THE GSS CODEC – MUSIC 422 FINAL PROJECT <<DRAFT>>

Greg Sell, Song Hui Chon, Scott Cannon

March 6, 2005

Audio files at: **ccrma.stanford.edu/~gsell/422Final/wavFiles.tar**

Code at: **ccrma.stanford.edu/~gsell/422Final/codeFiles.tar**

ABSTRACT

We present an MDCT based perceptual audio coder that provides excellent performance at data rates of 128 kbps and higher. Advanced coder functionality includes tonal/non-tonal perceptual models, transient specific block switching, water-filling, and normalized bit allocation. The coder performs well across a wide range of source material, from speech to percussion to orchestral music.

MOTIVATION

Our motivation for this project is to leverage current best-practices (MPEG, AAC, etc.) in the design of a coder that builds on assignments in Music 422. In this project we revisit a number of different approaches to key elements of the perceptual encoder we have been creating throughout the quarter. Our aim is to create a coder that creates indistinguishable output at data rates of 128kbps.

BACKGROUND

Here we present an overview of the basic coder framework that was developed in past assignments as a foundation on which we implement more advanced features.

<< WILL FILL IN THESE SECTIONS THIS WEEK to provide sufficient background>>

Floating Point Quantization:

MDCT (Modified Discrete Cosine Transform):

Perceptual Models: We began by using a Two-Slope perceptual model for our assignments.

Bit Allocation Strategies: Bit allocation was achieved by

IMPROVEMENTS

Tonal vs. Non-Tonal Models:

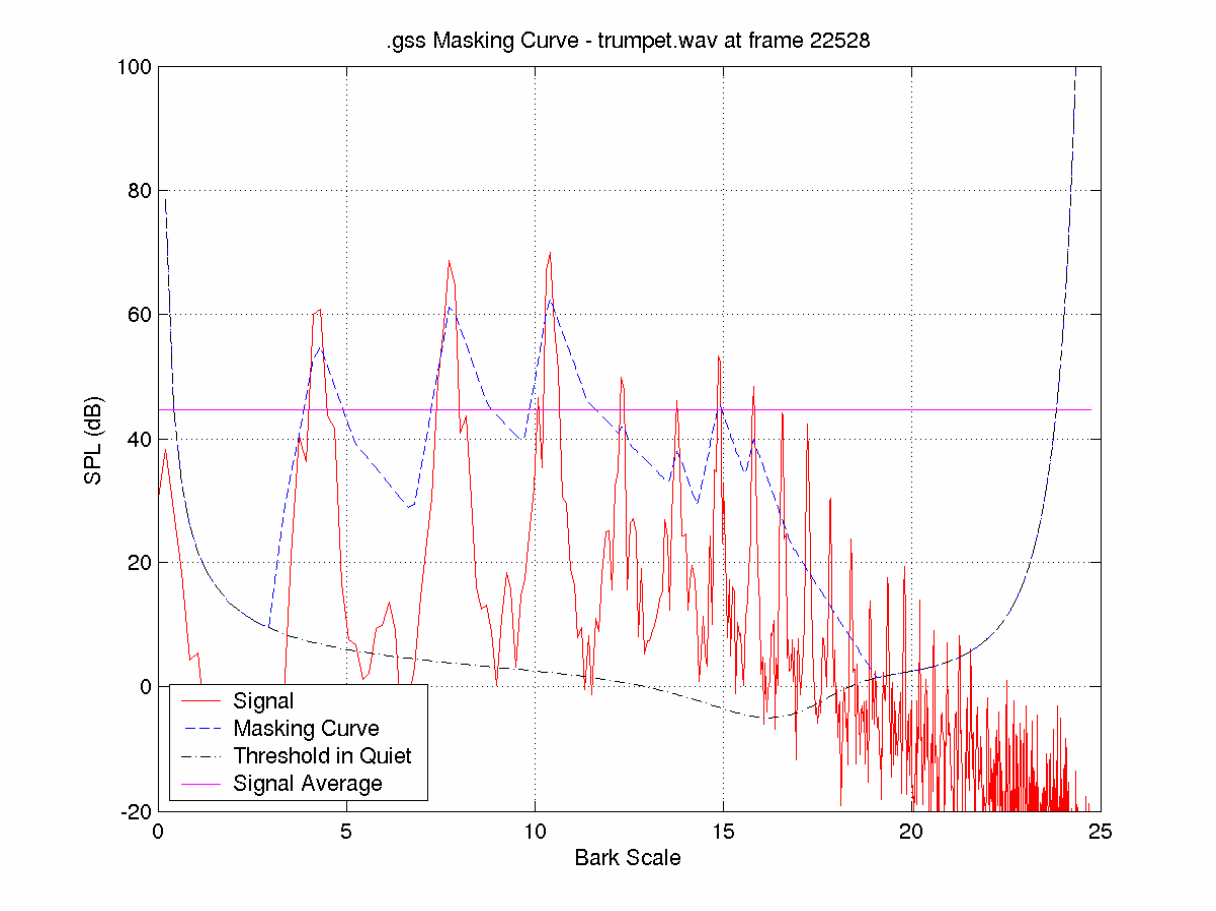
We implemented and tested the majority of spreading functions mentioned in the book: Schroeder spreading function, modified Schroeder spreading function, Model 2 spreading function, Model 1 spreading function, Terhardt spreading function, as well as the simple spreading function that we implemented for homework #4. Though the spreading functions differ one another, there was not a lot of difference in terms of the masked output using different spreading functions.

