**The Transmission and Storage of Data from the Electromagnetic Calorimeter of the Atlas Detector**

**(Tania Henry)**
**Summer 2001**

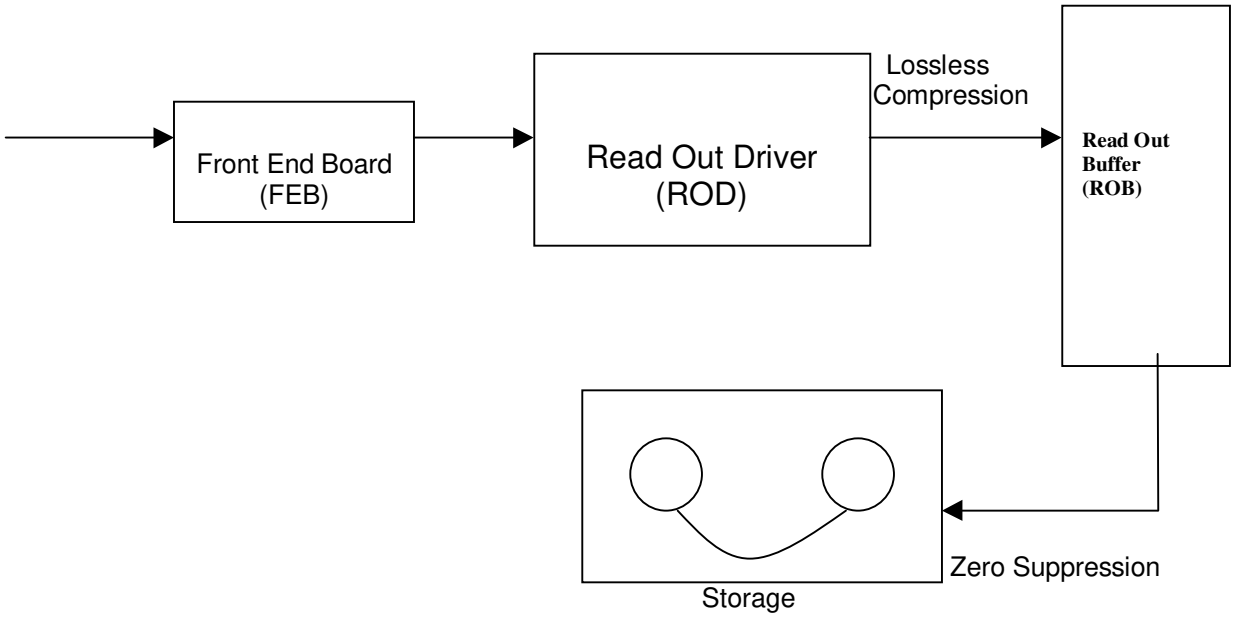



**Figure 1.0**

Our goal is to find a way to reduce the volume of digital data collected from the Electromagnetic calorimeter as well as reduce data transmission time. We will use data compression and zero suppression to carry out this process. Lossless compression makes use of redundancy in the data, while zero suppression eliminates unwanted data. In the above diagram, data leaves the Electromagnetic Calorimeter and travels through the Front End Board and Read Out Driver, which are electronics we are working on at Nevis Labs. The data is then compressed and travels through optical fibers to the Read Out Buffer after which zero suppression is carried out in this case to eliminate $\pi^{\circ}$'s (two photons), from single photons. This elimination process will help us to identify the Higgs boson, would help us explain more clearly the origin of mass.

## Data Compression

A Lossless data compression method is used because we need to preserve our original data as much as possible. Lossless data compression makes use of redundancy in the data. In the decompression process, the original data is recreated from the compressed data, by bringing back the removed redundancy. We cannot afford to allow any distortion of the information we collect, as we do not want to miss an interesting event.

Collisions occur every 25 nano-seconds in the electromagnetic calorimeter. The first trigger filters the interesting events, which hit the detector at a rate of 100 kHz. There are

approximately 200 000 channels, hence we are looking at bandwidth of 8x10^10 bytes per second. This enormous amount of data requires vast amounts of optical fiber for transmission, and also takes up a lot of storage space. As a result of this, very fast and accurate modes of data transmission and storage are needed.

## Overview

We will be packing data in an n-bit field, where n-bit is defined as a position in the bit stream where we will write a chunk of data. In the n-bit field, the most significant bit is bit n-1, and the last bit is bit 0.

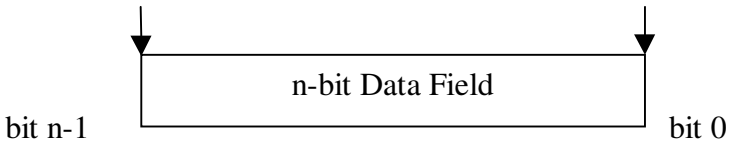



**Figure 1.1**

The data is in signed integer format, is packed into 32-bit words, and compressed by 16-word packets. A packet is a block of data to be compressed.

