9.35 Sensation And Perception

Spring 2009

Problem Set 2 Solutions Guide

9.35 Spring 2009

Contents

- Init
- 1
- 2
- 3
- 4

Init

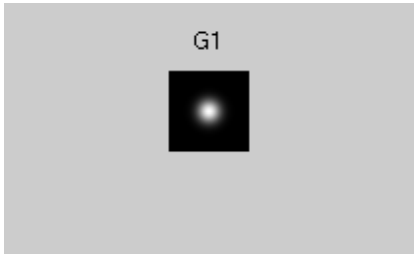
```
clear all
close all
clc
drawnow
```

1

```
[X Y] = meshgrid(-20:20);

GRF = @(sigma) exp(-.5*(X.^2 + Y.^2)/sigma^2);
```

1a)

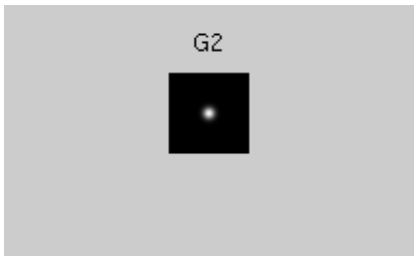
```
G1 = GRF(4);
G1 = G1/sum(G1(:));

figure
imshow(G1, [])
title('G1')
```



The intensity of each pixel in the filter is the weight of the receptive field at that point. If we assume that each cell projects to the nearest downstream cells (i.e., the cells right below it) with some spread, over a couple of projections, the probability of an RGC receiving input from a photoreceptor will look like a Gaussian.

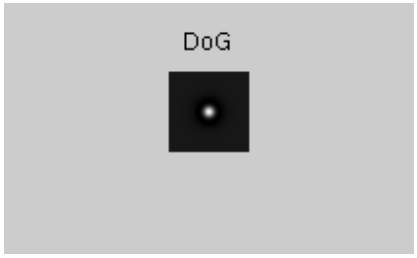
1b)

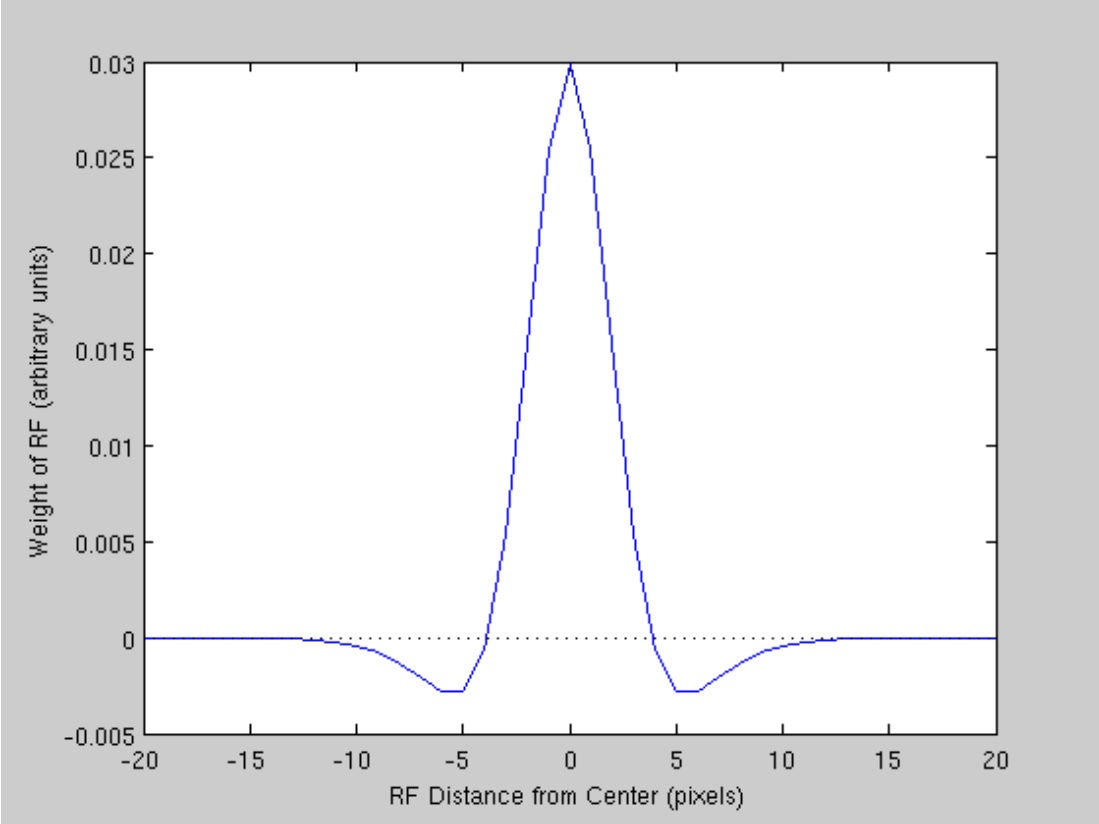
```
G2 = GRF(2);
G2 = G2/sum(G2(:));
figure
imshow(G2, [])
title('G2')
```



