# MRC Layer 3

Meredith Burkle, Ryan Burke, Carson Hicks

Music 422: Perceptual Audio Coding
Final Report

## Introduction:

For our final project, we decided to improve upon the baseline codec by implementing Block Switching, Huffman Coding (with a Bit Reservoir for VBR), and M/S Stereo Coding. These extensions proved to be extremely beneficial to the quality of the coder. We were able to achieve consensus transparency between the coded and reference signals for bit rates as low as 34 kb/sec, a drastic improvement from the 256 kb/sec bit rate required for transparency of the same signal in the baseline codec.

## Block Switching

### Motivation:

The baseline codec developed, while operational, fell short in many of the ways common to simplistic MDCT-based codecs. One of the most audible artifacts of the baseline codec is the presence of pre-echo, which causes a sound to be heard before it would actually occur in the original file. Most pre-echo is caused by sharp (usually percussive) attacks in the file being encoded. Pre-echo is also strongly linked to the data rate of the encoded file, so lowering the data rate greatly increases the audibility of the pre-echo.

### Goals:

Pre-echo results from the length of the MDCT blocks being processed. There are two requirements for pre-echo to occur: a sharp attack and a lack of bits. If a sharp attack is present in the block being processed and the coder does not have enough bits to push quantization noise below the mask, then the quantization noise is spread in the time domain. This time domain quantization spreading is the manifestation we refer to as pre-echo.

Pre-echo can be reduced by adding the capability to switch the size of the blocks being processed. Changing to a smaller block size does result in worse frequency resolution, so short blocks should only be used in the presence of an attack, or transient. Therefore, in order to optimally switch blocks we have to first detect transients. After a transient has been found, a smaller block size can be used, but almost all steps in the baseline codec depend on the block size. The baseline codec must be generalized to accept any block size to achieve improved transient processing.

### Methods:

#### Transient Detection:

Transient detection is separated into four steps:

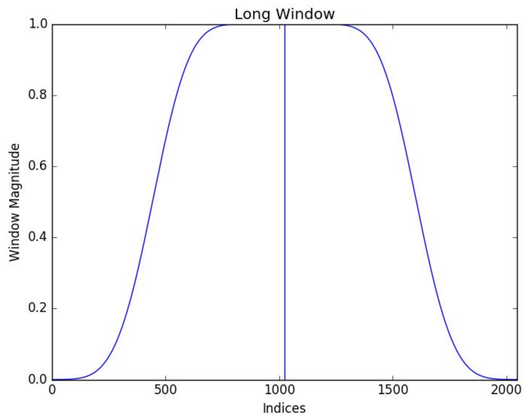
1) The current data set is highpass filtered
2) The data set is separated into sections the same length as the short block
3) The peak values of each data segment are found and stored

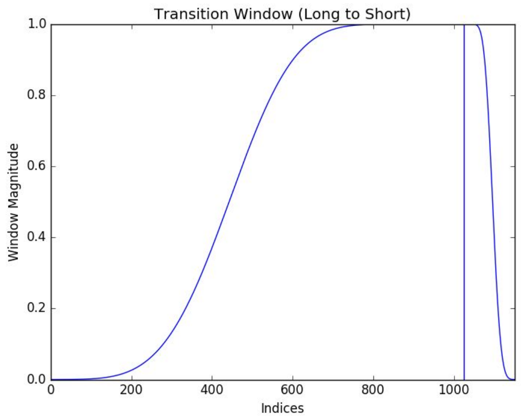
4) The peak values are compared to a set of thresholds to determine the presence of a transient

