Dear EECS 556 Class – Thank you for your and participation in class. Please accept my best wishes for an enjoyable summer. – Doug Noll

Final Exam Scores: Median = 91, 1st quartile = 94, 3rd quartile = 85
Overall Grade Ranges: 97 and up: A+; 91-96: A; 85-90: A-; 80-84: B+; 79 and below: B

1. Optimal deblurring and denoising.
   a. The superposition principle states:

$$f_{blur}(n,m) = S[f(n,m)] = S[f(n,m) ** d(n,m)] = S\left[\sum_k \sum_l f(k,l)d(n-k,m-l)\right]$$
$$= \sum_k \sum_l f(k,l)S[d(n-k,m-l)] = \sum_k \sum_l f(k,l)h(n,m;k,l)$$

   We now column stack all quantities in (*n*,*m*) and we'll get:

$$f_{blur}(p) = \sum_k \sum_l f(k,l)h(p;k,l)$$

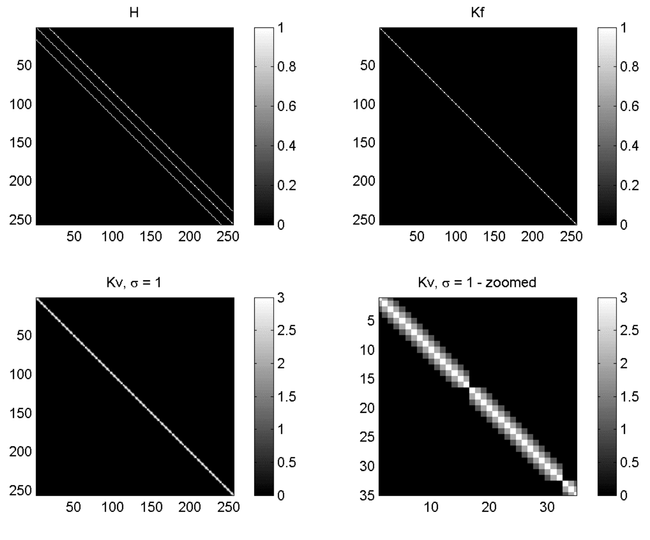
   and by letting $q = k*M+1$ (sorry – typo in the problem statement – should have been *M* not *N*), we get:

$$f_{blur}(p) = \sum_q f(q)h(p;q)$$

   which is just the matrix multiplication **Hf** where the elements of **H** are *h*(*p*;*q*), where *q* is the column index and *p* is the row index.

   b. I did the "brute force" method, but there are lots of more elegant ways. See code.

   c. *f* is a zero-mean, unit variance point Bernoulli process: $R_f(n,m) = d(n,m)$, so $\mathbf{K}_f$ is the identity matrix. *v* is a white Gaussian process convolved by a rect function: $R_v(n,m) = d(n)tri_3(m)$. See code for implementation of $\mathbf{K}_v$.

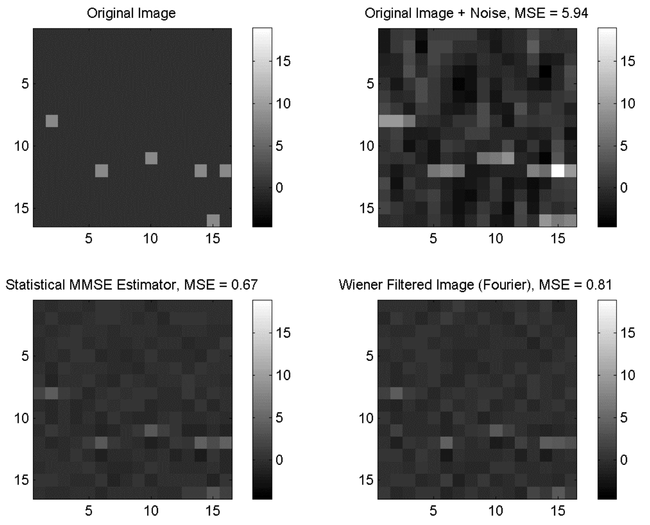


   d. We implement $\hat{\mathbf{f}} = (\mathbf{H}'\mathbf{K}_v^{-1}\mathbf{H} + \mathbf{K}_f^{-1})^{-1}\mathbf{H}'\mathbf{K}_v^{-1}\mathbf{g}$. See code.