G1 is more sensitive to light, since it pools its response over a larger area

```
DoG = G2-G1;
figure
imshow(DoG, []); title('DoG')

figure
plot(-20:20, DoG(X==0)); xlabel('RF Distance from Center (pixels)')
ylabel('Weight of RF (arbitrary units)')
hold on
plot(-20:20, zeros(41,1), 'k:')
hold off
```





2

2a)

```
ZeroZero = 1*G1(X==0 & Y==0)         % Intensity 1 times receptive field
PlusTwoThree = ZeroZero + 1*G1(X==2 & Y==3)
WholeField = sum(sum(ones(size(G1)).*G1))   % Since we normalized, === 1!!
```

```
ZeroZero =

    0.0099


PlusTwoThree =

    0.0166


WholeField =

    1.0000
```

2b)

```
im = double(imread('peppers.png'))/255;
im = im(:, :, 2);
```

```
imshow(im);
Output = sum(sum(DoG.*im(105:145, 105:145)))
```

```
Output =

   -0.0242
```

2c) We use convolution here because it is the equivalent to calculating the above output at every single pixel in the image. So, it will repeat the integration above for every RF position, and enable us to model a population of cells viewing this image.

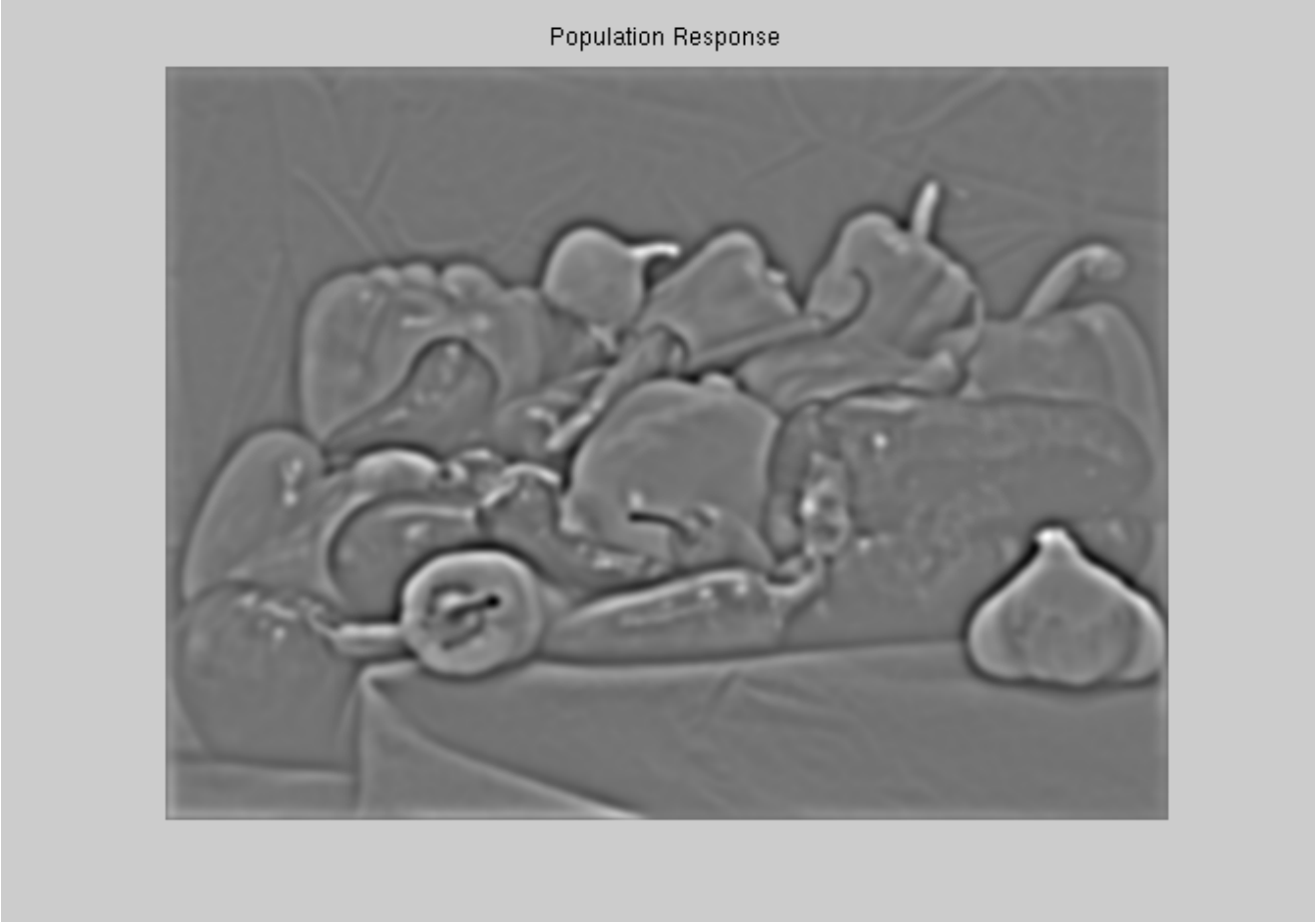
To implement the algorithm, we would:

- For every pixel, looping across all rows and columns:
- . Extract a local neighborhood centered on the current pixel
- . Multiply that neighborhood by our filter
- . Sum (integrate) the product
- . Store the result in a new image, in the same location as our current pixel

The tricky part is that near the edges of the image, the local neighborhood (which is the size of the filter we've defined) extends past the image border - we would need to find a clever way to avoid referencing non-existent pixels! So instead of implementing this ourselves, we just use `filter2`

2d)

```
figure
imshow(filter2(DoG, im), []); title('Population Response')
```

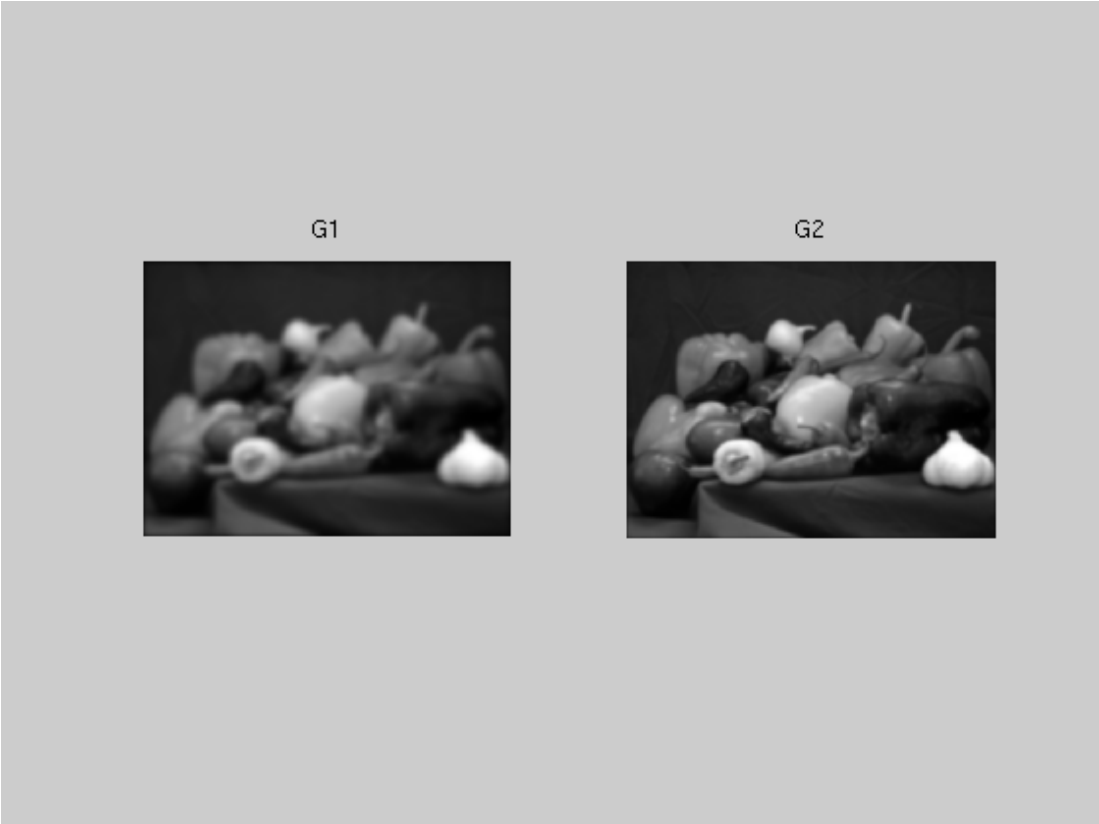


Negative values can mean one of two things. Either this cell is inhibited (firing rate at or near zero), or we can imagine an OFF RGC with a similar RF structure, at this location, which is excited.