A performance comparison of psychoacoustic models using ERB with those using critical bandwidth (Bark scale) was initially considered, but after discovering that all the material related to psychoacoustics in the text book was using critical bandwidth, we decided to not pursue this idea.

We followed the logic of MPEG 1 Psychoacoustic Model 1 approach for our tonal / noise masking code. Our approach could be described as a "pseudo-model 1" method, where the peaks are treated as tonal components. However, our approach is different from the model 1 method in that we didn't calculate the maskers from nontonal components. We mixed the "model 1 method" with what we implemented for homework #4 to come up with an efficient and yet computationally reasonable global masking curve. Here, instead of computing maskers from nontonal components and calculating the global masking curve by using three types of maskers (tonal maskers, nontonal maskers and threshold of quiet), we just used the maskers from tonal components and threshold of quiet. It turns out, from our tests, that not including nontonal maskers did not hurt the performance at all.

The implementation of tonal / nontonal masking is in mask.c. While delta, the downshift amount, was uniformly 15 dB in our homework #4 implementation, it is no longer a constant in the new code. We used the formula on p284 of the text book, mimicking the MPEG 1 Psychoacoustic Model1 approach. Typical values of delta fall between 7 and 12, which may be a contributor to our excellent codec at such a low sampling rate around 90kbps/ch.

The logic in the new sumcurve function now resembles the formula for the global masked threshold in MPEG 1Psychoacoustic Model1 on p285 of the textbook. But there is a slight difference since we did not calculate the masker for nontonal components. Still it proved to work pretty well, as shown in the following figure.



Results: Using an improved psychoacoustic model with a more efficient masking curve, we could achieve a very nice sounding codec at around 90kbps/ch. So far we had three test inputs ? tonal (trumpet solo), transient (castanets and guitar) and voice. The voice turned out to be the

hardest case to properly encode and decode, as expected. Both tonal and transient exampled sounded great, while the voice sounded decent enough but not as great as the previous two. Still, we are proud of our codec's performance nonetheless.