Our compression scheme is based on the Rice algorithm, which codes data close to the source entropy and can be extended to any entropy range. A reference is chosen from our list of raw data values. From this reference, a difference is calculated between each consecutive value and the reference. Hence the data to be compressed are (if $X_0$ = reference) $X_1$-$X_0$, $X_2$-$X_0$, $X_3$-$X_0$, ...., $X_{16}$ -$X_0$. These differences are then pre-processed to positive integers using the map function [2 x I + 1] if I <0 otherwise [2xI], where I is the input value. This map function is used because the sign bit is very random, and we want to avoid it appearing in the Most Significant Bit part of our data. A bit "split sample" technique is used to separate the pre-processed samples into **k** LSB (Least Significant Bits), and the remaining MSB (Most Significant Bits). The LSB is transmitted uncompressed, while the MSB is coded using a Fundamental Sequence Codeword (FS Code.)

Example sample 1000 0001 k-bits k=4
MSB LSB
FS – We reverse MSB for coding purposes

The value of **k** is chosen based on the highest compression ratio for 16 packets. The compression ratio is the ratio of uncompressed to compressed data.

$$\text{Compression Ratio} = \frac{\text{Uncompressed Data Size}}{\text{Compressed Data Size}}$$

The choice of **k** serves to minimize the size of the compressed words. Effective Lossless compression requires low entropy conditions, which is a measure of the smallest average number of bits that can be used to represent each sample. The number of bits used to represent the compressed data must be small, hence compression ratio large.

An approximate value of **k,** can also be calculated using the formula $k = \log_2 \sum_{i=1}^{N} \frac{|x_i|}{N}$, where xi is the input data, and N is the packet size. This method of finding k is derived from $Nk + \sum_{i=1}^{N} \frac{|x_i|}{2\text{^}k}$, and is based on the correlation between the best value of **k** and an estimate of the expectation of the magnitude of data from the preprocessor. We now have two values for **k**, an approximate value, and a value selected from finding the minimum compression size. Results show that these two values of k are very close. We however chose to use k found using the minimum compression size, to carry out our compression scheme.

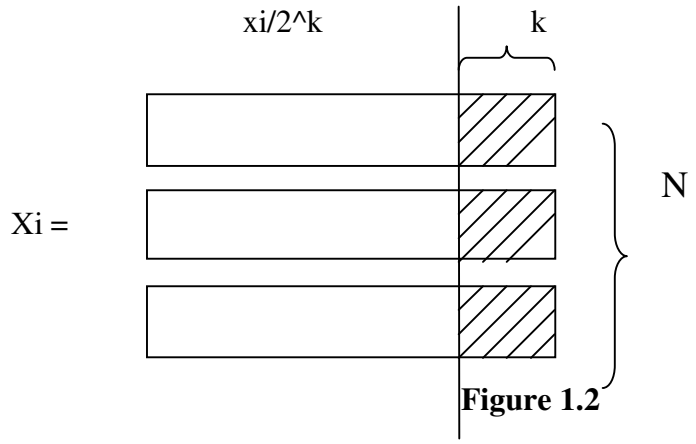



**Figure 1.2**

This diagram shows the relationship between k, the packet size (N), and the input data values (x).

The Fundamental Sequence (FS) codeword mentioned above is used to encode the (Most Significant Bit) MSB. The FS code consists of n zeros followed by a one to represent the preprocessed samples.

**Fundamental Sequence Codewords As a Function of the Preprocessed Samples**

**Figure 1.3**

| MSB | FS Codeword |
|---|---|

| | |
|---|---|
| 0 | 1 |
| 1 | 01 |
| 2 | 001 |
| 3 | 0001 |
| n | n zeros |

The kth sample splitting option described above, removes k-least significant bits (LSBs) from the binary representation of each preprocessed sample, and encodes the remaining bits with a FS codeword. This produces a codeword where LSB has length k-bits, and MSB has variable length. Data is compressed by filling the bit stream with k-bits, FS and then LSB in the following format, k, $FS_0$, $LSB_0$, $FS_1$, $LSB_1$, $FS_2$, $LSB_2$…., the whole packet is compressed.