It should be readilly apparent that this RF likes lines and edges in the image, which are approximately the same size as the center region. To be more selective for higher frequencies, we simply need to shrink the filter (for example, by decreasing sigma).

2e)

```
figure
subplot(1,2, 1)
imshow(filter2(G1, im), []); title('G1')
subplot(1,2,2)
imshow(filter2(G2, im), []); title('G2')
```

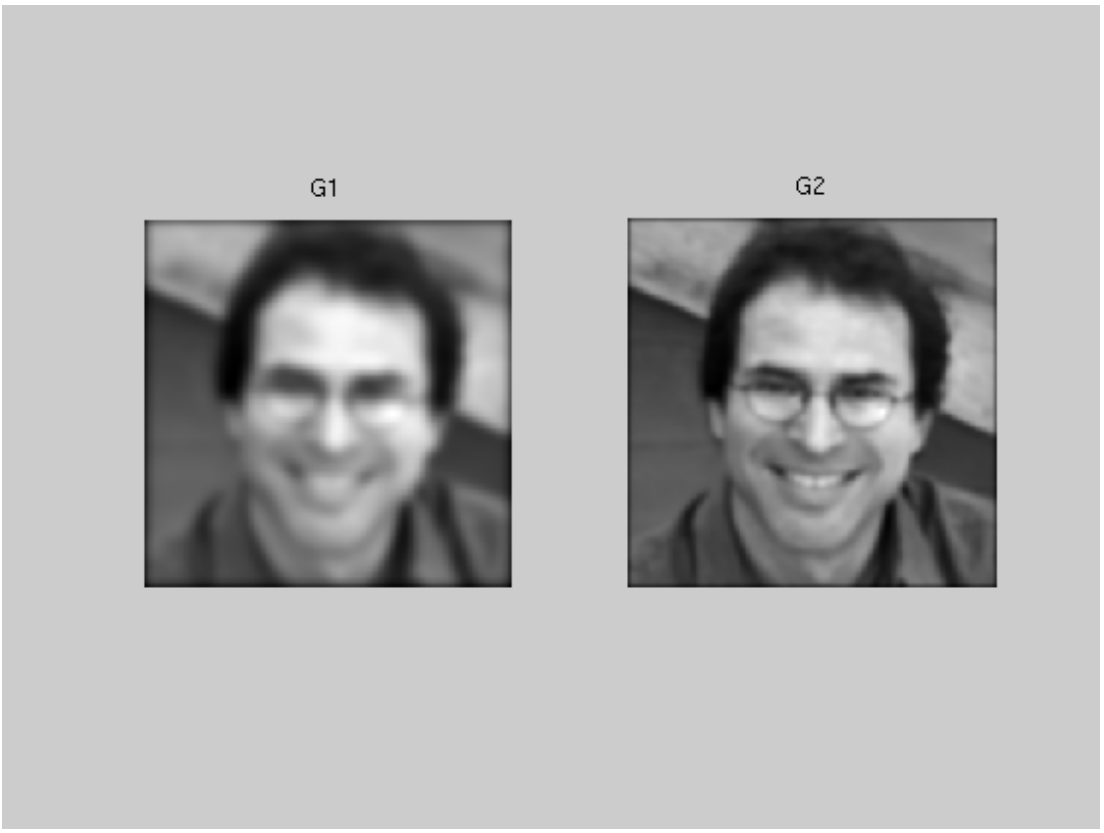


The difference is simply how large a region the Gaussians are blurring over (each of them is a low-pass filter). We normalize the integral to 1 because this type of filter is a local average - so it's important to maintain the local mean. If the integral were not 1, the overall intensity of the output would be mutliplied by a factor equal to the integral. On their own this would impose a gain on the cell; more importantly, if the two components of the DoG were not normalized, then the overall DoG integral would not be 0. This would give the DoG an ambient light response, which is contrary to our understanding of the retina.

