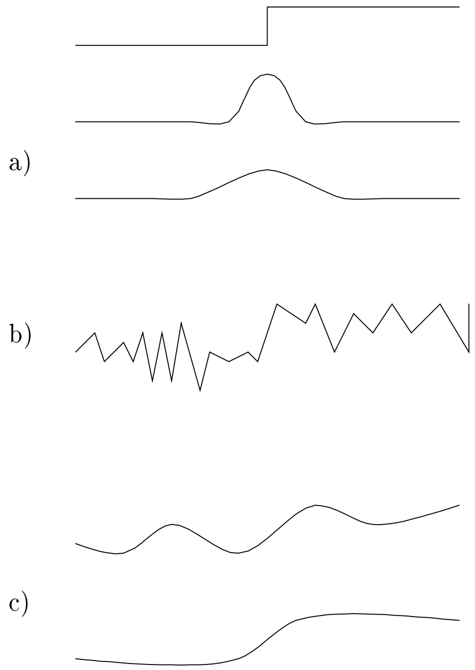

Figure 8: One dimensional derivatives of Gaussians; (a) a step-edge and its derivative at two scales of smoothing, (b) a step-edge with added noise and its derivative at one scale of smoothing, (c) a smoothed version of the step-edge from (b) making the step more apparent.

The smaller the value of σ, the smaller the domain over which $e(x)$ will be nonzero in response to a step edge. In other words, the smaller the value of σ, the better the edge is *localized*. This is illustrated in Figure 8a,which shows a step edge, and the convolution of the step edge with $G'_\sigma(x)$ for a smaller and a larger value of σ. Note how the peak corresponding to the step edge is narrower for the smaller σ. However, the smaller the value of σ, the less likely that we will *detect* an edge. In order to illustrate this, consider a step edge with noise superimposed, as illustrated in Figure 8b. Here it is difficult to see the step, because it is 'buried' in the added noise. Correspondingly, if the signal is smoothed with a small value of σ, there are relatively many large peaks in $e(x)$ (also illustrated in part b of the Figure). For larger values of σ there are fewer such local peaks, because the spatially local changes in the signal are smoothed out, leaving just the step edge. The effect of smoothing the noisy step edge with a Gaussian of large σ is illustrated Figure 8c. This makes the underlying step apparent. Thus using smaller values of σ leaves many local changes, whereas using larger ones leaves primarily the step edge (but makes its location less certain). We will return to the issue of choosing scales for smoothing below.

In one dimension, Canny shows that the operator $G'_\sigma(x)$ is an optimal detector for a step edge in terms of the detection/localization tradeoff. The derivation of this result is beyond the scope of these notes, but can be found in [2]. In 2D Canny similarly advocates (without formal arguments) the use of first spatial derivatives. As we saw above, however, the first derivative is an orientation sensitive operator (non-isotropic). Thus, rather than using the first derivative, Canny uses the squared gradient magnitude of the smoothed image,

$$m(x,y) = \|\nabla(G \otimes I_s)\|^2 = \left(\frac{\partial I_s}{\partial x}\right)^2 + \left(\frac{\partial I_s}{\partial y}\right)^2$$

which is isotropic (where I_s denotes the smoothed image).

Rather than simply thresholding $m(x,y)$ in order to identify edges, Canny's method detects local peaks in the magnitude and then thresholds the peaks. The justification for this is that edge locations will have a higher gradient magnitude than non-edges (i.e., will be local peaks in the gradient). However, rather than comparing the gradient magnitude at a given location to the values at all neighboring locations, Canny observes that we are only concerned that the gradient be a 'peak' with respect to its neighbors in the gradient direction, and not in other directions. To help understand this, consider the analogy of being on a ridge of a mountain range. The gradient direction points down from the 'top' of the ridge toward the valleys on either side. The ridge itself may move up and down such that at a given location we can be standing on the ridge but not be at a local peak in all directions (the ridge to both sides of us is higher). In the gradient direction, however, we are at a peak whenever we are on the ridge (assuming that the direction of steepest descent is always into the valley, not along the ridge). Thus defining a peak of the gradient magnitude with respect to the gradient direction allows us to detect step edges that have this type of ridge effect.

