Compression of Images

Instructions: Email the indicated figure and 3 values to tleise@amherst.edu by next Wednesday.

Recall that we store images as matrices whose entries give the color of each pixel in an image. For grayscale, the usual convention is for entries to be integers between 0 and 255, where 0 is black and 255 is white. For color images, each entry is a vector of 3 values that codes for the color. We'll continue to focus on grayscale images, but similar ideas can be applied to color images.

Exercise 1 (Horizon image) Most images you can download from the web are already the result of compression (e.g., stored in a compressed format like jpg), so it's difficult to gain much from further compression. We'll start with the "Horizon" image posted on the Math 320 webpage, which is a raw image that is compressible. Load the Horizon.mat file into Matlab via `load('Horizon');` and then plot it in Matlab, imitating what you did in the previous lab. *Don't email this image–it's rather large!*

Recall that we can apply the 2D Haar transform to an image using the function `HT2D`, and invert using `IHT2D` with the scripts from the edge detection lab.

Apply 3 iterations of the 2D Haar transform to the Horizon image, imitating what we did in the edge detection lab. Plot all the approximation and detail matrices together as a single image:

```
C3=[A3 BV3; BH3 BD3];
C2=[C3 BV2; BH2 BD2];
C1=[C2 BV1; BH1 BD1];
figure;
imagesc(abs(C1));
[M,N]=size(C1); axis equal; axis([0 N 0 M])
colorbar
```

Here we plot the absolute value because we are interested in comparing how many Haar coefficients are close to zero versus how many are significantly larger than zero in absolute value. What you should observe in the decomposed image is a lot of pixels that are essentially zero, with all of the information about the image compressed into a small proportion of the coefficients. *Don't email this image either–it's rather large!*

Exercise 2 (Energy of images) Let's check that the information in the image really is being concentrated into a relatively few Haar coefficients. Sort the absolute values of the coefficients in descending order for `A` and `C1` and compare how rapidly the coefficients decay to zero:

```
figure;
plot(sort(abs(A(:)),'descend'),'k'); % A(:)  converts matrix to column vector
hold on;
plot(sort(abs(C1(:)),'descend'),'r');
legend('Image','Haar coefficients')
```

The *energy* in a vector or matrix is the sum of the squared coefficients, which we'll normalize by dividing by the total energy. The following code calculates the cumulative energy in a sorted array of the image entries:

```
energyA=sum(A(:).^2);
sortedenergyA=cumsum(sort(A(:).^2,'descend')/energyA)';
```

Find the analogous sorted and normalized cumulative energy array for `C1`, then plot the cumulative energy arrays for `A` and `C1` together in a figure. Submit this figure with legend and axes labels added.

Exercise 3 (Threshold compression method) One method to compress the image is to select a tolerance or threshold that retains 99.9% of the energy in the matrix `C1` (which stores the 3rd iterate's approximation matrix along with the detail matrices from all 3 iterations, so it has all the information needed to restore the full image). To determine this threshold, we want to find the first index for which the `C1` cumulative energy array entry is at least 0.999, find the value of the corresponding coefficient, and use that value as our tolerance in generating the compressed version:

```
index=find(sortedenergyC1>=0.999,1,'first');
sortedC1=sort(abs(C1(:)),'descend');
tolerance=sortedC1(index);
C1comp=C1.*(abs(C1)>=tolerance);
```

Email me the value of `tolerance` and what proportion of the entries of `C1comp` are nonzero, which you can compute via `index/(M*N)`.

The proportion of coefficients retained is pretty small (and would be even smaller if we used a more appropriate type of wavelet). However, compressing the image isn't quite as simple as throwing away the zeros, as we need to record the indices of where the nonzero coefficients are. So the actual compressed file won't save quite as much space as this simple calculation might suggest. An efficient algorithm for storing the nonzero entries in a sparse matrix like `C1comp` is a further key ingredient in compression, but beyond our purview. If you are interested, look up Huffman coding, a common approach to compactly encoding this type of information. To save space, we would also quantize the coefficients to be integers 0 to 255, which requires only 8 bits of storage, rather than the 64 bits required for floating point numbers.

Exercise 4 (Uncompressing the image) To "uncompress" the image, iterate the inverse Haar transform three times applied to the thresholded matrices (or take the updated matrices directly from `C1comp` by figuring out the ranges of indices corresponding to each submatrix). The first two steps are shown below:

```
A2uncomp=IHT2D(A3.*(abs(A3)>=tolerance),BH3.*(abs(BH3)>=tolerance),...
BV3.*(abs(BV3)>=tolerance),BD3.*(abs(BD3)>=tolerance));
A1uncomp=IHT2D(A2uncomp.*(abs(A2uncomp)>=tolerance),BH2.*(abs(BH2)>=tolerance),...
BV2.*(abs(BV2)>=tolerance),BD2.*(abs(BD2)>=tolerance));
```

We can calculate the distortion due to compressing via the ℓ_2 relative error:

```
error=sqrt(sum((A0comp(:)-A(:)).^2)/energyA)
```

Calculate the relative error for this example and state the value in your email.