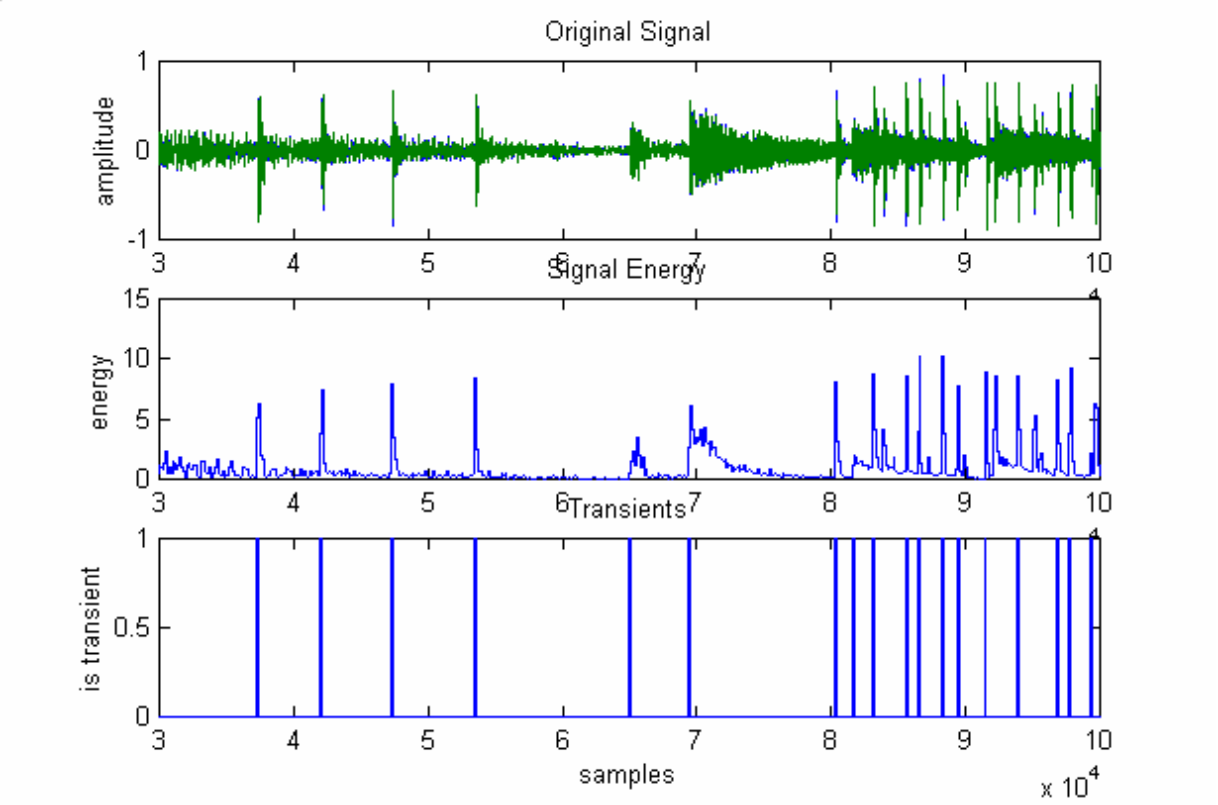
Water-filling:
Our bit allocation algorithm employs water-filling. The function is given the maximum SMR value for each band for each block. It sorts the SMRs from maximum to minimum, and then gives bits to the largest SMR. It then repeats that process (sorting then allocating) until all SMRs have either been reduced below a threshold, or until all available bits have been allocated. The algorithm can be described as follows.

Step 1: Sort subbands according to their SMR in a decreasing order.
Step 2: Fill the subband with the largest SMR, if enough bits are available.
    2.1: If the subband is empty, fill it with 2 bits per line
    2.2: If the subband is not empty, fill it with 1 bit per line
Step 3. Decrease SMR of the subband
    3.1: by 12 dB (from step 2.1)
    3.2: by 6 dB (from step 2.2)
Step 4. Subtract from P (the available number of bits)
    4.1: Nb (from step 3.1)
    4.2: 2*Nb (from step 3.2)"

Bit Reservoir:
The bit reservoir keeps track of the leftover bits for each block. In many cases, a bit allocation will not require all the available bits (in the case of silence or other easily encoded sections). The bit reservoir remembers these bits so that, if a block later in the file requires extra bits to properly encode, it can use those that were left over by previous blocks.