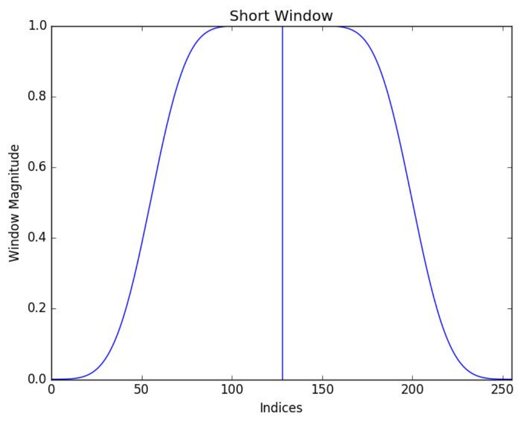
For this specific transient detection system, The input data blocks have two channels each with 1024 samples and the short blocks process 128 samples. The highpass filter used is a 20th order Chebyshev type II filter with 40 dB minimum stopband attenuation and 9 kHz cutoff frequency. The filter is separated into second order sections for improved stability. After the input data is filtered, the peaks of successive sets of 128 samples are found.The greatest peak of the set of peaks is compared to a threshold; if the value is not greater than the threshold (i.e. the filtered signal is too quiet) then a full block of 1024 samples is passed. If the peak is great enough, then the ratio of each peak to the previous peak is compared to a second threshold. When this threshold is surpassed, a transient has been detected. This method requires that the peak of the last short block is stored for comparison during the next round of transient detection.

**Block Switching:**

With transient detection available, relatively few changes have to be made to the baseline codec to accommodate block switching. The first change is in the windowing of data. The baseline codec passes data to a windowing function that constructs a symmetric window the same length as the data, and returns the product of the window and data. Instead of a symmetric full window, block switching requires a symmetric piecewise window. This means that the left and right blocks of the window are always the right or left parts of the symmetric 2048 or 256 length full windows. If two long blocks are processed, then the left and right window blocks are the left and right parts of the symmetric 2048 window. But if a long to short transition block is being processed, then the left window block is the left part of the 2048 symmetric window and the right window block is the right part of the 256 symmetric window. The figures below demonstrate each of the possible windows for the codec.







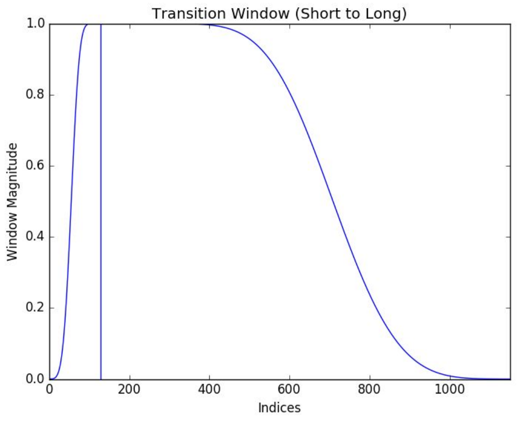


Figure 1) The different windows that can be used to process the various sizes of data blocks. The vertical line separates the blocks of data being passed down the pipeline.

The windows operate in tandem with the MDCT to achieve perfect reconstruction (if no quantization is performed). The MDCT must be modified to account for the changing location of the 'center' of the window (the location of the vertical line in the plots above). This is achieved by simply changing the phase term of the MDCT to

$$n_0 = (b+1)/2$$

where b is the length of the second block of data being processed (1024 or 128).

The final change to make block switching theoretically operational is to implement a look ahead. If we try to perform transient detection and block switching on the block we are currently processing, quantization noise could still be spread to the previous block of data through the overlap and add. In order to guarantee complete separation of the transient from long blocks we have to process the block preceding the newest set of transient data. This look ahead incurs a delay of one block length on the codec processing time.

**Block Switching Applied:**

All the tools to make block switching operational are present, but some semantics have to be added for the codec to operate. The first is addition is two bits to every block. These two bits inform the decoder of the lengths of the blocks being processed (1 for 128 samples and 0 for 1024). The decoder can use these bits to reconstruct the scale factor bands and extract the data. If the codec is implemented in this manner, then severe bit starvation occurs. The bit starvation results from the increased overhead from processing eight short blocks each with 25 scale factor bands instead of one long block. Typical block length allows for

$$((128\ kb/s\ *\ 1024\ lines\ /\ 48\ kHz)\ -\ 4\ bits * 26\ -\ 4\ bits * 25)/1024\ lines\ =\ 2.47\ bits/\ line$$

(for 1024 MDCT lines, 48 kHz sampling rate, 4 scale bits, 4 mantissa size bits, and a target bit rate of 128 kb/s) . When 8 blocks are processed for the same samples, we find

$$((128\ kb/s\ *\ 1024\ lines\ /\ 48\ kHz)\ -\ 4\ bits * 8 * 26\ -\ 4\ \ bits * 8 * 25)/1024\ lines\ =\ 1.07\ bits/\ line$$

