Audio Compression Project

Fundamentals of Digital Signal Processing
UC Berkeley Spring 2004

Gregory Pat Scandalis

1.0 Step 1

Read the paper by Davis Pan.

I read the paper. I have to admit I made a wrong turn for several days trying to directly implement the flow diagram and equation (1).

I recognize that equation (1) is an inner loop optimization and that C is a modified impulse response to account for some of the optimization it.

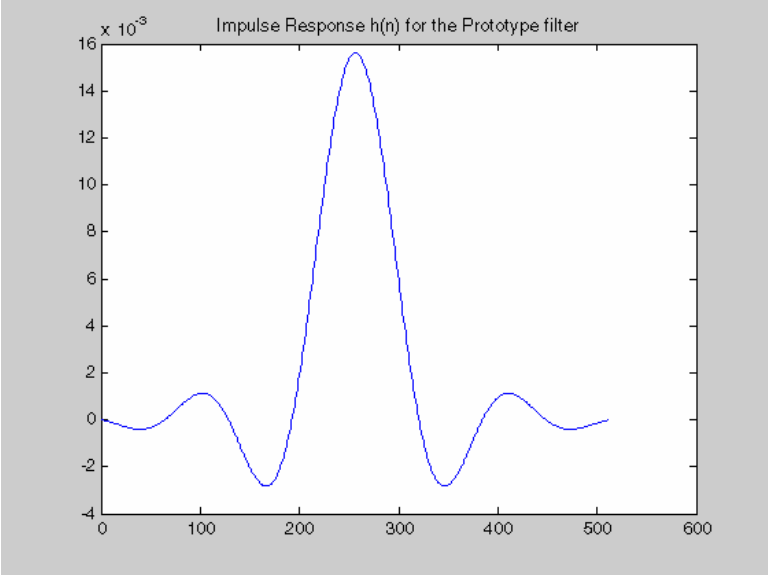
Now that I've switched to implementing equation (2) things are going much better.

2.0 Step 2

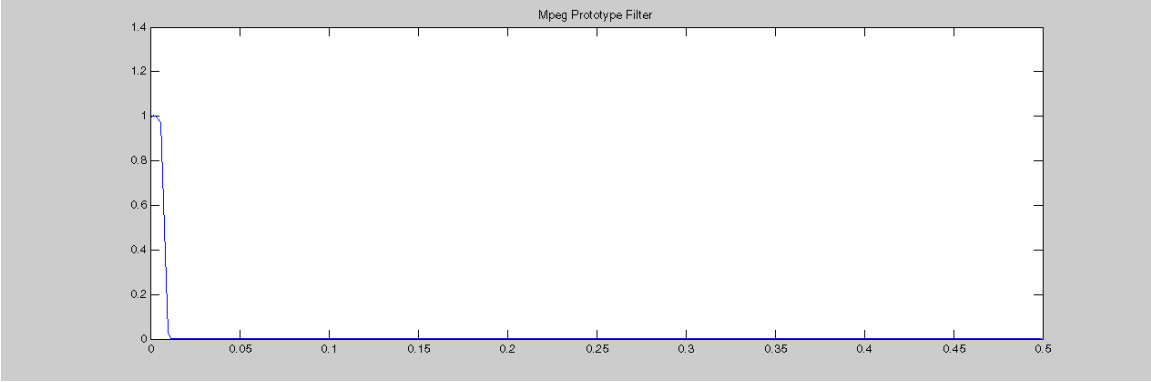
Write a routine to calculate the filter coefficients for the FIR prototype filter h[n] used in EQ(2) of Pan's article. This filter should have linear phase and it can have as many as 512 taps. It should be a lowpass filter with a half amplitude point at F = 1/128.

I modified filterdesignexamp.m to accomplish this. My file is called **PrototypeFilterDesign.m** (both step 2 and step 3). Basically I set the Fc = 1/128, the number of taps to be 512 and the AS to b 40 db.

Here is the filter h[n] and its corresponding H[F]:



Looks pretty much like the h[n] in figure 3-2 of the Davis Pan article.



Prototype filter

Here are the output parameters from the design:

```
Results...
Number of coefficients used =  512
Min Stopband Attenuation (dB)=30.23615
Target was=40.00000 (dB)
Max Passband Variation = 0.02717
Target was= 0.01000 (dB)
```

I did not spend a great deal of time iterating on the design this filter. The filter has linear phase because per the proof in the notes: a symmetric impulse response produces a constant group delay.