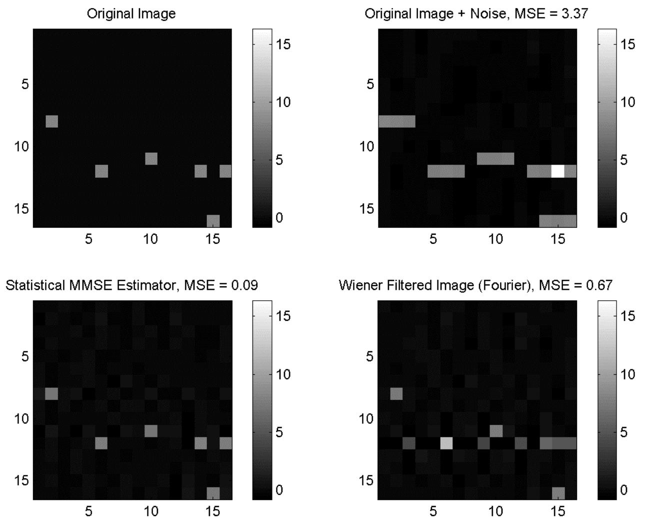
   e. $B(w_x, w_y) = 1 + 2\cos w_x$

f. $P_f(w_x, w_y) = 1$ and $P_v(w_x, w_y) = (nsd)^2 (1 + 2\cos w_y)^2$

g. $H(w_x, w_y) = \dfrac{P_f(w_x, w_y) B(w_x, w_y)}{P_f(w_x, w_y) B^2(w_x, w_y) + P_v(w_x, w_y)}$. See code for implementation and plots for results. MSE for Part d. is better, but not by much.



h. After reducing the noise variance, we can see more substantial differences between the standard (Fourier) Wiener filter and the statistical MMSE estimator.



At low noise levels, the filter is doing mainly deblurring. The statistical MMSE estimator is better because it properly models the spatially variant behavior at the edges of the object. The Wiener filter assumes circulant behavior or zeros in the case of zero padding – both assumptions are inaccurate.

Code:

```
% eecs 556, 2003, final exam, problem 1
rand('state',0);randn('state',0);
nx = 16; ny = 16; nsd = 1;
```

```
p=0.015;
ff = ((rand(ny,nx) < p) - p)/sqrt(p*(1-p)); % the desired signal
vv = nsd*randn(ny,nx);
% add this
vv = conv2(vv,ones([3 1]),'same');     % the noise
blurff = conv2(ff,ones([1 3]),'same'); % the blurred signal
gg = blurff + vv;                      % the measurement
g1 = gg(:);                            % make image a vector

% build burring matrix
H = zeros([nx*ny nx*ny]);
for lpx = 1:nx
    for lpy = 1:ny
        deltaim = zeros([nx ny]);
        deltaim(lpx,lpy) = 1;
        h = conv2(deltaim,ones([1 3]),'same');
        % The follow line will produce a circulant matrix with identical
        % results as the Fourier method (if we also use a circulant Kv)
        % h = real(ifft2(fftshift(fftshift(fft2(deltaim)).*B)));
        matxind = lpx + (lpy-1)*nx;
        H(:,matxind) = h(:);
    end
end

% build arrays to calculate covariance matrix
[xx yy] = ndgrid([-nx/2:nx/2-1],[-ny/2:ny/2-1]);
x1 = xx(:);
[x1a x1b] = ndgrid(x1);
% the following array contains the difference between
% the x-coordinates for the covariance elements
xinddiff = (x1a - x1b);

y1 = yy(:);
[y1a y1b] = ndgrid(y1);
% the following array contains the difference between
% the x-coordinates for the covariance elements
yinddiff = (y1a - y1b);

% now calculate the covariance matrices
% since the autocorrelation function is separable, we can do our
% calculations on x and y to build the final covariance matrix
kvx = (3-abs(xinddiff));
kvx = kvx.*(kvx > 0);
kvy = (yinddiff == 0);
kv = nsd^2*kvx.*kvy;
kv2 = nsd^2*eye(nx*ny);
kf = eye(nx*ny);

fhat1d = inv(H'*inv(kv)*H+inv(kf))*H'*inv(kv)*g1;
fhat = reshape(fhat1d,size(ff));

Pf = ones(nx,ny);
wx = [-nx/2:nx/2-1]./nx*2*pi;
Pv = nsd^2*((1 + 2*cos(wx)).^2)'*ones(1,nx);
Pv2 = nsd^2*ones(nx,ny);
B = ones(ny,1)*(1 + 2*cos(wx));
Hw = Pf.*B./(Pf.*(B.^2) + Pv);
fhat2 = real(ifft2(fftshift(fftshift(fft2(gg)).*Hw)));

mseor = mean(mean((gg-ff).^2));
msew = mean(mean((fhat-ff).^2));
msew2 = mean(mean((fhat2-ff).^2));

%diplay filtered results
figure(1)
clim = [min(min(gg)) max(max(gg))];
subplot(221); imagesc(ff,clim); colormap gray; colorbar; axis('image')
title('Original Image')
subplot(222); imagesc(gg,clim); colormap gray; colorbar; axis('image')
title(sprintf('Original Image + Noise, MSE = %.2f',mseor));
subplot(223); imagesc(fhat,clim); colormap gray; colorbar; axis('image')
```

```
title(sprintf('Statistical MMSE Estimator, MSE = %.2f',msew));
subplot(224); imagesc(fhat2,clim); colormap gray; colorbar; axis('image')
title(sprintf('Wiener Filtered Image (Fourier), MSE = %.2f',msew2));

figure(2)
subplot(221); imagesc(H); colormap gray; colorbar; axis('image'); title('H')
subplot(222); imagesc(kf); colormap gray; colorbar; axis('image'); title('Kf')
subplot(223); imagesc(kv); colormap gray; colorbar; axis('image'); title('Kv, \sigma =
1')
subplot(224); imagesc(kv); colormap gray; colorbar; axis('image');
axis([1 35 1 35]); title('Kv, \sigma = 1 - zoomed')
```