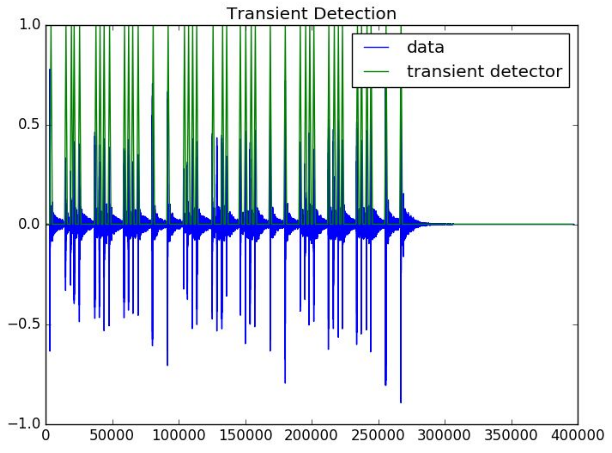
The result is a severe lack of bits for encoding the frequency domain information. This problem is mitigated by reducing the number of scale factor bands being processed down from 25 to 9.

$$((128\ kb/s\ *\ 1024\ lines\ /\ 48\ kHz)\ -\ 4\ bits * 8 * 10\ -\ 4\ \ bits * 8 * 9)/1024\ lines\ =\ 2.07\ bits/\ line$$

This reduction also results in a poorer quality codec, but it practically restores the number of bits/line to the original value.

Results:

The transient detector results are precisely as desired. The graphs below show transient detection for the castanets and german male SQAM files.



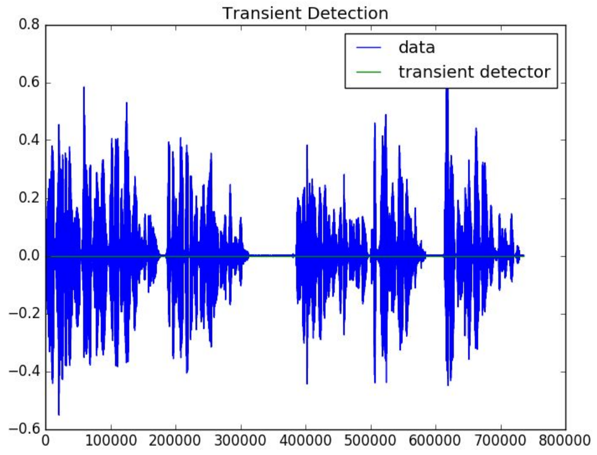


Figure 2) Transient detection for the castanets(left) and german male (right) SQAM files. All castanet strikes are detected except for one, and no peaks are detected in the german male speech.

The quality of the codec with just block switching is improved as far as pre-echo is concerned, but even data rates as high as 128 kb/s show significant bit starvation. This bit starvation is mitigated however, by Huffman coding.

## Huffman coding

**Background:**

Huffman coding is a classic method of entropy coding that allows for an increased compression rate by using fewer bits to encode more common input values. By lowering the bits required for common input values, there will be bit savings that improve the data rate overall. In order to implement this feature, we created Huffman tables whose codes were trained on a variety of different audio signals. We then integrated this functionality into our baseline coder via the steps below.

**Method:**

1) Write Huffman infrastructure

The first step is to write basic Huffman infrastructure, which consists of a) generating a frequency table based on input values, b) creating a Huffman tree based on the frequencies, and c) "walking" the tree to create the codes themselves. Generating a frequency table is a trivial step, as we simply iterate through the entire file creating a dictionary of key: value pairs, with the keys being the inputs themselves, and the values being the frequency with which that input occurs. Creating a Huffman tree involves setting up a tree for which all the leaves are the input values, and the number of branches required to reach that leaf corresponds directly to the frequency with which that input occurs. After creating the tree, we begin at the root node, and create a code by traversing either left (which indicates adding a "0" to the code string), or right (which indicates adding a "1" to the code string) until we reach a leaf. We had codes of type String, so an example code would be something like "010011". While it may be tempting to instead encode the table as a decimal value and a corresponding length (such as 19,6), it is advisable to use an object like a string, array, or even a stack so as to better keep track of leading zeros, and to make tree traversal more intuitive. (Additionally, the mantissas can only be coded as one value, as opposed to a value and a length, which presents a problem with codes such as 10, which would be (2,2), and 0010 (2,4) which correspond to completely different mantissa values yet may falsely appear to be the same code if you do not have the length. Using objects like strings removes this ambiguity.) This section is left brief as this methodology is well known. For a more in depth tutorial, visit the following link: https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html

