CMSC 426 Problem Set 2

Lorin Hochstein - 016843860

March 3, 2003

1 Convolution with Gaussian

Claim *Let $g(t, \sigma^2)$ be a Gaussian kernel, i.e.*

$$g(t, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{t^2}{2\sigma^2}\right)$$

Then for any function $x(t)$,

$$g(t, {\sigma^2}_1) * g(t, {\sigma^2}_1) * x(t) = g(t, {\sigma^2}_2) * x(t)$$

where ${\sigma^2}_2 > {\sigma^2}_1$.

Proof

$$g(t, \sigma^2) * g(t, \sigma^2) * x(t) = g(t, \sigma^2) * \big(g(t, \sigma^2) * x(t)\big)$$

$$= g(t, \sigma^2) * \int_{-\infty}^{\infty} g(t - \tau_1) x(\tau_1) d\tau_1$$

$$= \int_{-\infty}^{\infty} g(t - \tau_2, \sigma^2) \left(\int_{-\infty}^{\infty} g(\tau_2 - \tau_1) x(\tau_1) d\tau_1\right) d\tau_2$$

$$= \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} g(t - \tau_2, \sigma^2) g(\tau_2 - \tau_1, \sigma^2) d\tau_2\right) x(\tau_1) d\tau_1$$

$$= \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{(t - \tau_2)^2}{2\sigma^2}\right) \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{(\tau_2 - \tau_1)^2}{2\sigma^2}\right) d\tau_2\right) x(\tau_1) d\tau_1$$

$$= \frac{1}{2\pi\sigma^2} \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} exp\left[-\frac{1}{2\sigma^2}\left[(t - \tau_2)^2 + (\tau_2 - \tau_1)^2\right]\right]\right) x(\tau_1) d\tau_1$$

$$= \frac{1}{2\pi\sigma^2} \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} \exp\left[-\frac{1}{2\sigma^2}(t^2 - 2t\tau_2 + \tau_2^2 + \tau_2^2 - 2\tau_2\tau_1 + \tau_1^2)\right] d\tau_2\right) x(\tau_1) d\tau_1$$

$$= \frac{1}{2\pi\sigma^2}\int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} exp\left[-\frac{1}{2\sigma^2}(2\tau_2^2 - 2t\tau_2 - 2\tau_2\tau_1)\right]d\tau_2\right) exp\left[-\frac{1}{2\sigma^2}(t^2+\tau_1^2)\right]x(\tau_1)d\tau_1$$

$$= \frac{1}{2\pi\sigma^2}\int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} exp\left[-\frac{1}{2}\left(\frac{2}{\sigma^2}\right)\tau_2^2 + \left(\frac{t+\tau_1}{\sigma^2}\right)\tau_2\right]d\tau_2\right) exp\left[-\frac{1}{2\sigma^2}(t^2+\tau_1^2)\right]x(\tau_1)d\tau_1$$

For the inner integral, we can use the identity: $\int_{-\infty}^{\infty} exp(-\frac{1}{2}Ax^2+Zx)dx = \sqrt{\frac{2\pi}{A}}exp(\frac{Z^2}{2A})$.

$$= \frac{1}{2\pi\sigma^2}\int_{-\infty}^{\infty}\sqrt{\frac{2\pi}{\frac{2}{\sigma^2}}}exp\left[\left(\frac{t+\tau_1}{\sigma^2}\right)^2 / \left(\frac{4}{\sigma^2}\right)\right] exp\left[-\frac{1}{2\sigma^2}(t^2+\tau_1^2)\right]x(\tau_1)d\tau_1$$

$$= \frac{1}{\sqrt{4\pi\sigma^2}}\int_{-\infty}^{\infty} exp\left[\frac{(t+\tau_1)^2}{4\sigma^2} - \frac{t^2+\tau_1^2}{2\sigma^2}\right]x(\tau_1)d\tau_1$$

$$= \frac{1}{\sqrt{4\pi\sigma^2}}\int_{-\infty}^{\infty} exp\left[\frac{t^2+2t\tau_1+\tau_1^2}{4\sigma^2} - \frac{2t^2+2\tau_1^2}{4\sigma^2}\right]x(\tau_1)d\tau_1$$

$$= \frac{1}{\sqrt{4\pi\sigma^2}}\int_{-\infty}^{\infty} exp\left[-\frac{1}{4\sigma^2}(t^2-2t\tau_1+\tau_1^2)\right]x(\tau_1)d\tau_1$$

$$= \int_{-\infty}^{\infty}\frac{1}{\sqrt{2\pi(2\sigma^2)}}exp\left(-\frac{(t-\tau_1)^2}{2(2\sigma^2)}\right)x(\tau_1)d\tau_1$$

$$= g(t, 2\sigma^2) * x(t)$$

■

2 1D edge detection

(a) Gaussian kernel

We can generate a Gaussian kernel simply by sampling the Gaussian function. We sample it at intervals of 1 unit. Figure 1 shows a Gaussian pulse and the samples.