The unit vector in the gradient direction (which note is normal to a step edge) is given by

$$\hat{\nabla}(x,y) = \frac{\nabla(G \otimes I)}{||\nabla(G \otimes I)||} \; .$$

If we denote $\hat{\nabla}(x,y)$ by the unit vector $(\delta x, \delta y)$, then $m(x,y)$ is defined to be a local peak in the gradient direction when

$$m(x,y) > m(x+\delta x, y+\delta y)$$

and

$$m(x,y) > m(x-\delta x, y-\delta y) \; .$$

That is, we say that $m(x,y)$ is a local peak whenever it is greater than the values in the gradient direction and the opposite of the gradient direction. Canny calls this local peak detection operation *non-maximum suppression* (NMS). Note that in practice it is generally better to use $>$ in one direction and $\geq$ in the other, rather than $>$ in both directions, to allow for 'wide' edges as peaks.

The NMS operation still leaves many local 'peaks' that are not very large. These are then thresholded based on the gradient magnitude (or strength of the edge) to remove the small peaks. The peaks that pass this threshold are then classified as edge pixels. Canny uses a thresholding operation that has two thresholds, lo and hi. Any local maximum for which $m(x, y) >$ hi is kept as an edge pixel. Moreover, any local maximum for which $m(x, y) >$ lo and some neighbor is an edge pixel is also kept as an edge pixel. Note that this is a recursive definition — any pixel that is above the low threshold and adjacent to an *edge* pixel is itself an *edge* pixel. This form of using two thresholds allows the continuation of weaker edges that are connected to strong edges, and is a form of hysteresis.

To summarize the steps of processing for the Canny edge detector, for a discrete image $I[x, y]$,

1. Smooth the image using a 2D Gaussian (with the two 1D filters, as described in Section 3), $I_s = G_\sigma \otimes I$.

2. Compute the gradient and squared magnitude of the smoothed image,

$$\nabla I_s = \left(\frac{\partial I_s}{\partial x}, \frac{\partial I_s}{\partial y} \right)$$

$$m(x, y) = \left(\frac{\partial I_s}{\partial x} \right)^2 + \left(\frac{\partial I_s}{\partial y} \right)^2$$

3. Use the unit vector $\frac{\nabla I_s}{|\nabla I_s|} = (\delta x, \delta y)$ at each point to estimate the gradient magnitude in the gradient direction and opposite of the gradient direction. This can be done by a weighted average of the neighboring pixels in the direction $(\delta x, \delta y)$, or more simply by selecting the neighboring pixel closest to the direction $(x + \delta x, y + \delta y)$.

4. Let $p = m(x, y)$, $p_+ = m(x + \delta x, y + \delta y)$, $p_- = m(x - \delta x, y - \delta y)$. Define a peak as

$$(p > p_+ \wedge p \geq p_-) \vee (p > p_- \wedge p \geq p_+) \; .$$

5. Threshold 'strong' peaks in order to get rid of little peaks due to noise, etc. Use $|\nabla I_s|$ as measure of edge strength. Use a hysteresis mechanism as described above with two thresholds on edge strength, lo and hi.

The edges in Figure 1 are from the Canny edge detector. In practice, this edge operator (or variants of it) is the most useful and widely used.

6 Multiscale Processing

A serious practical problem with any edge detector is the matter of choosing the *scale* of smoothing (the value of σ to use). For many scenes, using the Canny edge detector with $\sigma = 1$ seems to produce 'good' results, but this is not very satisfactory. Clearly, as σ increases less and less of the detailed edges in an image are preserved (and spatial localization gets worse and worse). For many applications it is desirable to be able to process an image at multiple scales, in order to determine which edges are most significant in terms of the range of scales over which they are observed to occur.

Witkin [6] has investigated more thoroughly the idea of multi-scale signals derived from smoothing a signal with a Gaussian at different scales. He calls this *scale space* — which is a function defined over the domain of the original function, plus another dimension corresponding to the scale parameter. For example, say we have an image $I(x, y)$. The corresponding scale-space function is

$$\mathcal{I}(x, y, \sigma) = I(x, y) \otimes G_\sigma(x, y),$$

where σ is the scale parameter.

A number of natural structures can be extracted from the scale-space function, the most common of which is the 'edges' at each scale. These edges can be identified as extrema of gradient magnitude (as in the Canny operator) or as zero crossings of the Laplacian (as in the Marr-Hildreth operator). In either case, the result is a binary-valued function (or space) $\mathcal{E}(x, y, \sigma)$. Figure 9 shows a grey-level image and the edges at various scales of smoothing.