2. Pyramid coding.

a. See code.

b. See code.

c. Based on variance measures and on the fact that there are 16 times as many e0 pixels than f2 pixels, one might expect to assign 1.2668 for e0 and 3.7173 for f2. Without variable length codewords, we stand no chance of meeting our desired bit rate of 1.5, but we may if the probabilities levels of e0 are very non-uniformly distributed and we use a variable length code. It also looks like we are giving few levels than we'd like to give for f2 – $2^{3.7173} \approx 13$, if we were to use a uniform length code, but in general, our allocation scheme seems to be reasonable based on the variance measures.

d. Huffman code for 8 levels {110, 10, 00, 1110, 11110, 111111, 111110, 01}. Bit rate = 2.5420 bits/symbol.

e. $H = 2.5154$. Our code came very close to meeting the entropy rate. The uniform bit rate would be 3 bit/symbol. There is some savings (15%), but not a huge savings.

f. See code.

g. See code.

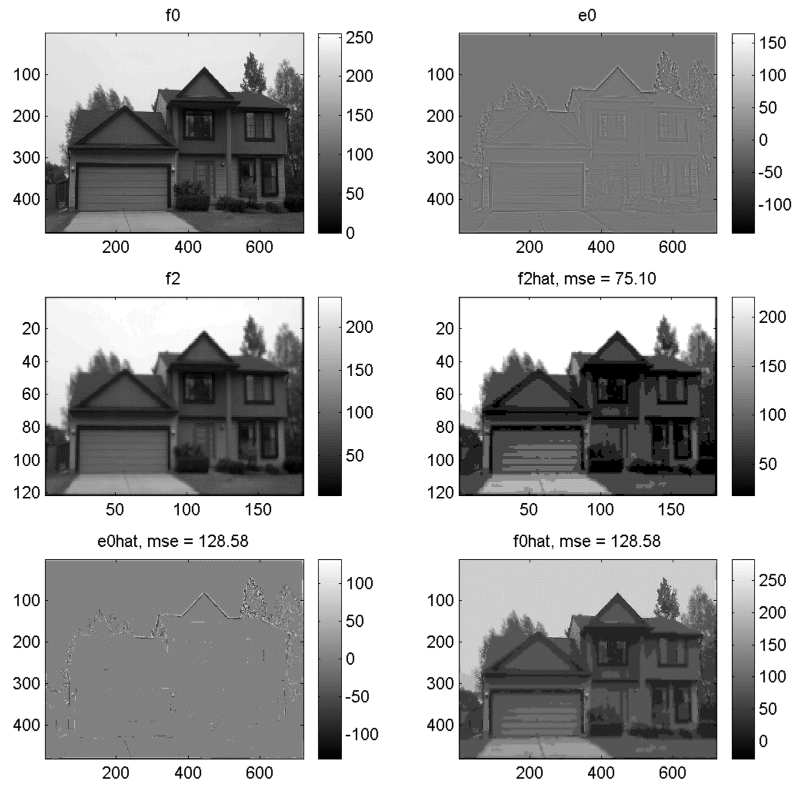
h. See code.