The width of the Gaussian kernel will depend upon the standard deviation, σ. The larger σ is, the larger the kernel must be to capture 99% of the area. It turns out that a filter of size $(5.2\sigma)+1$ tends to have just over 99% of the area, at least for $\sigma < 25$. Figure 2 shows the area under the Gaussian pulse (without normalization) if we use our scheme, for $\sigma < 25$. Just to make sure, we capture enough area, we can always check and see if the sum of the elements is greater than 0.99. If not, we just increase the length and try again.

The code to generate the kernel is in file *gauss.m*. Figure 3 shows a plot of the kernel for $\sigma = 2$.

Below is the source code for *gauss.m*.

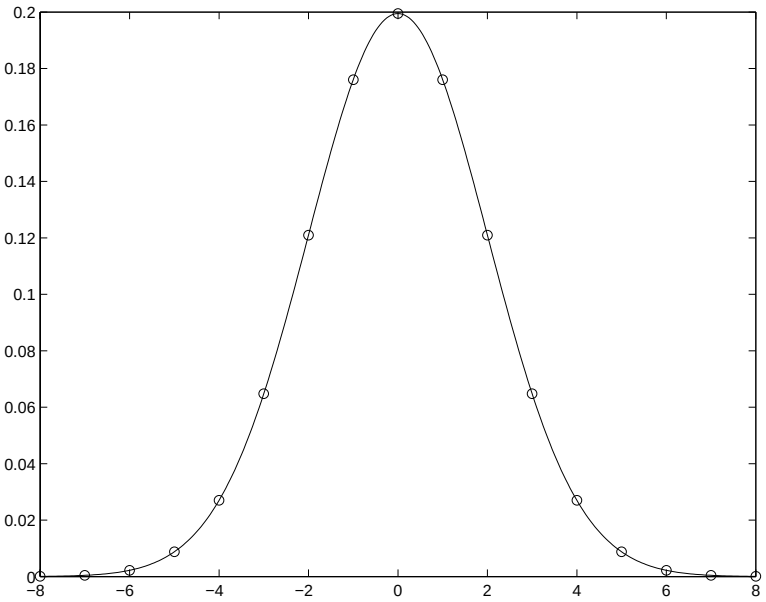

Figure 1: Sampling a Gaussian pulse

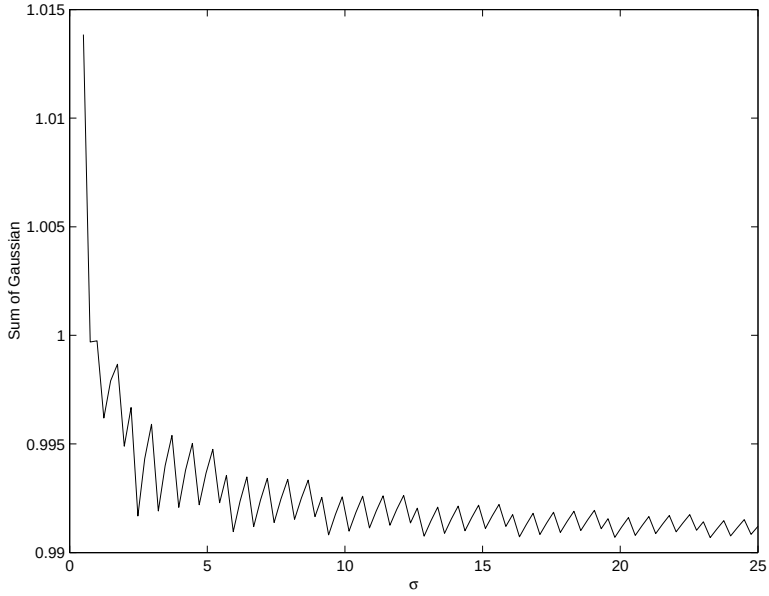

Figure 2: Area under the Gaussian pulse

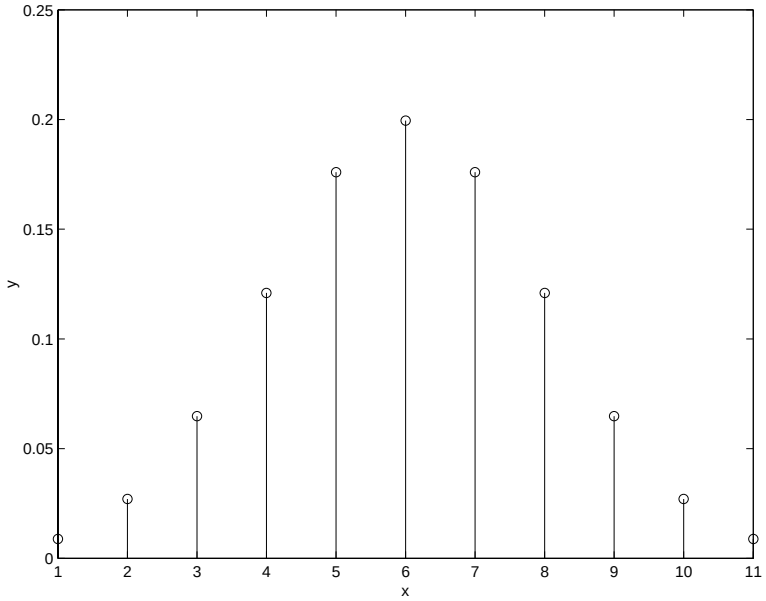

Figure 3: Gaussian kernel, $\sigma = 2$

Listing 1: gauss.m

```
function y = gauss(sigma,k)
% GAUSS generate a Gaussian kernel
%      Y = GAUSS(SIGMA,K) generates a Gaussian kernel with standard deviation
%      SIGMA. K is an optional parameter which is used to help determine the
%      size of the kernel.
%
%      Note that the generated kernel will always be of odd length
%
if nargin == 1
  k = 2.6;
end

if sigma == 0
  y = [1];
  return
end

% We need to guess a good value for the number of points.
% Some simple experiments show that 2.6*sigma will result in
% a pulse that captures 99%, without making it larger than necessary.
% We also loop to make sure that the sum is large enough.

n = max(1,round(k*sigma));

s = 0;
while s<0.99
  x = (-n):n;
```

```
  y = (1/ sqrt(2*pi*sigma^2)) * exp(-(x .^ 2) / (2*sigma^2) );
  s = sum(y);
  n = round(n * 1.1); % increase n for next iteration
end

y = y / s;
```

(b) Convolution

The formula for convolution is:

$$y[n] = \sum x[k]h[n-k]$$