Note that as we would expect, for larger values of σ there are fewer edges (extrema of the first derivative of the smoothed function), and the spatial localization of the edges becomes poorer. Another interesting observation is that the edges do not simply appear or disappear at random as the scale parameter changes. Witkin stated two assumptions about such scale space trees, which were more or less verified later by others

1. Zero crossings connected at adjacent scales correspond to the 'same' event in $f(x)$ – no new zero crossing are introduced as σ increases.

2. The location of a zero crossing in $f(x) \otimes G_\sigma(x)$ tends to its true location in $f(x)$ as $\sigma \rightarrow 0$.

The fact that no new edges are introduced as σ increases, means that the edges form a "tree", with the root at the largest value of sigma for which there is an edge. This in principle allows a given edge to be tracked across scales (from coarse to fine), although this edge tracking problem is not easy. Note that scale space edges still do not solve the problem of what scale to pick! One option is to do further processing using the entire scale space tree, and thus avoid the problem of ever picking a scale of processing. The area where this has worked the best is in matching one-dimensional function $f(x)$, where the entire scale space tree of zero crossings is compared in order to determine the similarity of $f(x)$ and $g(x)$.

The scale space approach does partially address the issues of the tradeoff between detectability and localization. As σ gets smaller, the localization gets better and the detection gets worse (as discussed in Section 5). With the scale space tree we in part get the best of both worlds because it is possible to pick a value of σ where the detection is good, and then follow the edge contour down in scale until the localization is also good. The major problem with this approach is when a given edge contour splits into two disjoint edges as σ decreases, because then we are left with the issue of which path to