i. Huffman code for 5 levels {0000, 001, 1, 01, 0001}. Bit rate = 1.0573 bits/symbol.

j. $D = 128.58$. Overall bit rate = 1.2178 bits/symbol – we've met our target. This approach was better than PCM, not as good as VQ (and we hadn't even used variable length coding with VQ).

k. Interestingly, $D = MSE_0$. $MSE_2 = 74.10$. There is apparently a rather complicated relationship between MSE_0 and MSE_2 and the final distortion. In particular, the quantization errors of the higher stages only have an indirect effect on the final distortion. Shannon's bit allocation scheme was derived from sum of error from each quantization level – this clearly does not apply to pyramid coding and thus, another system might be better (though Shannon's rule isn't a bad starting point).

Images:



Code:

```
% eecs 556, 2003, final exam, problem 2
load hw7image;
f0 = hw7image;
f0(end+1,end+1)=0;
% make filter
a = .4;
h1 = [(.25-a/2) .25 a .25 (.25-a/2)];
h = h1'*h1;
f0l = conv2(f0,h,'same');
f1 = f0l(1:2:end,1:2:end);
f1l = conv2(f1,h,'same');
f2 = f1l(1:2:end,1:2:end);

% linear interp kernel
h2 = [1 2 3 4 3 2 1]/4;
hh = h2'*h2;
% build estimate
f0lhat = zeros(size(f0));
f0lhat(1:4:end,1:4:end) = f2;
f0lhat = conv2(f0lhat,hh,'same');
e0 = f0 - f0lhat;

subplot(421); imagesc(f0); colormap(gray); colorbar; title('f0');
subplot(422); imagesc(e0); colormap(gray); colorbar; title('e0');
subplot(423); imagesc(f2); colormap(gray); colorbar; title('f2');

% variances and bit rates
ve0 = var(e0(:));
vf2 = var(f2(:));
vm = (ve0^(16)*vf2).^(1/(16+1));
tmpbpp = 1.5*prod(size(f0))/(prod(size(e0))+prod(size(f2)));
be0 = (tmpbpp + 0.5*log2(ve0/vm))
```

```
bf2 = (tmpbpp + 0.5*log2(vf2/vm))

% quantize f2
lf2 = 8;
mnf = min(f2(:)); mxf = max(f2(:)); deltf = (mxf-mnf)/lf2+0.01;
levelf2 = floor((f2-mnf)/deltf);
f2hat = levelf2*deltf + deltf/2+mnf;
msef2 = mean((f2(:)-f2hat(:)).^2);
subplot(424); imagesc(f2hat); colormap(gray); colorbar;
title(sprintf('f2hat, mse = %.2f',msef2));

% build estimate
f0lhat2 = zeros(size(f0));
f0lhat2(1:4:end,1:4:end) = f2hat;
f0lhat2 = conv2(f0lhat2,hh,'same');
e02 = f0 - f0lhat2;

% quantize e0
le0 = 5;
mxf = max(abs(e02(:))); mnf = -mxf; deltf = (mxf-mnf)/le0+0.01;
levele0 = floor((e02-mnf)/deltf);
e0hat = levele0*deltf + deltf/2+mnf;
msee0 = mean((e02(:)-e0hat(:)).^2);
subplot(425); imagesc(e0hat); colormap(gray); colorbar;
title(sprintf('e0hat, mse = %.2f',msee0));

% final estimate
f0hat = f0lhat2 + e0hat;
msef2 = mean((f0(:)-f0hat(:)).^2);
subplot(426); imagesc(f0hat); colormap(gray); colorbar;
title(sprintf('f0hat, mse = %.2f',msef2));

% find probs, entropy and bit rate
for lp=1:lf2
    pr(lp) = sum(levelf2(:) == (lp-1))./prod(size(f2));
end
Hf2 = -sum(pr.*log2(pr))
lenf2 = [3 2 2 4 5 6 6 2];
Bbarf2 = sum(pr.*lenf2)

for lp=1:le0
    pre(lp) = sum(levele0(:) == (lp-1))./prod(size(e0));
end
He0 = -sum(pre.*log2(pre))
lene0 = [4 3 1 2 4];
Bbare0 = sum(pre.*lene0)

Bbartot = (Bbare0*prod(size(e0hat))+Bbarf2*prod(size(f2hat)))/prod(size(f0hat))
```