Matlab has a built-in *conv* function that does convolution, padding the boundaries with zeros. However, this function increases the size of the signal by (length of the kernel) - 1. We will implement convolution here by reimplementing the *conv* function and truncating the output on each side so that it is the same size as the input. This truncation only works properly if the kernel is of odd length (otherwise, the output signal will be an odd number of pixels larger than the even signal and truncation becomes more complicated). So our convolution function restricts kernels to be of odd length. This function is implemented in the file *myconv.m*, listed below. Note that this implementation is quite slow because of the use of for loops.

Listing 2: myconv.m

```
function y = myconv(x,h)
% MYCONV convolution
%       Y = MYCONV(X,H) convolves vectors X and H. The resulting
%       vector is length LENGTH(X), assuming that H is of odd length
%       (otherwise, it will not perform the convolution)

if mod(length(h),2) == 0
  error('kernel must be of odd length')
end

N = length(x)+length(h)-1;
y = zeros(1,N);

% This is the convolution algorithm, which is what conv does
% y = conv(x,h);
%
% Note: this implementation is very slow because it doesn't use vectors
for n=1:N
  for k=1:length(x)
    if ((n-k+1)>=1) & (k <= length(x)) & (n-k+1)<=length(h)
      y(n) = y(n) + x(k) * h(n - k + 1);
    end
  end
end
```

```
% Make the output the same size as the input
y = y((length(h)-1)/2 + 1 : end - (length(h)-1)/2 );
```

Figure 4 shows the Gaussian kernel convoluted with itself two times and three times (i.e. $g(n) * g(n) * g(n)$) and $g(n) * g(n) * g(n) * g(n)$).

Note how both signals have the same shape, but the second Gaussian pulse is lower and wider than the first (i.e. its variance is higher). The source code to compute these figures is from file *two_b.m*, shown below.