Note that when you run PrototypeFilterDesign.m that pauses are inserted at various steps. You need to type a <CR> to continue to the next step.

3.0 Step 3

Using the prototype filter compute the i-th filter impulse response. Use the formula:

$$H_k[n] = h[n]\cos((2*pi)((2k+1)/128)(n-16)$$

For n=0,1,2,3…N-1 and k=0,1,2…31. Where N I is the number of taps used in the prototype filter. What is the effect of multiplying the prototype cy this cosine function? Use both the theory and MATLAB to figure it out, by plotting the frequency response for each Hk[n].

Theory, from the lecture on cosine modulation:

$$H(F) = \sum_{k=0}^{M-1} bk*e(-j*2*pi*F*k)$$

substitute `bk = bk*cos(2*pi*F0*k)`

Do the math…

$$= \tfrac{1}{2}*(H(F-F0) + H(F+F0))$$

The ½*H(F-F0) shifts into the region above nyquist and can be ignored. The ½*H(F+F0) shifts the prototype filter tp the right by F0 forming a band pass filter. Cosine Modulation can be exploited to construct a bank of band pass filters from a simple low pass filter.

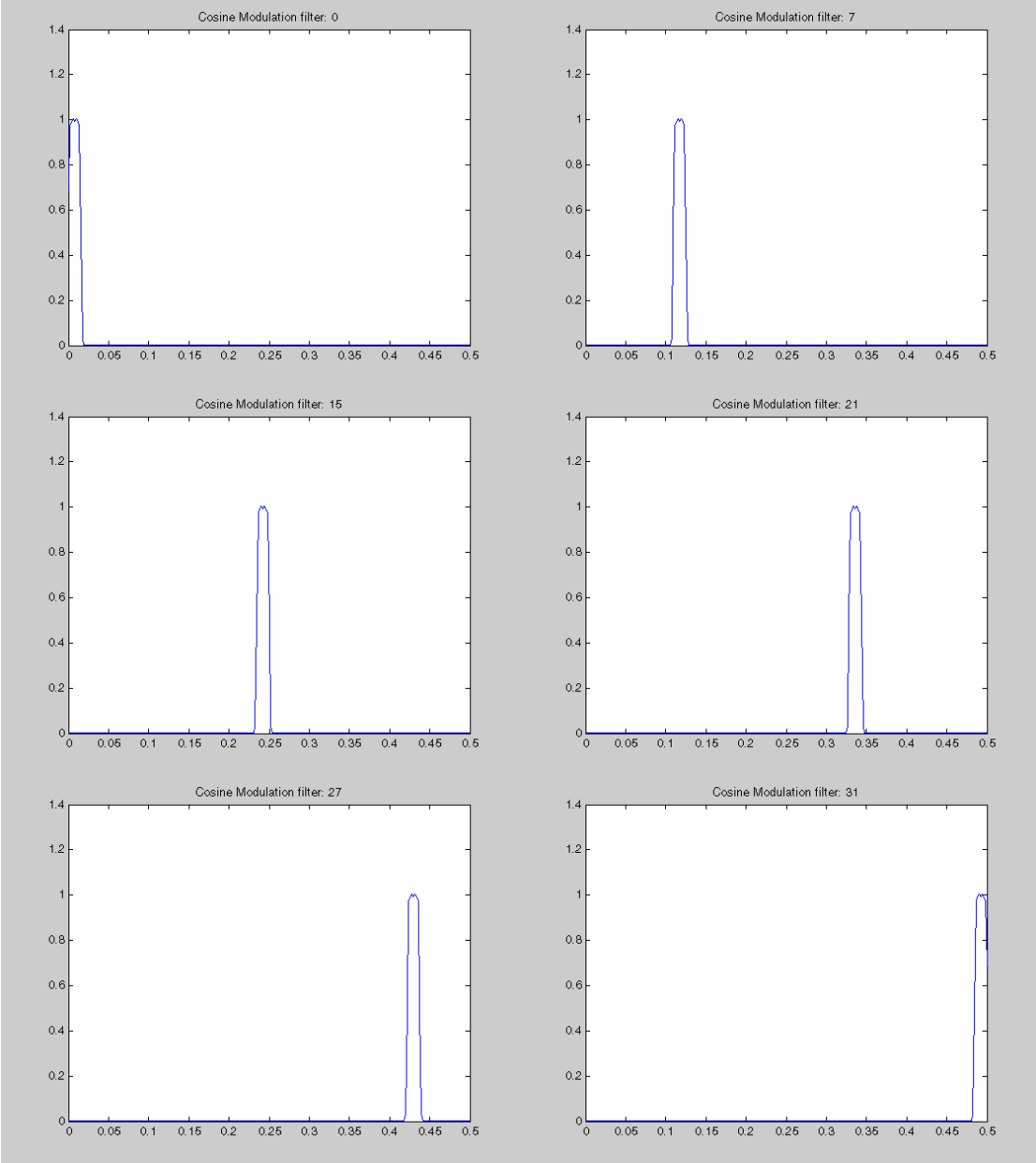
So in MATLAB, I implemented this as a 2 level loop. Because MATLAB indexes arrays from 1, I needed to make a slight change to the equation:

```
% computing the k-th filter impulse response.
% using Cosine Modulation
N = length(hw);
for k=0:31
   for n=1:N
      hk(n) = 2*hw(n)*cos(pi*(((2*k)+1)/64)*((n-1)-16));
   end

% frequency response;
[Hk,F] = iirfreqresp(hk);
plot(F,abs(Hk));
s = ['Cosine Modulation filter: ', num2str(k)];
title(s);

pause;
end
```