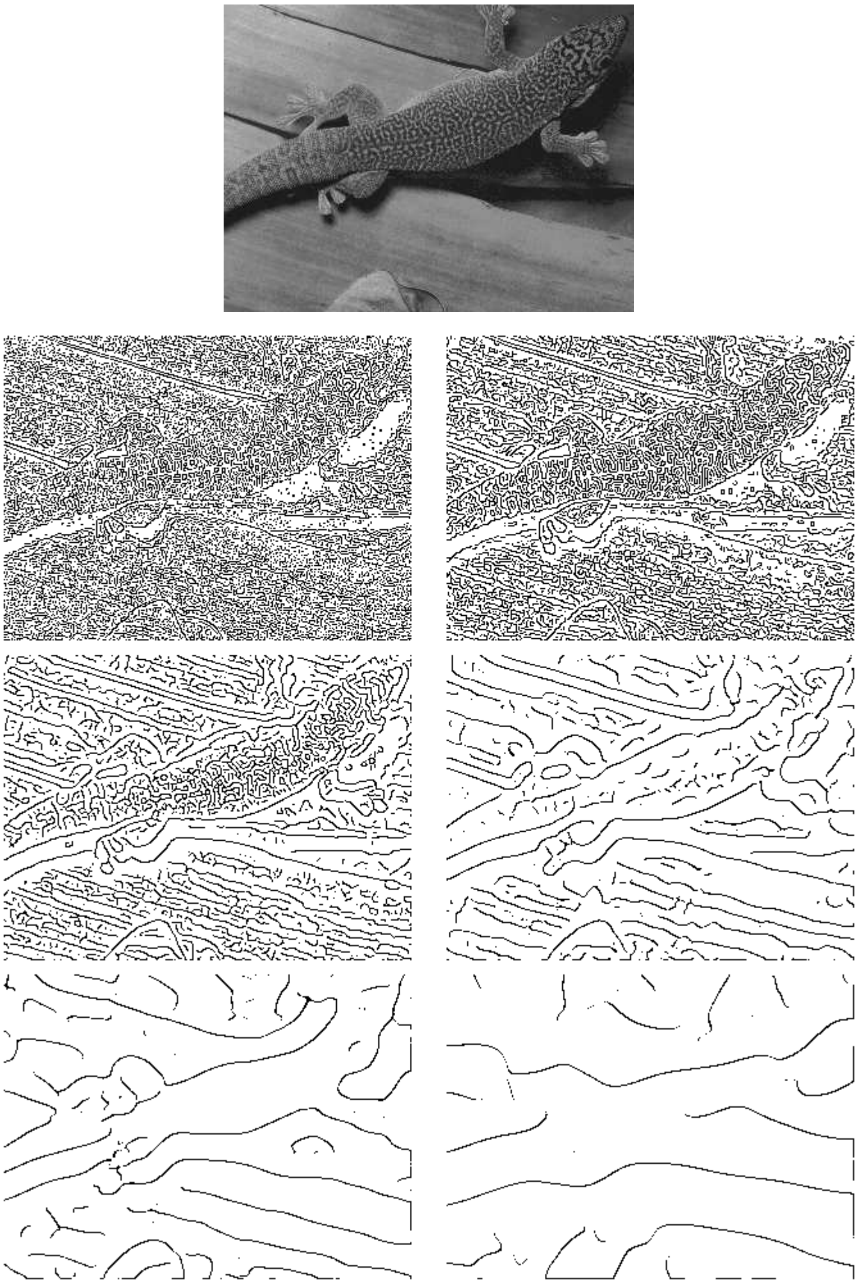

Figure 9: Canny edges at multiple scales of smoothing, $\sigma = .5, 1, 2, 4, 8, 16$. The scale-space edges are a 'stack' of these binary images (for all values of σ, not just those shown here) forming a three-dimensional binary space $\mathcal{E}(x, y, \sigma)$.

choose. Over ranges of σ where a contour does not split, however, this is a good technique for overcoming the detection/localization tradeoff.

A natural question that arises is that of reconstructing $I(x, y)$ given the edges at multiple scales, $\mathcal{E}(x, y, \sigma)$. This question has been investigated by a number of authors (e.g., [4]). Here we are not concerned with the details of these results, but simply note that for certain classes of functions $I(x, y)$ can be reconstructed up to multiplicative and additive constants. More interestingly, it is sufficient to know $\mathcal{E}(x, y, \sigma)$ just for certain values of σ in order to reconstruct $I(x, y)$ in this manner. These values are simply doublings of σ (or octaves), such as the edges shown in Figure 9 (we will return to this briefly below where dyadic wavelet transforms are introduced).

Wavelets

The entire issue of multi-scale representations and the characterization of signals from their edges at multiple scales can be viewed in terms of wavelet theory (cf. [4]). Here we briefly discuss wavelet transforms and how they apply in this context. A wavelet function is a function whose integral is zero,

$$\int_{-\infty}^{\infty} h(x) = 0 \ ,$$

and which has a scaling property. This scaling property is simply

$$h_s(x) = \frac{1}{s} h\left(\frac{x}{s}\right) \ .$$

The wavelet transform of f at scale s is then defined as

$$W_s^h f(x) = f(x) \otimes h_s(x) \ ,$$

where h is a wavelet function (has integral zero and the scaling property). That is, a wavelet transform of a function, $f(x)$, is computed by convolving the function with a wavelet function at some scale. When the wavelet function h is clear from the context, we will generally denote the wavelet transform by $W_s f(x)$ rather than $W_s^h f(x)$.

The Gaussian that we have been using for smoothing and edge detection has the scaling property

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}$$

where the 'scale factor' s is σ. Analogously the derivative (or n-th derivative) of a Gaussian is such a scaled function. The derivative of a Gaussian also has the property that its area is zero. Thus the derivative of a Gaussian is a wavelet transform. This means that the scalespace edge representation of an image, $\mathcal{E}(x, y, \sigma)$, introduced above is a form of wavelet transform of $I(x, y)$. In fact, one could think of all of the edge detectors that operate by smoothing and differentiation (which have operators whose integral is zero) in terms of wavelets. Now we take a look at why it might be worth thinking of edge detection in terms of wavelets.

First of all, wavelet transforms are a highly redundant representation of a function. One useful result regarding wavelets is that the so-called dyadic wavelet transform (the smoothing at doublings in scale, or octaves) is a complete (in fact redundant) representation of a function. The sequence of wavelets at doublings in scale is referred to as the dyadic wavelet transform,

$$\mathcal{W}f = (W_{2^j} f(x))_{j \in N}$$

where N represents the positive integers $\{1, 2, 3, \ldots\}$. From this transformation $\mathcal{W}f$ it is possible to reconstruct f, but more interestingly it is often possible to reconstruct f (or a scaled version) just from from the 'edges' of $\mathcal{W}f$. These 'edges' are the extrema of the wavelet transform when the wavelet function h is a first derivative or the zero crossings when it is a second derivative. Such a set of edges is shown in Figure 9. This means that just the edges, obtained at several scales, capture most of the information in the original image (or function). In other words, simply representing an image in terms of its edges for doublings of σ is sufficient to reconstruct the image (up to overall intensity scaling). The details of these results are beyond the scope of this course, but more information can be found in [4]. Note that similar results have also been derived without the wavelet formulation.