Listing 3: two_b.m

```
g = gauss(2);
g1 = myconv(g,g);
g2 = myconv(g1,g);
g3 = myconv(g2,g);

figure;
subplot(2,1,1)
stem(g2)
grid on
xlabel('x')
ylabel('y')
title('Gaussian convolved with itself two times');
axis([0 12 0 .15])

subplot(2,1,2)
stem(g3)
grid on
xlabel('x')
ylabel('y')
title('Gaussian convolved with itself three times');
axis([0 12 0 .15])

print -deps 2b
```

(c) 1D edge detector

We construct an edge detector as mentioned in class, generating a kernel by convolving a Gaussian pulse with a first derivative approximation $[-1, 0, 1]$, convolving this new kernel with the signal, and then comparing the result against the threshold to locate the locations in the signal with the highest rate of change. Our edge detector has been implemented in the file *find_edges.m*, shown below

Listing 4: find_edges.m

```
function e = find_edges(x,s,t)
% FIND_EDGES find edges in a 1D signal
%       E = FIND_EDGES(X,S,T) locates the edges in the signal X, using a
%       Gaussian kernel for smoothing with standard deviation S, and a
%       threshold T for the strength of the edge. E is a vector which contains
%       the indexes which contain edges in X.
```

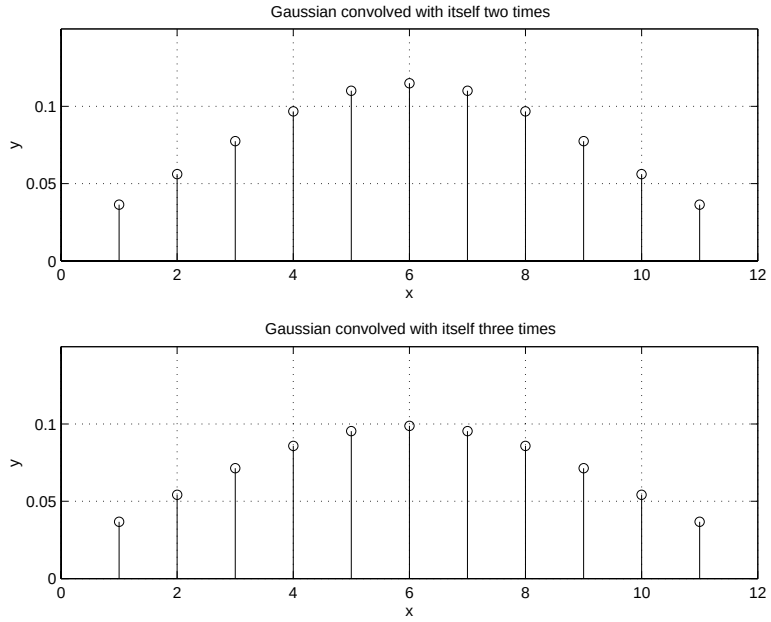

Figure 4: Convolution 2 and 3 times with Gaussian kernel, $\sigma = 2$

```
g = gauss(s);

% Generate the kernel by combining a Gaussian filter and a first derivative
% operator. Use built-in conv because we don't care about kernel length
% increasing
k = conv(g,[1 0 -1]);

% Apply the kernel and take the absolute value
y = abs(myconv(x,k));

% Threshold y and convert to indexes to find candidates points for peaks
candidates = find(y>=t);

e = [];
% Check each candidate to make sure it's a local max. Don't consider
% borders for edges
for c=candidates
  if (c>1) & (c<length(y)) & (y(c)>=y(c-1)) & y(c)>y(c+1)
    e(end+1) = c;
  end
end
```

We test our edge detector by applying it to signal I.

If we apply our edge detector with no smoothing and a threshold of 0.5, we get the following result:

```
>> I = [zeros(1,50),.9*ones(1,10),zeros(1,10),.6*ones(1,40),zeros(1,50)];
>> e = find_edges(I,0,.5)
```

```
e =

    51    61    71   111
```

The image is shown in Figure 5. The circles in Figure 5 indicate the locations where our edge detector found edges.