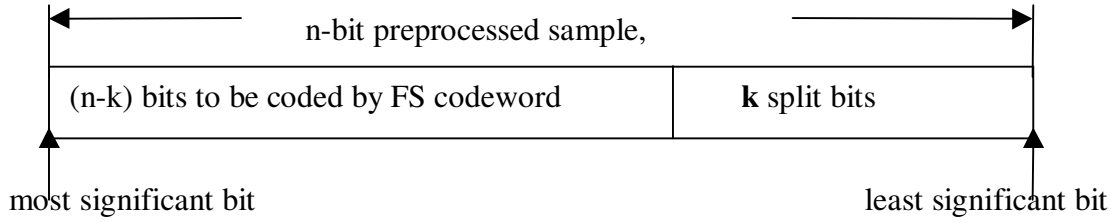



**Figure 1.4**

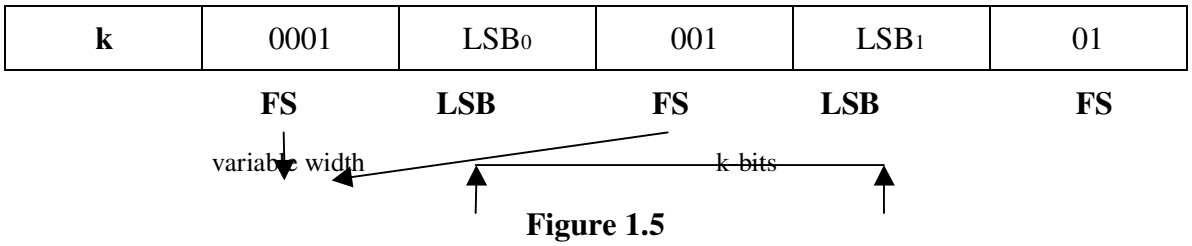
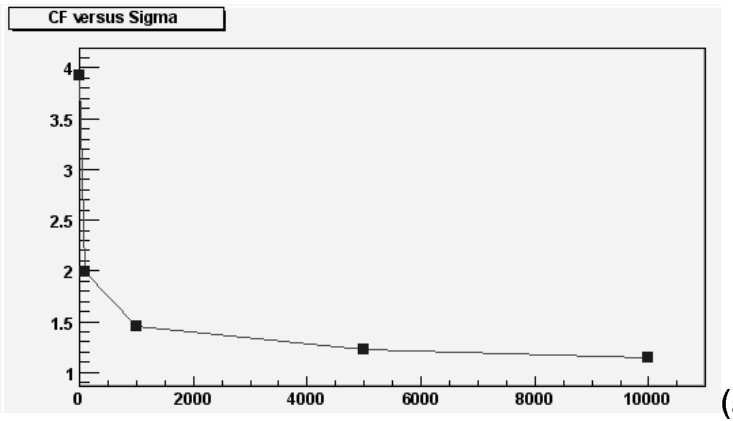



**Figure 1.5**

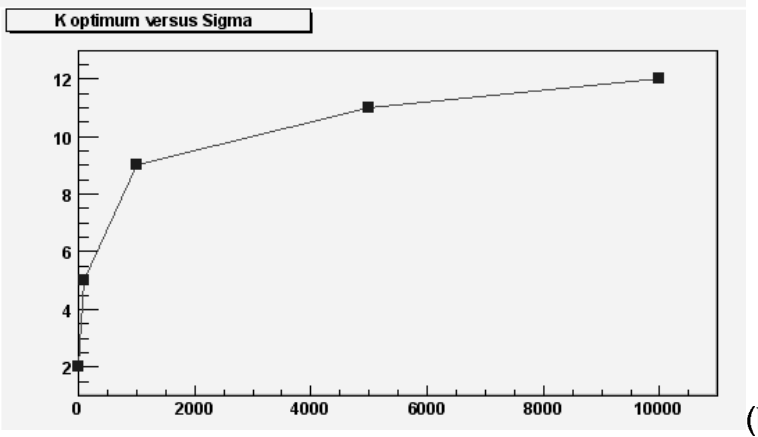
**Analysis**

We wrote a program that simulates the uncompressed values, and calculates the compressed parts for a list of **k** values. Entropy (randomness) is specific to the data we are analyzing, and k automatically adapts to it. We chose the value of **k** which corresponded to the highest compression ratio. **k** is a constant calculated from the minimum compression size for a list of 16 consecutive values generated from a set of random numbers.

Figure 1.5 (a) represents the compression ratio versus sigma. Sigma is a representation of the width of the gaussian distribution of a set of randomly generated numbers. We are using this distribution to analyze the relationships between **k**, the compression ratio, and sigma, and to calculate the most suitable **k**.



**(a)**



**(b)**

**Figure 1.6**

Figure 1.6 (a) shows the relationship between the compression ratio, and sigma. As we can see as sigma increases the compression ratio decreases. Sigma has a direct relationship to disorder/entropy, so as entropy increases, the compression gets worse. In relation to our randomly generated numbers used for analysis, these results show that there is a greater number of variations in the difference between the reference we chose and **k**, as sigma increases.

Figure 1.6 (b) is a plot of the 'optimum k'-value versus sigma, and it can be seen that **k** increases with sigma. Again as sigma increases, the gaussian distribution gets wider, and remember k is chosen based on the size of the number of bits needed to be compressed, so as a consequence we have a larger bit representation, hence an increase in **k**.