3. Coordinate transformations, Fourier transforms and sampling.

a. First, recognize that $\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{R}_q \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_r \\ y_r \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix}$, where $\mathbf{R}_q = \begin{bmatrix} \cos q & \sin q \\ -\sin q & \cos q \end{bmatrix}$,

and x_r and y_r are rotated coodinates (e.g. shifts are in the rotated frame). We know that rotations of the image domain lead to rotations in the Fourier domain and shifts in the image domain lead to phase shifts in the Fourier domain. Thus,

$$G(u,v) = F(u\cos q + v\sin q, v\cos q - u\sin q)\exp(i2p\,(a(u\cos q + v\sin)) + v(v\cos q - u\sin q)))$$

or $G(\mathbf{u}) = F(\mathbf{R}_q\mathbf{u})\exp(i2p\begin{bmatrix} a & b \end{bmatrix}\mathbf{R}_q\mathbf{u})$.

b. $\mathbf{T}(q', a', b') = \mathbf{T}^{-1}(q, a, b) = \begin{bmatrix} \cos q & -\sin q & (-a\cos q + b\sin q) \\ \sin q & \cos q & (-a\sin q - b\cos q) \\ 0 & 0 & 1 \end{bmatrix}$, which for

$q' = -q = -45°$, $a' = -0.3536 \neq -a$, and $b' = 0.0707 \neq -b$. In the transformation

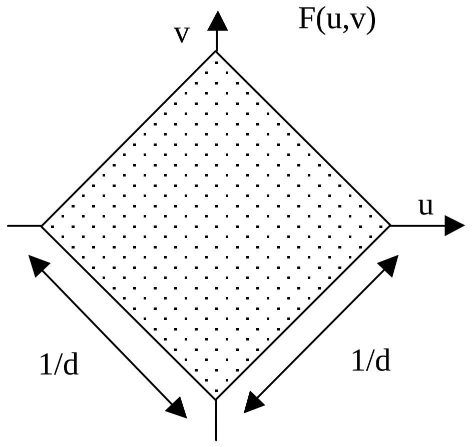
$\mathbf{T}(q, a, b)$, the rotation happens first followed by translation. One inverse would be to translate by *–a* and *–b* the rotate by *–q*. $a' \neq a$ and $b' \neq b$ because in implementing as another matrix in the form **T**, we do the operations as rotation then translation, so the inverse translations have to be projected through the *–q* rotation matrix:

$$\mathbf{R}_{-q}\begin{bmatrix} x-a \\ y-b \end{bmatrix} = \mathbf{R}_{-q}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \cos q & -\sin q \\ \sin q & \cos q \end{bmatrix}\begin{bmatrix} -a \\ -b \end{bmatrix} = \mathbf{R}_{-q}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -a\cos q + b\sin q \\ -a\sin q - b\cos q \end{bmatrix}$$

c. We know from sampling theory that $F_S(u,v) = \sum_k \sum_l F(u - \frac{k}{d}, v - \frac{l}{d})$ and

$$G_s(u,v) = \sum_k \sum_l G(u - \tfrac{k}{d}, v - \tfrac{l}{d})$$
$$= \sum_k \sum_l F(\mathbf{R}_q \begin{bmatrix} u - \frac{k}{d} \\ v - \frac{l}{d} \end{bmatrix}) \exp(i2p\,[a \quad b]\mathbf{R}_q \begin{bmatrix} u - \frac{k}{d} \\ v - \frac{l}{d} \end{bmatrix})$$