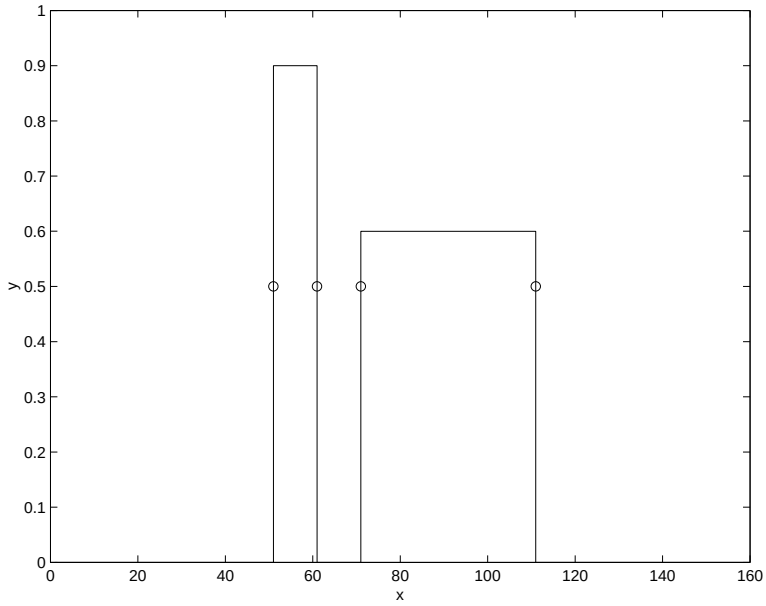

Figure 5: A 1D signal with sharp edges

(d) Smoothing

In this section, we show how well our edge detection algorithm works under Gaussian smoothing, in the absence of noise. Figure 6 shows how the location of detected edges varies with the standard deviation of the Gaussian kernel. Since there is no noise, we use a very low threshold of .01.

One of the edges begin to drift somewhat around $\sigma = 2$. However, performance is still fairly good until around $\sigma = 8$, when one of the edges is no longer detected.

The Matlab file *two_d.m* which generated this diagram is shown below.

Listing 5: two_d.m

```
I = [zeros(1,50),.9*ones(1,10),zeros(1,10),.6*ones(1,40),zeros(1,50)];
thresh = .01;

n = 1;

figure

for s = .5:.5:25
```

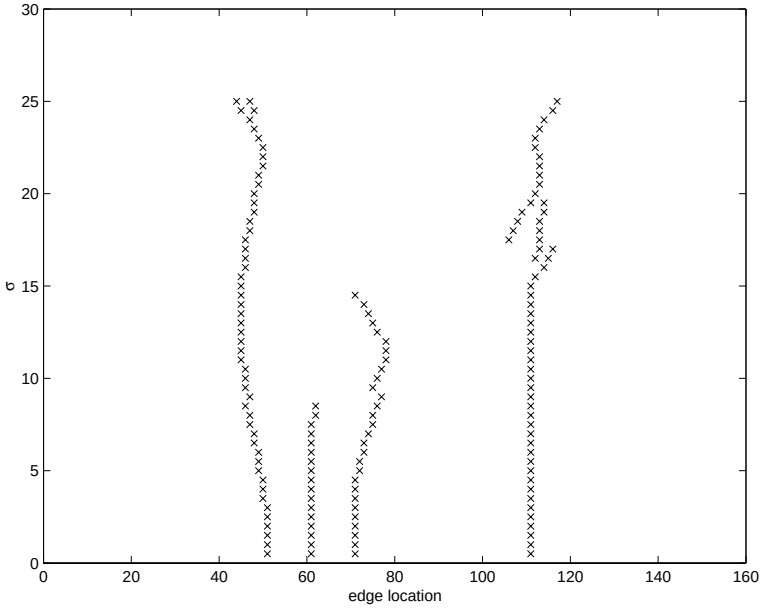

Figure 6: How edge location varies with σ

```
    e = find_edges(I,s,thresh);
    plot(e,s*ones(size(e)),'x')
    hold on
    n = n + 1;
end

axis([0 160 0 30])
xlabel('edge location')
ylabel('\sigma')

print -deps 2d
```

(e) Noise

Now we add noise to the input signal to see how robust our edge detection algorithm is. Figure 7 shows our input signal once white Gaussian noise has been added.

In this case, we must use a larger threshold otherwise we will detect many spurious edges due to the noise. Figure 8 shows how the detected edge locations vary with σ under the presence of noise. Note that at very low values of σ, many false edges are detected. Only in the approximate range of $6 < \sigma < 7$ do we get the 4 edges detected.

The source code for this section is from file *two_e.m*, listed below.