2) Generate Huffman lookup tables

Before coding, generate a variety of Huffman lookup tables with mantissa values and their Huffman codes. These Huffman lookup tables are trained based on "genres" of audio signals, such as speech, tonal, percussive, and a table for silence/background noise. Each Huffman table representing a "genre" is calculated using multiple files. For each file, the method is to go through the entire file, block by block, channel by channel, calculating the scale factors, mantissas, bit allocations, and all other parameters as usual, and keeping track of the frequency with which mantissa values occur. At the end of calculating all the frequencies of mantissa values for every file, we should have a dictionary made up of all the mantissa values as dictionary keys, and the frequency of those mantissas as the dictionary values. At this point, we would typically use these frequencies to generate a Huffman tree, with the least frequently occurring mantissa values having the longest codes, and the most frequently occurring mantissa values having the shortest codes. Unfortunately, herein lies a complexity with Huffman coding. In order to be able to properly decode a given signal, the Huffman table *must* account for every possible input it could receive. Compressing things like text files (as in the quintessential Huffman coding problem) is easier because the range of possible input values (also known as the "alphabet") is simply the letters a-z. However, in our case, the alphabet is every possible mantissa value. Thus, the Huffman tree, and its ensuing codes can be extremely long - up to hundreds of bits long for a single (very infrequently occurring) mantissa value! But we still must account for every possible input - we would be in trouble if we received a mantissa value and simply did not have any possible code for it. The solution to this problem is to have something called an "escape code." This escape code represents all the not-very-frequently occurring mantissa values and signals to the Huffman table so that any value we ever receive will either have its own Huffman code, or will be mapped to the escape code. In this way, we represent all possible input values without having the codes get too long.

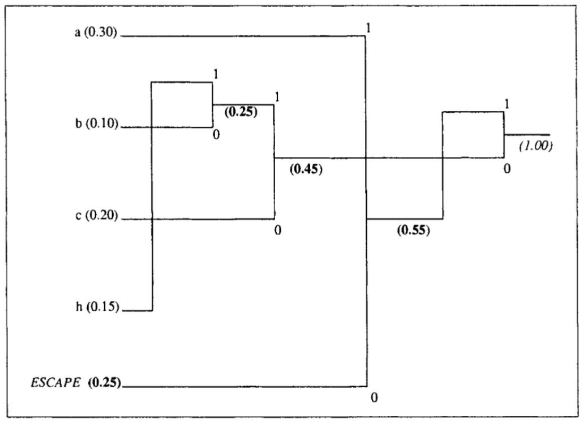
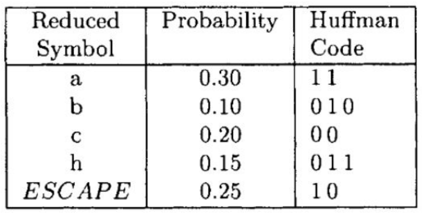


*Fig. 2.2* Huffman Tree Construction for the reduced symbols in Example 2.

The tree above represents the new frequency table, modified to have an escape code, with the "improbable" or infrequent values removed, and replaced with an escape code that represents the frequency (or probability) with which an escape value is required. The resulting Huffman code table is printed below. Both images are courtesy of the *JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures* book by Tinku Acharya and Ping-Sing Tsai.

*Table 2.2* Huffman Code for Reduced Alphabet



| Reduced Symbol | Probability | Huffman Code |
|---|---|---|
| a | 0.30 | 1 1 |
| b | 0.10 | 0 1 0 |
| c | 0.20 | 0 0 |
| h | 0.15 | 0 1 1 |
| *ESCAPE* | 0.25 | 1 0 |

The escape codes present a slight complication when re-coding mantissas, which we will discuss in the next sections. For our coding purposes, we saved the forward-direction Huffman table (meaning mantissa value → Huffman code), the reverse-direction Huffman table (Huffman code → mantissa value), and the root Huffman tree object (useful for walking the tree in the ReadDataBlock section).