Block Switching:
Block switching improves an audio coder by selectively choosing the length of an analysis window based on whether a given block contains a transient or not. Shorter analysis windows lead to fewer quantization noise artifacts, often found before transients, an effect known as “pre-echo.” However long windows provide good high frequency resolution, and are therefore used anytime a transient is not present. By dynamically choosing window lengths, our coder is optimal during both steady-state and transient areas.

Transient Detection: Transients were detected in the time domain by comparing the windowed signal energy in a given block to an average of the block energies over some number of previous blocks. Energies were calculated by first windowing blocks with a Kaiser Bessel Derived window. If the signal energy for the specific block is substantially greater, than we mark the block as a transient. A noise-floor threshold is used, and blocks with signal energy below this threshold are not candidates for transients. This process was prototyped in MATLAB and ported to C++. Results of detection are shown below for a section of the castanets.wav file (lines in the third graph indicate transient detection).



Block Switching implementation: In order to provide easier block switching, the input file is read in by increments of small blocks (the block length we choose for a transient to be encoded in). If no transient is detected, the read data is loaded into a buffer of length long block (the block length we choose for all non-transient data to be encoded in). When the buffer fills, it is analyzed, quantized, and written to the encoded file. If a transient is detected, then the information in the buffer is immediately sent into the encoder, regardless of its length. That data is broken into two pieces. The last chunk of the buffer of size small block is one of the pieces, and the rest is the other. This is done so that the transient can be encoded in the smallest block possible to allow for the most precise output as possible. Essentially, the end result is that a transient is encoded by encoding its block as well as the block before it in the smallest block.

In order to allow for this flexibility, we determine the size of a given block based on two factors: the largest block, and the ratio of the largest block to the smallest. Based on those two numbers, we can pass only the number of small block chunks it would take to fill the size of the current block. For example, if the largest block size is 1024, and the smallest is 256, then the ratio is 4. If we the current block is of size 512, we only need to pass the number 2 in order to preserve the information. This implementation is useful for two reasons. First of all, we can have a variable block size for any of the blocks while only needing to encode a small number (2 instead of 512). The other reason is that we can encode data with different large block sizes and different ratios, and yet they can all still be decoded with the same decoder. So, this system provides us with a great deal of flexibility.

The main drawback to the implementation is that the window (in both the encoder and the decoder) needs to be recreated for every block, so the computation time is not ideal for the method. However, we decided that the longer computation time was worth the added flexibility.

Quantization Normalization:

Find and store the maximum value in each band to take full advantage of the quantization range in hopes of lowering the bit allocation without perceived loss. For every block, the maximum

absolute value mdct coefficient is found. This value is then rounded up to the nearest integer. This ensures that all values will fall within the range of –1 to 1. And, this also takes full advantage of the range that the quantizers are capable of encoding. The value of the normalization factor is written to the encoded file (as the variable norm), and so the decoder is able to return the values to the correct range after dequantization.

PERFORMANCE:

<< Fill in this section – how do coders perform ?>>

- discuss achieved data rates, sound qualities for different data rates
- results of listening tests for this coder
- what types of music perform well. What types of music perform poorly.

CONCLUSION

We are happy with the current implementation of our codec in the short time frame that was available to us. With more time, we would have ventured in to experimenting with Huffman coding as well as a better tonal / nontonal masking. Future work would also include efforts to improve transient detection, specifically high frequency transients that are not obvious in the time domain, but may be clear in the frequency domain when substantial broadband information emerges (may get to this during the week).

REFERNECES << need to specify these more >>

Marina Bosi Textbook.
Scott Levine Thesis – section on transient detection.
"A Tutorial on MPEG/Audio Compression", by Davis Pan. (for watefilling)