Listing 6: two_e.m

```
I = [zeros(1,50),9*ones(1,10),zeros(1,10),3*ones(1,40),zeros(1,50)];
NI = I + randn(size(I));

figure;
```

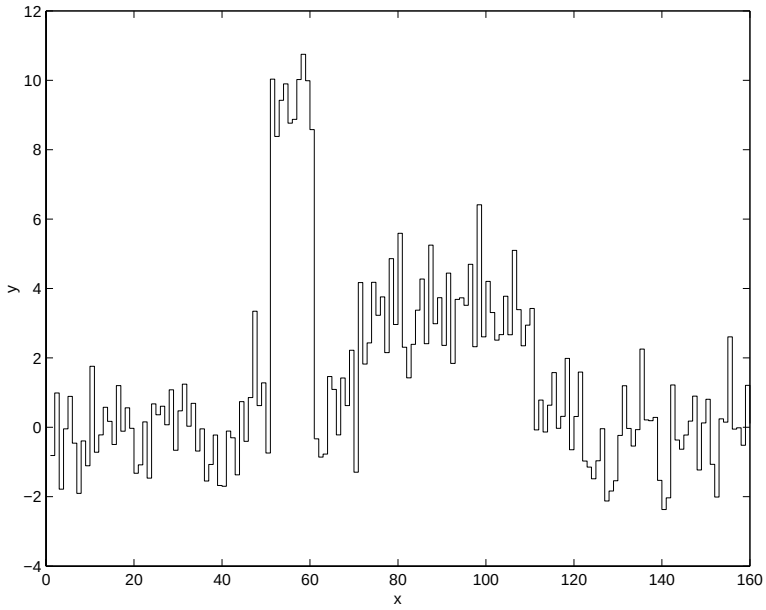

Figure 7: Signal corrupted with Gaussian noise

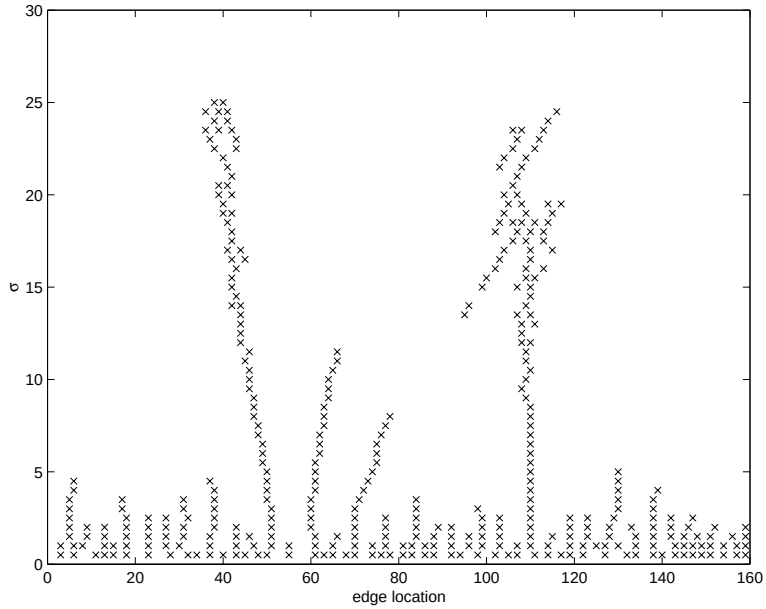

Figure 8: How edge location varies with σ, with Gaussian noise

```
stairs(NI)
xlabel('x')
ylabel('y')

print -deps noisy

thresh = .1;

n = 1;

figure

for s = .5:.5:25
  e = find_edges(NI,s,thresh);
  plot(e,s*ones(size(e)),'x')
  hold on
  n = n + 1;
end

axis([0 160 0 30])
xlabel('edge location')
ylabel('\sigma')

print -deps 2e
```

3 2D edge detection

We begin by trying to do edge detection on a relatively simple image. Figure 9 shows a photograph of the Maryland state flag. It should be easy to detect the boundaries of the shapes in this flag because they are so clearly delineated.

If we run a Canny edge detector algorithm on this image, with the default parameters, we get the result of Figure 10.

With the default parameters, the edge detector performs fairly well, but not optimally. We can improve performance by altering the parameters. Figure 11 shows the results with a better set of parameters ($thresh = [.095, .12], \sigma = .1$). The circled areas or those where the edge detector found a false edge. In both these areas, the flag is slightly wrinkled (i.e. the orientation of the surface changes rapidly). This rapid change in surface orientation results in sharp changes of lighting across these two patches of flag. This causes the detector to falsely classify the wrinkles as edges.

Figure 12 shows a picture of a camel in the desert. This is clearly a much more difficult problem. Both the camel's skin and the background are much more complex surfaces than the Maryland flag. Also, because of the camel's hair, the edges of the camel are not so well defined.