2f)

```
adelson = imread('adelson.tiff');
figure
imshow(adelson);
figure
imshow(filter2(DoG, adelson), []); colormap gray; axis image; axis off;
figure
subplot(1,2, 1)
```

```
imshow(filter2(G1, adelson), []); title('G1')
subplot(1,2,2)
imshow(filter2(G2, adelson), []); title('G2')
```



3

3a)

```
grating = @(f, phi) sin(2*pi*f*(X*cos(phi)+Y*sin(phi)));

figure
imshow(grating(4/41,0), []); title('Grating')
```

3b)

```
PixelsPerInch = 300;
CyclesPerPixel = 4/41;
CyclesPerInch = CyclesPerPixel*PixelsPerInch;

% If you draw a triangle, this is pretty easy
InchesPerDegree = 2*36*tand(.5); % Inches which subtend 1 degree at 36"
```

```
CyclesPerDegree = CyclesPerInch*InchesPerDegree

% Now set CPD and see where we stand
CyclesPerDegree = 10;
InchesPerDegree = CyclesPerDegree/(PixelsPerInch * CyclesPerPixel);
ViewingDistance = 0.5*InchesPerDegree/tand(.5) % In inches

% Set distance and CPD, find wavelength
ViewingDistance = 4*12;
InchesPerDegree = 2*ViewingDistance*tand(.5);
PixelsPerCycle = PixelsPerInch*InchesPerDegree/CyclesPerDegree
```

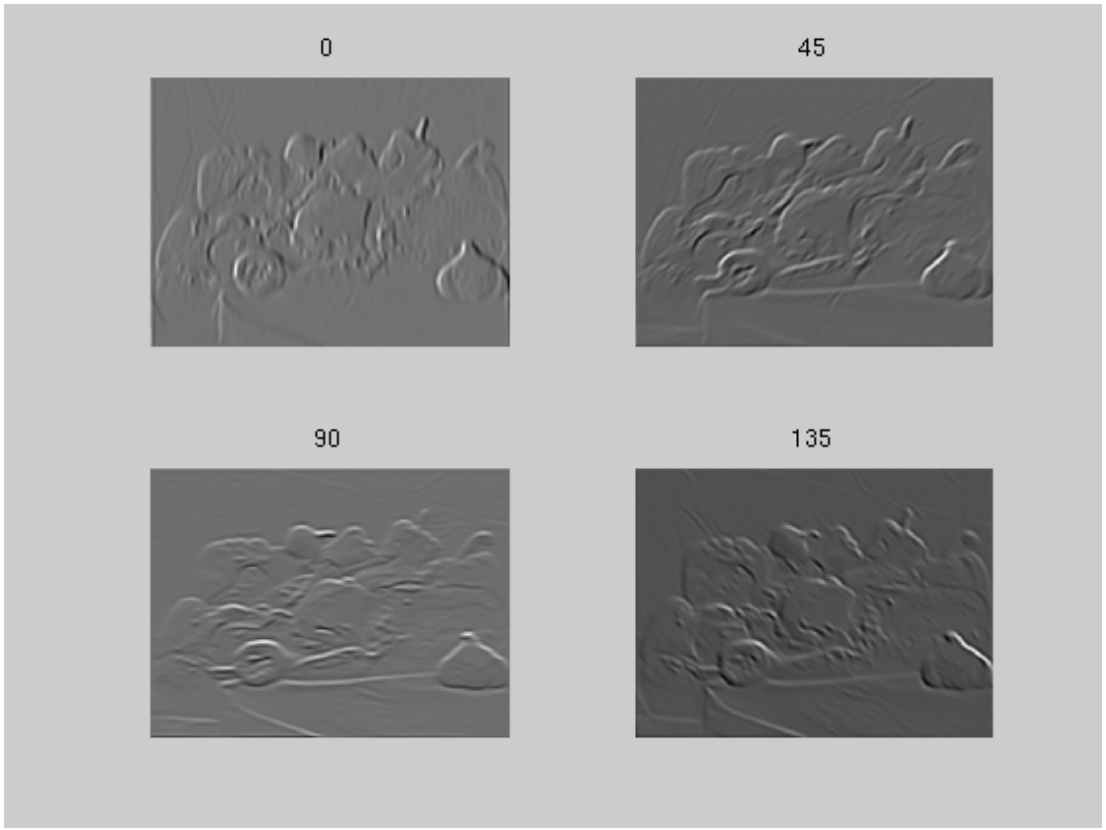
```
CyclesPerDegree =

   18.3903


ViewingDistance =

   19.5756


PixelsPerCycle =

   25.1334
```