3) Calculate Huffman gain (decide whether to use Huffman or not)

During the encoding stage, we calculate the scale factors and mantissas as we normally would, but after this we decide whether to re-code the mantissas instead as Huffman codes. This can be decided on a block by block (and per channel) basis. In order to determine whether to use Huffman tables, we need to see if there would be bit savings; i.e., if fewer bits are used for a particular Huffman table than would normally be used by just encoding the mantissa as is. We calculate the bit savings by iterating through each possible Huffman lookup table, and seeing how many bits would be used to encode that particular block's mantissa values as compared to encoding the mantissa values normally.

4) Re-code mantissas

After deciding which, if any, Huffman table to use, we then go through each mantissa value and determine its Huffman code. To accomplish this, we simply open the Huffman table and get the code string corresponding to the given mantissa value. However, here is when the escape codes present a problem. If we just encode the escape code alone, we will have no way of knowing what the corresponding real mantissa value is when we decode. The literature was not clear on a single correct way of handling this situation, but we decided to encode the escape code, immediately followed by the original mantissa as was described in *JPEG2000*. An example escaped mantissa might look like "010101/253", where "010101" is the escape code and 253 is the mantissa value that needed to be escaped.

5) Write to the bitstream

Writing to the bitstream proceeds as normal, except for the addition of writing which Huffman table is being used, and we also need to write the codes instead of the original mantissas.

6) Read from the bitstream and recover original mantissa values

Reading from the bitstream is a little different - we used to read the number of bits corresponding to the bit allocation for a given critical band, but in the case of the Huffman codes, which are variable length, we do not know how many bits to read. As a result, we need to open up the corresponding Huffman tree, and “walk” the tree reading one bit at a time from the bitstream until we reach a leaf, and can then find the corresponding code in the reverse Huffman table (reverse meaning Huffman code → mantissa value). Once we are able to find this Huffman code, we can either decode it directly, or if we find it is an escape code, we can read further to get the original mantissa value (which has presumably been encoded directly following the escape code). One critically important implementation note is the format of the mantissa array. The mantissa array, originally of length 1024, can be separated into N critical bands. The incoming mantissa array (incoming to WriteDataBlock), after being encoded, is typically less than 1024 lines due to the fact that some of the critical bands have a bit allocation of zero. Because that band’s bit allocation is zero, the mantissa values are simply not included in the array. However, in the mantissa array that is created in ReadDataBlock, the length *is* 1024. Instead of omitting those indices and thus shortening the mantissa array, there are zeros placed in the indices corresponding to the 0-bit-allocation critical band instead to maintain the length of 1024, and the Decode function will then find the bands for which the bit allocation is not zero, and index into them using the lowerLine and upperLine information in codingParams. If the length of the mantissa array is *not* 1024 as expected, then the lowerLine and upperLine information will be used improperly and will lead to very subtle bugs!

7) Pass this to the Decode function, which has remained the same

The decode function did not need to be changed, as the mantissas that are passed in should already have been reconstructed to be their original values.

## Joint Stereo Coding

**Introduction:**

To further improve the quality of our codec, we implemented Mid Side (M/S) Stereo Coding, a variant of Joint Stereo Coding that takes advantage of signal redundancies in a stereo recording in order to redistribute bits more efficiently.

In a typical stereo recording, there is a strong correlation between the left and right channels. If we intelligently identify when and which sections of the left and right channels are correlated, we can remove a large amount of redundancy in our bit allocation routines and can allocate more bits to the critical portions of the stereo signal.

When this strong interchannel correlation is identified, rather than coding the left and right channels separately, M/S Stereo Coding transmits the normalized sum and difference of the left and right channels. In comparison to other joint stereo coding techniques, M/S Stereo Coding has some extremely useful properties

- Perfect Reconstruction
  By simple substitution, one can see that converting between L/R and M/S representation is an invertible transform.

$$M = \frac{L+R}{2} \qquad S = \frac{L-R}{2}$$

$$L = M + S \qquad R = M - S$$

- Full Spectral Range Applicability
  The M/S transformation is invertible and thus preserves the original spatial information, which allows for very selective scale factor band by scale factor band application of M/S coding.

- Spatial Masking
  M/S coding additionally allows for the coder to account for  Binaural Masking effects, extending the baseline tonal/atonal masking models already accounted for.