The Canny edge detector does very poorly with the default parameters. The results can be seen in Figure 13.

We can improve the performance greatly by altering the parameters. The best performance we could achieve is shown in Figure 14. The diagram shows two locations where the

Figure 9: Maryland flag

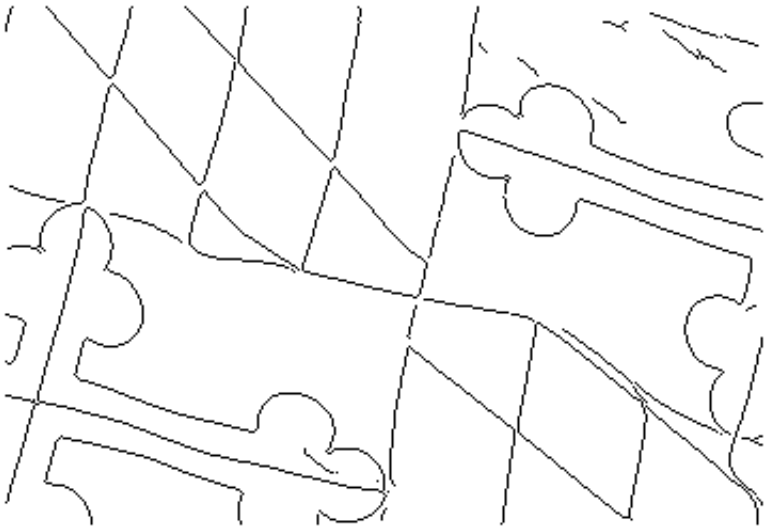

Figure 10: Canny edge detector, default parameters (thresh=$[.02, .1], \sigma = 1$)

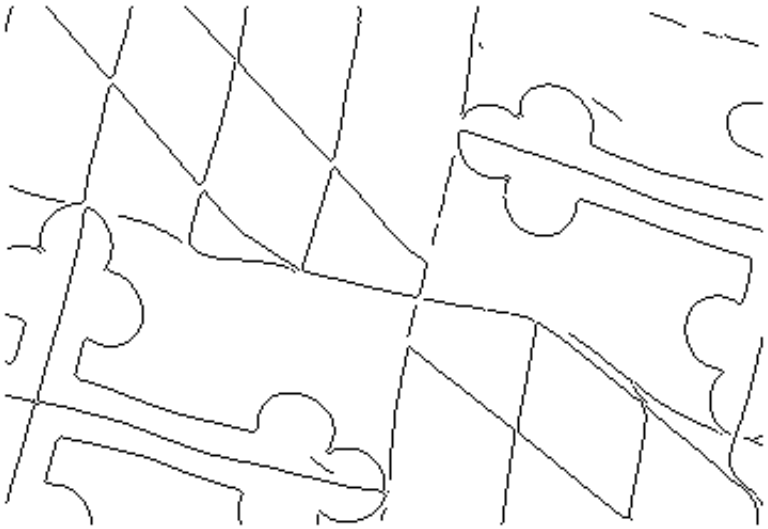

Figure 11: Canny edge detector, best results (thresh=$[.095, .12], \sigma = 1$)