d. To reconstruct the *f* from g_s, we first can consider that the relationship between *f* and *g* is known and invertible. Thus, we can consider what properties of *g* can be reconstructed and relate those back to *f*. *g* can be reconstructed from g_s if *G*(*u*,*v*) is non-zero only for $|u| < \frac{1}{2d}$ and $|v| < \frac{1}{2d}$. Our conditions on *f* are then that *F*(*u*,*v*) must be non-zero only for $|u\cos q - v\sin q| < \frac{1}{2d}$ and $|v\cos q + u\sin q| < \frac{1}{2d}$, as shown below. This particular distribution does not allow reconstruction from f_s. To allow reconstruction from both g_s and f_s we would need to add these conditions: $|u| < \frac{1}{2d}$ and $|v| < \frac{1}{2d}$ (an octagonal shaped region). To handle an arbitrary angle *q*, the condition is $\sqrt{u^2 + v^2} < \frac{1}{2d}$.



4. Wavelet transforms.

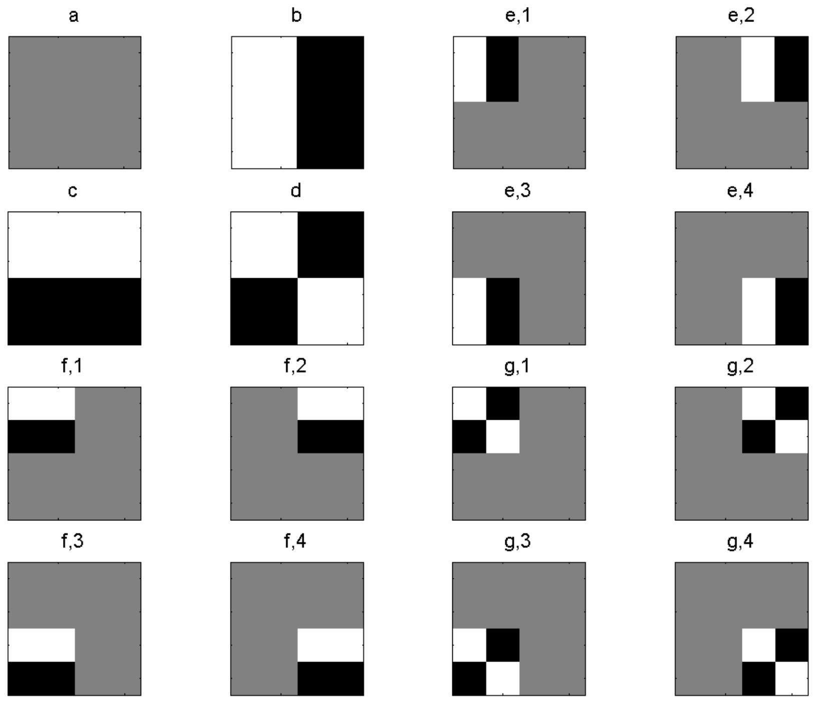
a. We can quickly see that $\mathbf{A}^T\mathbf{A} = \mathbf{I}$, therefore, **A** is a unitary or orthogonal matrix, which has orthogonal rows and is energy preserving.

b. See code.

c. This is not energy preserving – observe that this is approximately a 2x2 2D Haar transform with basis functions [1 1] and [1 –1]. The energy preserving basis functions are $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\end{bmatrix}$ and $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & -1\end{bmatrix}$. Since we are doing the transform in both x and y, the output of haar2.m must be divided by 2. See haar2new.m below.

d. Basis functions, by region:
   a. $a_x(0)$ and $a_y(0)$
   b. $a_x(1)$ and $a_y(0)$
   c. $a_x(0)$ and $a_y(1)$
   d. $a_x(1)$ and $a_y(1)$
   e. $a_x(2)$ and $b_y(0)$, $a_x(3)$ and $b_y(0)$, $a_x(2)$ and $b_y(1)$, $a_x(3)$ and $b_y(1)$
   f. $a_y(2)$ and $b_x(0)$, $a_y(3)$ and $b_x(0)$, $a_y(2)$ and $b_x(1)$, $a_y(3)$ and $b_x(1)$
   g. $a_x(2)$ and $a_y(2)$, $a_x(3)$ and $a_y(2)$, $a_x(2)$ and $a_y(3)$, $a_x(3)$ and $a_y(3)$