Each of these properties and how they are accounted for in implementation will be described in the subsequent sections.

**Implementation and Procedures:**
In order to fully utilize the broadband spectral reconstruction property of the M/S Transform, we decided to switch between M/S and L/R coding on a block by block, scale factor band by scale factor band basis.

To determine if M/S coding will be turned on for the current scale factor band, we compare the sum and difference energy content of the band, as demonstrated in the following formula:

$$\left|\sum_{lower}^{upper} (l_k^2 - r_k^2)\right| < 0.8 \left|\sum_{lower}^{upper} (l_k^2 + r_k^2)\right|$$

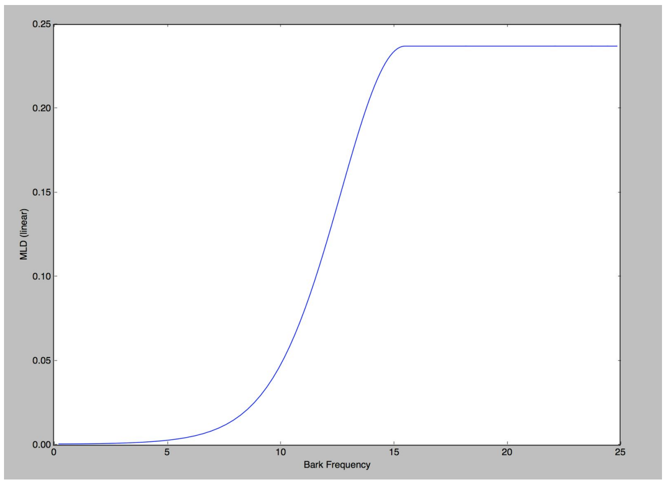
Where *lower* and *upper* are the lower and upper indices of the current scale factor band, and *l* and *r* are the respective MDCTs of the left and right channels. Note that sending this information to the decoder will use an additional n bits per block where n is the number of scale factor bands.

Now that we know which critical bands will use M/S coding, we must now account the different masking threshold calculations for the critical bands which M/S coding is activated. If L/R coding is enabled for a certain subband, the masking curves and subsequent Signal to Mask ratios are calculated identically as in the baseline codec. If M/S coding is activated, we must factor in binaural masking effects when producing our final masking curve and final SMRs.

Due to the joint nature of processing both channels simultaneously under this transform, we allocate bits in a joint nature as well, applying our normal waterfilling algorithm to both channels at the same time with double our usual bit budget.

To calculate the masking curves for M/S coding, generate the basic masking curve from the baseline codec for the Mid signal and Side signal following the same procedure for the L/R channels. Next, in order to account for the aforementioned stereo masking, an additional frequency dependent drop is applied to the masking curves. This Masking Level Difference factor can be approximated as a function of Bark frequencies in the following formula derived based on the plots and additional fits in J.D. Johnston's *Sum Difference Stereo Transform Coding*, (cited below):

$$MLD(z) = 10^{\left(1.25\left(1-cos\left(\pi\frac{min(z,15.5)}{15.5}\right)\right)-2.5\right)}$$



We then downshift our M/S thresholds by the output of this function then choose between the new downshifted versions and the original curves using this formula presented in Johnston's paper.

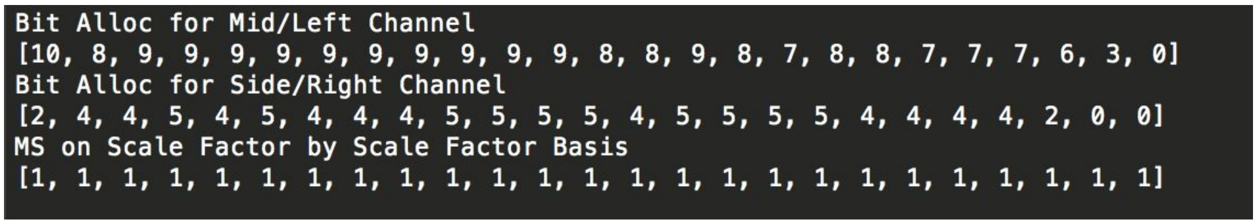
$$MidFinal = max(MidNormal, min(SideNormal, SideDownshift)$$
$$SideFinal = max(SideNormal, min(MidNormal, MidDownshift)$$

As described above, the M/S masking thresholds will then go through a jointly processed Bit Allocation and the final bit allocations are formed on a scale factor band to scale factor band basis depending on if M/S is turned on for the current band.