Figure 12: Camel

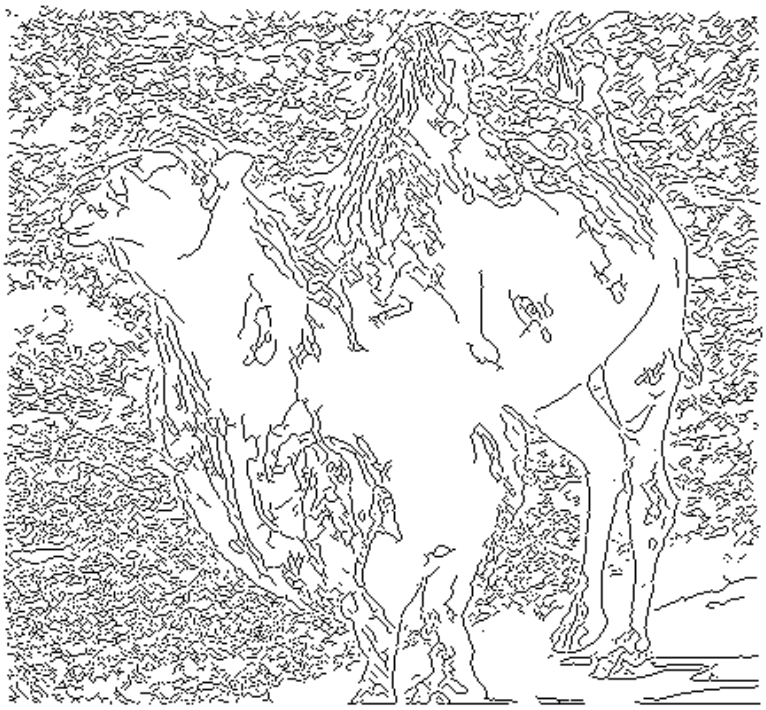

Figure 13: Canny edge detector, default parameters (thresh=$[.02, .1], \sigma = 1$)

edge detector made errors, although there are others. In the bottom-left area, the detector failed to locate an edge. If we look at the original picture, we see that there is not much difference in color intensity between the camel's hair and the road in this area. On the camel's hump, a spurious edge is detected. This seems to be due to the shadow cast by the camel's hump. When the shadow ends, there is an abrupt change in light intensity, and the detector falsely classfies this as an edge.

Figure 14: Canny edge detector, best results (thresh=$[.1, .35], \sigma = 3.5$)

4 Challenge problem

We will define the kernel k as centered around the point $\theta = \pi$, i.e.

$$k(\theta) = 1, |\pi - \theta| \leq m$$

If we repeat k periodically (with period 2π), then we can represent it as a linear combination of sinusoids. Note that such a function is even. Since sine functions are odd, and cosine functions are even, k can be expressed purely as a sum of cosine terms (plus a constant term), i.e.

$$k(\theta) = \sum_{n=0}^{\infty} a_n \left(\frac{\cos(n\theta)}{||\cos(n\theta)||} \right)$$

where $||\cos(n\theta)||$ is defined as follows:

$$||\cos(n\theta)|| = \sqrt{\langle \cos(n\theta), \cos(n\theta) \rangle} = \sqrt{\int_0^{2\pi} \cos^2(n\theta) d\theta}$$

$$= \sqrt{\frac{1}{2} \int_0^{2\pi} (1 + \cos(2n\theta)) d\theta} = \sqrt{\frac{1}{2} \left(\int_0^{2\pi} d\theta + \int_0^{2\pi} \cos(2n\theta) d\theta \right)}$$

$$= \sqrt{\frac{1}{2}(2\pi + 0)} = \sqrt{\pi}$$

For the case when n=0, we have:

$$\sqrt{\int_0^{2\pi} d\theta} = \sqrt{2\pi}$$

Note that the basis elements are orthogonal, i.e.

$$\langle \cos(n\theta), cos(m\theta) \rangle = 0, n \neq m$$

This implies that $\langle k(\theta), \frac{1}{\sqrt{\pi}} \cos(n\theta) \rangle = a_n$ which gives us a formula for computing a_n.

$$a_n = \int_0^{2\pi} k(\theta) \frac{\cos(n\theta)}{\sqrt{\pi}} d\theta = \frac{1}{\sqrt{\pi}} \int_{\pi-m}^{\pi+m} \cos(n\theta) d\theta = \left[\frac{1}{\sqrt{\pi} n} \sin(n\theta) \right]_{\pi-m}^{\pi+m}$$

$$= \frac{1}{\sqrt{\pi} n} \left[\sin(n\pi + nm) - \sin(n\pi - nm) \right]$$

$$= \frac{1}{\sqrt{\pi} n} \left[\sin(n\pi)\cos(nm) + \cos(n\pi)\sin(nm) - \sin(n\pi)\cos(nm) + \cos(n\pi)\sin(nm) \right]$$

$$= \frac{2}{\sqrt{\pi} n} \cos(n\pi)\sin(nm) = \frac{2}{\sqrt{\pi} n} (-1)^n \sin(nm)$$

For a_0, we have:

$$a_0 = \frac{1}{\sqrt{2\pi}} \int_0^{2\pi} k(\theta) d\theta = \frac{1}{\sqrt{2\pi}} \int_{\pi-m}^{\pi+m} d\theta = \frac{2m}{\sqrt{2\pi}}$$

Therefore, we can represent the kernel as:

$$k(\theta) = \frac{2m}{\sqrt{2\pi}} + \sum_{n=1}^{\infty} \frac{2}{\sqrt{\pi} n} (-1)^n \sin(nm) \left(\frac{\cos(n\theta)}{\sqrt{\pi}} \right)$$