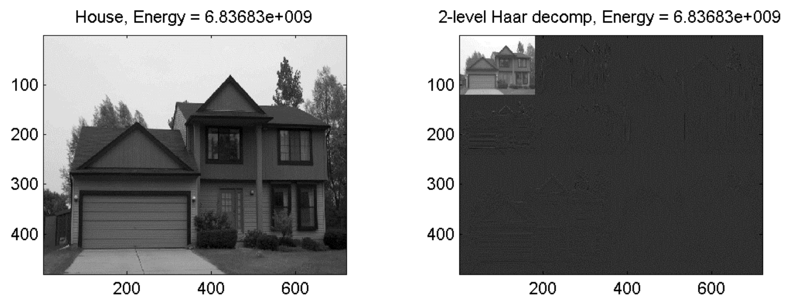
   where $b_d(0) = \begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0\end{bmatrix}$ and $b_d(1) = \begin{bmatrix}0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\end{bmatrix}$. We can see these visually in the following figure. This was generated by running "ihaar2.m" from part f. on a 4x4 delta function image with the delta function moved to all 16 possible locations specified by a-g. This is close, but not exactly a 4x4 2D Haar transform.

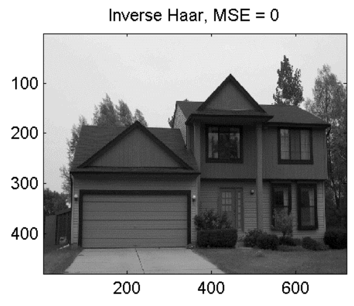


e. They are all orthogonal, as can be readily seen from the above figure. It is true that the *b* bases are not orthogonal with $a_d(0)$ and $a_d(1)$, but the *b* bases are always in combination with *a* bases in the orthogonal direction so the 2D bases are orthogonal.

f. "haar2new.m" implements a 2x2 2D Haar transform. It turns out that the inverse transform is exactly the same for the 2x2 case, e.g. $\mathbf{A}^T=\mathbf{A}$. The real challenge here is just the reordering of the transformed image pixels. See code and "ihaar2.m".

Images:


House, Energy = 6.83683e+009
2-level Haar decomp, Energy = 6.83683e+009
100
200
300
400
200
400
600



Code:

```
% 2D wavelet problem
load hw7image;
h1 = haar2new(hw7image);
[a b] = size(h1);
h1(1:a/2,1:b/2) = haar2new(h1(1:a/2,1:b/2));

subplot(221); imagesc(hw7image); colormap gray;
title(sprintf('House, Energy = %g',sum(hw7image(:).^2)));
subplot(222); imagesc(h1); colormap gray;
title(sprintf('2-level Haar decomp, Energy = %g',sum(h1(:).^2)));

h2 = h1;
h2(1:a/2,1:b/2) = ihaar2(h1(1:a/2,1:b/2));
h2 = ihaar2(h2);
subplot(223); imagesc(h2); colormap gray;
title(sprintf('Inverse Haar, MSE = %g',sum((hw7image(:)-h2(:)).^2)));


function imout = haar2(im)
subimLL = (im(1:2:end,1:2:end) + im(2:2:end,1:2:end) + im(1:2:end,2:2:end) +
im(2:2:end,2:2:end));
subimHL = (im(1:2:end,1:2:end) - im(2:2:end,1:2:end) + im(1:2:end,2:2:end) -
im(2:2:end,2:2:end));
subimLH = (im(1:2:end,1:2:end) + im(2:2:end,1:2:end) - im(1:2:end,2:2:end) -
im(2:2:end,2:2:end));
subimHH = (im(1:2:end,1:2:end) - im(2:2:end,1:2:end) - im(1:2:end,2:2:end) +
im(2:2:end,2:2:end));
imout = [subimLL subimLH; subimHL subimHH]./2;  % added divide by 2


function imout = ihaar2(im)
[a b] = size(im); imout = zeros([a b]);
subimLL = im(1:a/2,1:b/2);
subimHL = im(a/2+1:a,1:b/2);
subimLH = im(1:a/2,b/2+1:b);
subimHH = im(a/2+1:a,b/2+1:b);
imout(1:2:end,1:2:end) = (subimLL + subimHL + subimLH + subimHH)/2;
imout(2:2:end,1:2:end) = (subimLL - subimHL + subimLH - subimHH)/2;
imout(1:2:end,2:2:end) = (subimLL + subimHL - subimLH - subimHH)/2;
imout(2:2:end,2:2:end) = (subimLL - subimHL - subimLH + subimHH)/2;
```