A second, and perhaps more interesting, property of wavelet transforms is that they can be used to represent the 'degree of discontinuity' at the edges (sharp changes) in a function. These methods even apply to discrete (sampled) representations of functions such as occur in digitization. In order to investigate this use of wavelets we must first introduce the notion of Lipschitz regularity. Let $f(x)$ be a 'generalized function', which includes any function as well as things such as the Dirac delta, $\delta(x)$ introduced above (i.e., $f(x)$ may be unbounded but $\int_a^b f(x)$ is bounded for any finite interval (a, b)). We say that $f(x)$ is uniformly Lipschitz α over an interval (a, b) if and only if there exists a constant K such that for any $x_1, x_2 \in (a, b)$

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|^\alpha ,$$

where $-1 \leq \alpha \leq 1$.

We refer to the uniform Lipschitz *regularity* of $f(x)$ as the upper bound α_0 of all such α. The larger this upper bound at x_0, the more 'regular' $f(x)$ is at x_0. For example, if $f(x)$ is differentiable at x_0 then it is Lipschitz $\alpha = 1$ at x_0. Intuitively, when f is differentiable its slope is bounded by some K and $|f(x_1) - f(x_2)|$ is thus bounded by $K|x_1 - x_2|$. On the other hand, if $f(x)$ is discontinuous but bounded at x_0 then it is Lipschitz $\alpha = 0$. Intuitively, the slope is not bounded, but the magnitude of the difference $|f(x_1) - f(x_2)|$ is bounded by some constant because the function is bounded. Finally if $f(x)$ is discontinuous and unbounded at x_0 (such as a δ function) then it is Lipschitz $\alpha = -1$. Intuitively, the difference $|f(x_1) - f(x_2)|$ becomes arbitrarily large, but only as the interval gets arbitrarily small. That is, it is bounded by some constant times $|x_1 - x_2|^{-1}$.

A theorem about wavelets is that $f(x)$ is uniformly Lipschitz α over (a, b) if and only if there exists a constant K such that for all $x \in (a, b)$ the wavelet transform satisfies

$$W_{2^j} f(x) \leq K(2^j)^\alpha .$$

In particular, this implies that if the uniform Lipschitz regularity condition at x_0 is positive then the amplitude of the wavelet transform at x_0 should decrease as the scale decreases. On the other hand if it is zero the magnitude should be independent of scale, and if it is negative, the magnitude should decrease as the scale increases. This provides a method of classifying the type of edge (differentiable, step edge or delta function) based on the evolution of the wavelet magnitude as a function of scale — does the magnitude of a peak in the smoothed derivative get larger, smaller or stay the same as the scale changes? While for sampled functions (e.g., digitized images) notions such as differentiable do not make sense, there is an analogous property for sampled functions where there is a set of scales over which an edge 'appears continuous' or 'appears to be a discontinuity'. This range of scales can be detected by examining the evolution of the magnitude of the wavelet transform edges over a range of scales. This ability to classify edges based on their type can be of considerable utility. For example, edges between an object and the background are generally intensity discontinuities, whereas shadow edges are generally continuous.

References

[1] D.H. Ballard and C.M. Brown, *Computer Vision*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

[2] J.F. Canny, "A Computational Approach to Edge Detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 8, No. 6, pp. 34-43, 1986.

[3] B.K.P. Horn, *Robot Vision*, MIT Press, Cambridge, Mass., 1986.

[4] S. Mallat and S. Zhong, "Characterization of signals from edges", *IEEE Trans. Pat. Anal. and Mach. Intel.*, 14(7), pp. 710–732, 1992.

[5] D. Marr and E. Hildreth, "Theory of Edge Detection", *Proc. of the Royal Society of London B*, Vol. 207, pp. 187-217, 1980.

[6] A.P. Witkin, "Scale Space Filtering", *Proc. of International Joint Conference on Artificial Intelligence*, pp. 1019-1022, August 1983.