Here are several of the Hk[F] for several shifts.



4.0 Step 4

Implement the quantization function. This was a straight forward mapping of the MATLAB signal range (-1.0 .. 1.0) to the range of some word with 2^b bits. Because of the MATLAB signal range I was able to simplify the equation a bit:

```
z = floor((2^b)*(y+1)/2);
```

If for instance b is set to 8, then z will map values -1.0 ... 1.0 into values from 0..255 I did not actually store the values as their bit representation, but they are indeed quantized. The only special thing here is that x and y are vectors. So I quantize and de-quantize a whole vector at a time.

5.0 Step 5

Compute the Gk synthesis filters. Again this was straight forward. I just needed to be a little careful about the MATLAB indexing.

```
% as before just calculate hk,
% calculate the k'th filter using cosine modulation
% Notice that the equation is slightly different than
% from the notes because MATLAB indexes from 1 rather than 0
for n=1:N
  hk(n) = 2*h(n)*cos(pi*(((2*k)+1)/64)*((n-1)-16));
end

% now calculate gk
% extra -1 because MATLAB indexes from 1 instead of 0
for n=1:N
  gk(n) = 32*hk(N-(n-1));
end
```

6.0 Step 6

Implement the de-quantizer. Again, straight forward:

```
w = (z/(2^(b-1)))-1;
```

The only special thing here is that w and x are vectors. So I quantize and de-quantize a whole vector at a time.

7.0 Step 7

Implement the compressor and the quantizer. The MATLAB function **compress.m** implements the analysis and quantization. Compress has a debug flag that if set to 1 will display each analysis filter.

```
% convolution.  Since we are sub sampling every 32
    % its not necessary to calculate all y(t).
    for t=1:32:length(x)
        y(t) = 0;           % initialize y(t)
        for n=1:N
            if (t-(n-1)) > 0
                y(t) = y(t) + x(t-(n-1))*hk(n);
            end
        end

     % insert the subsample y into the k'th location in s.
     s(t+k) = y(t);
    end
```

Several tricky things here. Since we are sub-sampling its not necessary to calculate every y(t) for each k filter. I set up the loop to iterate by 32 thus only calculating a fraction of the y(t). Effectively the sub-sampling is built into the loop. Again the managing of indexs in MATLAB is a challenge. Finally the output vector contains the sub-sampled values for all filters grouped every 32.

8.0 Step 8

Implement the de-quantizer and the de-compressor. The MATLAB function **decompress.m** implements the dequantization and synthesis decompress has a debug flag that if set to 1 will display each synthesis filter.

```
% initialize skZeroPad to all zeros.
    skZeroPad = zeros(length(s), 1);
    %make it a row
    skZeroPad = skZeroPad';
    % zero padding
    for t=(k+1):32:length(s)
        skZeroPad(t-k) = s(t);
    end

    % convolve the zero padded sk with the inverse filter.
    for t=1:length(s)
       xk(t) = 0;           % initialize x(t)
       for n=1:N
           if (t-(n-1)) > 0
               xk(t) = xk(t) + skZeroPad(t-(n-1))*gk(n);
           end
       end
   end

   % add together all the xk samples
   x = x + xk;
```

A nice trick is used to extract and zero pad each stream of filter coeffs. The filter sythesis uses the inverse filter bank Gk. The outputs of all of the filterbanks are added together (actually accumulated) via a vector add.