The signal is then quantized and dequantized on a band by band basis, checking which MDCT information to use based on if M/S coding is on or off for the current block. Finally, the dequantized M/S data is transformed back into L/R by using the inverse transformation formula in the properties section.

**Examples of M/S Coding:**
To get a sense of how M/S coding improves the overall codec performance, let's analyze a specific example found during the encoding process of the classic German Male Speaker reference SQAM file in block number 171.

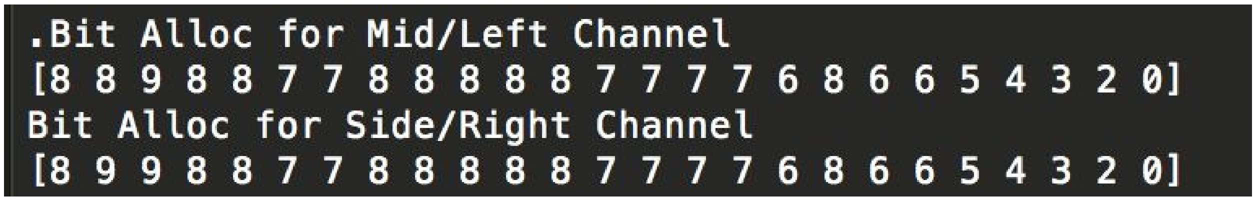
In this specific block, there is a high correlation between the left and right channel, and M/S coding is turned on for every scale factor band.



```
Bit Alloc for Mid/Left Channel
[10, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 8, 9, 8, 7, 8, 8, 7, 7, 7, 6, 3, 0]
Bit Alloc for Side/Right Channel
[2, 4, 4, 5, 4, 5, 4, 4, 4, 5, 5, 5, 5, 4, 5, 5, 5, 5, 4, 4, 4, 4, 2, 0, 0]
MS on Scale Factor by Scale Factor Basis
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Due to the extreme similarities between both channels, one can achieve a higher quality coding by encoding the normalized sum and difference of the two channels. Since the side channel is so small relative to the normalized sum, more bits are allocated to the mid channel first. Once the mid channel (which contains the majority of the information) is filled, the side channel is then allocated bits according to the normal water filling procedure.

The results without any M/S coding are below:



```
.Bit Alloc for Mid/Left Channel
[8 8 9 8 8 7 7 8 8 8 8 8 7 7 7 7 6 8 6 6 5 4 3 2 0]
Bit Alloc for Side/Right Channel
[8 9 9 8 8 7 7 8 8 8 8 8 7 7 7 7 6 8 6 6 5 4 3 2 0]
```

As you can see, the two channels have almost exactly the same number of bits allocated to each subband, implying a strong correlation between the two channels and suggesting a more efficient method of bit allocation should be considered, like M/S stereo coding.

## Results:

To test our new and improved MRC Layer 3 codec, we encoded and decoded signals from each category recommended for the listening tests. We used (shortened versions of) the Castanets SQAM file for our percussive reference, the German Male Speaker SQAM file for our speech reference, and the Frequency Tones SQAM file for our tonal reference.

We started out at conservatively high bit rate and lowered the target bits per sample until consensus transparency was not observed. We then noted the last trial where transparency was observed across the board and calculate the resulting bit rate from the pacfile size. The results are summarized below and can be viewed in the “demoCodedFiles” section of our project submission:

| | Duration (sec) | Size of Pacfile (kb) | Rate (kb/sec) |
|---|---|---|---|
| German Male | 4.12 | 139 | 34 |
| Castanets | 6.57 | 297 | 45 |
| Tones | 7.47 | 296 | 40 |

Moving forward, we would like to have more test subjects undergo our transparency test and solidify and adjust our preliminary data. We are very happy with the current implementation, but we would like to continue improving MRC Layer 3 as advances in parametric coding are made.

## References:

[1] J. Johnston and A. Ferreira, Sum-difference stereo transform coding," in Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on, vol. 2, pp. 569-572, vol.2, mar 1992.