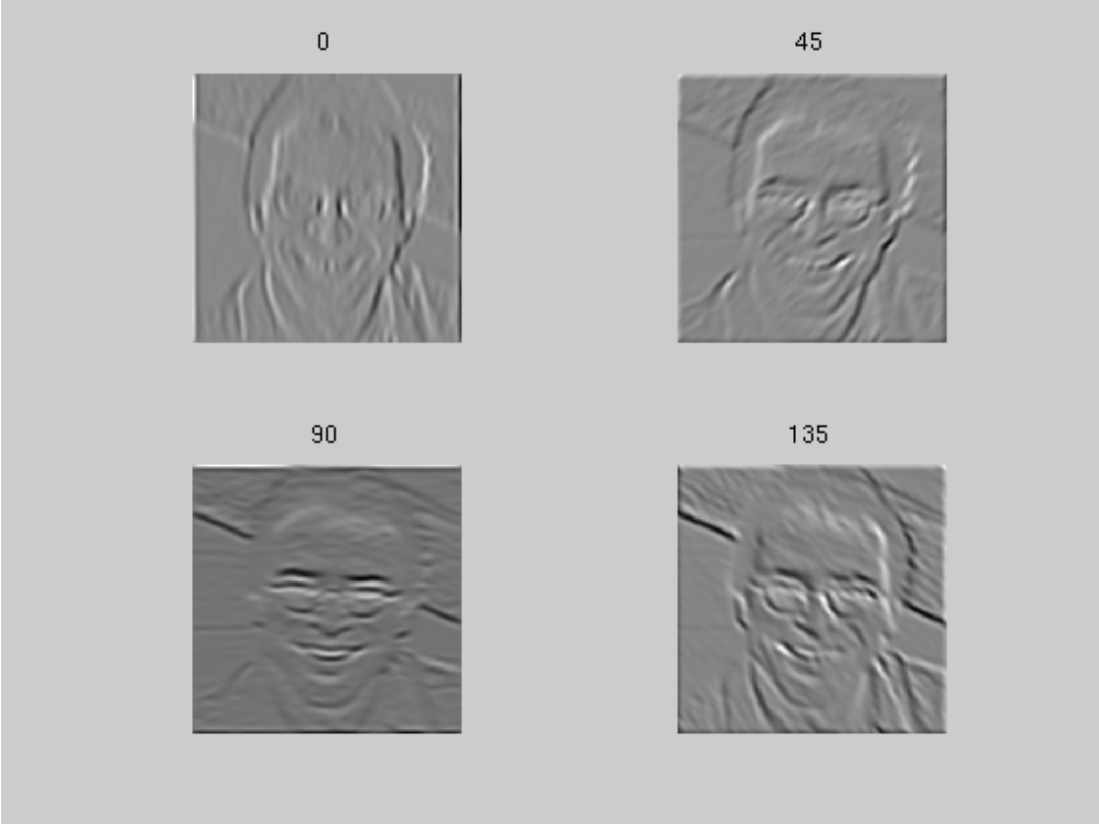
3c)

```
f = 4/41;
gab0 = G1.*grating(f, 0);
err = sum(gab0(:)) % Should be close to 0, so we won't bother normalizing
figure
imshow(gab0, [])
title('Gabor at 0 degrees')
phi = pi*(0:3)/4;
phiName = {'0', '45', '90', '135'};
images = {im, adelson};
for imidex = 1:2
    figure
    for i = 1:4
        subplot(2, 2, i)
        imshow(filter2(G1.*grating(f, phi(i)), images{imidex}), []);
        title(phiName{i})
    end
end
```

```
err =

   3.6038e-17
```


Gabor at 0 degrees


0
45
90
135



3d) Since these are *linear* models, they obviously do not explain the nonlinearities found in the visual system. Some good examples of this are many temporal dynamics (like refractory periods), other types of selectivity (like motion/direction selectivity), the cell's history (like if it's fatigued or not), any interactions it may have with it's neighbors (like lateral inhibition), physical limitations on the cell's firing rate (like it can't go below 0 or above 1/refractory period spikes per second), nonlinear gains, etc.

4

I spent approximately 20 minutes coding this and 2 hours proofing for your reading pleasure.

```
close all
```

Published with MATLAB® 7.6