I implemented a function test() that loads a sound file, plays it compresses it with user defined quantization, decompresses it, plays the decompressed data, and then saves it to a file. Here are the invocation arguments for test().

```
% test    - no debug, q = 16 bits
% test(0) - no debug, q = 16 bits
% test(1) - debug (displays analysis and synthesis filters),
            q = 16 bits
% test (0, 8) - no debug, q = 8;
```

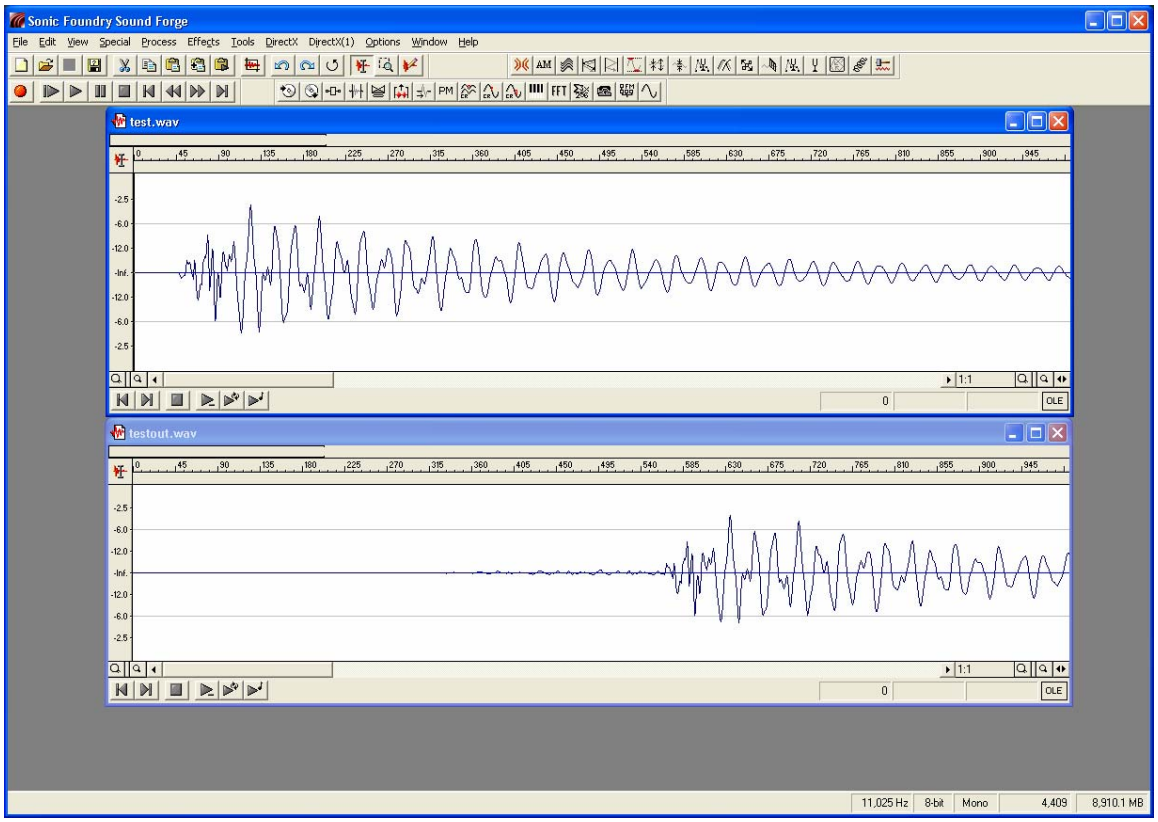
9.0 Step 9

Don't have to implement Layer 2 and Layer 3. Ok, perhaps another day.

10.0 Algorithmic Delay Question

What is the algorithmic delay of the system. We designed the filters to have a symetic impulse response because this yields constant group delay. The group delay for both the analsis and synthesis filters is (N-1)/2 Thus the algorithmic delay is the sum of the phase delays which is 511 samples.

This can be verified by viewing the input and output soundfile in an editor with a display unit set to 1 sample.



11.0 Compression Ratio

I assuming that the original file was stored at 16 bits/sample.

When quantizing the compression ration of orig:compressed is $\mathbf{2^{16}:2^{b}}$

So if I quantize to 8 bits the compression is 1:2 or ½ the original size.

I tested all quantizations from 16 .. 1. 12 sounds pretty good to me. 8 has noticeable noise. 4 starts to be inaudible and 1 is noise. The image below compares b = 16 and b = 6. You can see the quantization noise riding on the signal.