We used the Rice algorithm described above to calculate the compression ratio (csize), and the uncompressed and compressed data size.

We also used our approximate method for calculating k, using the formula formula $k = \log_2 \sum_{i=1}^{N} \frac{|x_i|}{N}$, where xi is the input data, and N is the packet size.

We calculated the average of the sum of the number of bits the compressed data will occupy, using both the real and approximated k values. Using packet sizes of 16, 32, 64 and 128 consecutive energies, we compared these two averages for different distribution sizes ($\sigma$). For each packet size (len), we saw that the average compression size found using the approximate and exact k-values were very close (See table 1.a). Also for the different packet sizes, as distribution width ($\sigma$) is increased, the average compression sizes increase as well (using the same k). This is again due to the fact that an increase in sigma gives rise to random entropy, so compression size increases as a result of this.

For different values of sigma, graphs were plotted of the average number of bits used to represent the compressed data versus packet size (len), for the same k, (the corrected value found using the minimum compression size (csize_min).

**Table 1.a**

| Sigma | Correct k<br><br># of bits for one value | Approximate k<br><br># of bits for one value |
|---|---|---|
| 2 | 3.17 | 3.18 |
| 5 | 4.14 | 4.47 |
| 10 | 5.45 | 5.46 |
| 20 | 6.45 | 6.46 |
| real data | 3.90 | 3.89 |

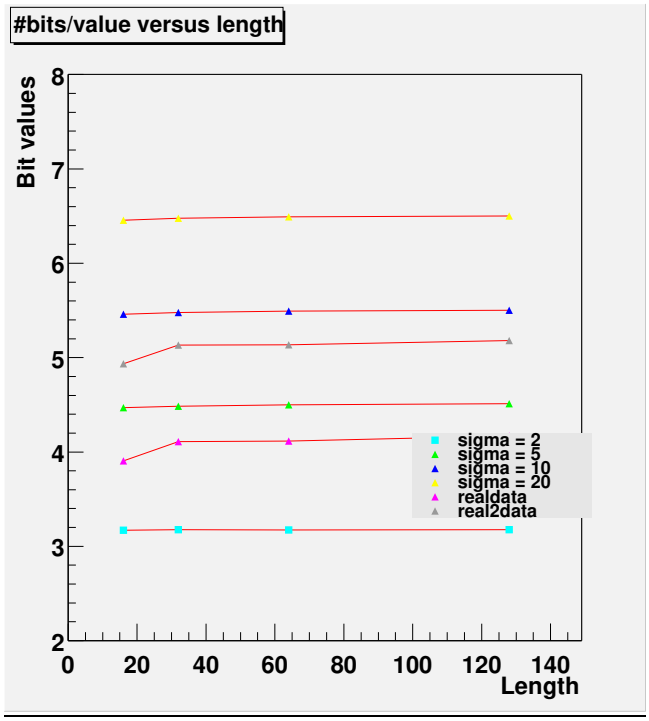



**Figure 1.7**

From the graphs, it is clear the packet size hardly influences compression ratio. We also looked at graphs of these values using 'real data', that is a distribution of correlated energy values. The previous analyses were done with simulated data values.

## Data Compression

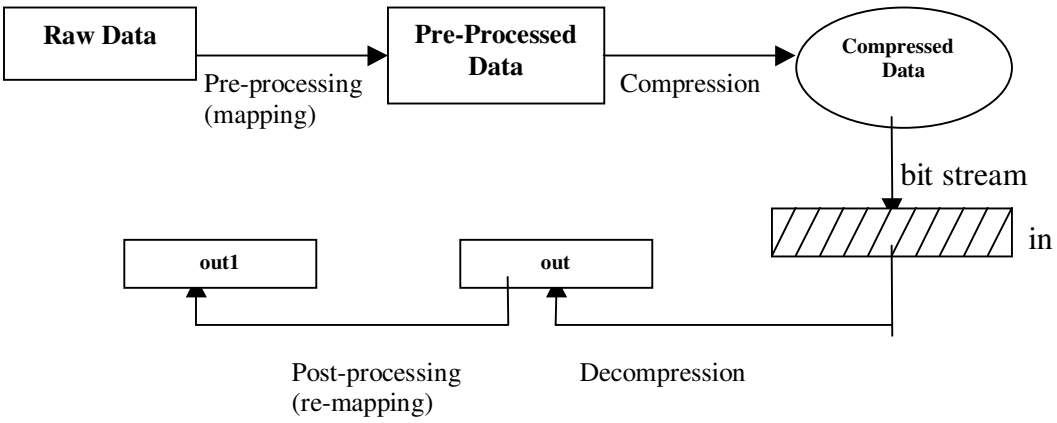



**Figure 1.8**

The above diagram describes the compression and decompression process. Since we have now determined k (by finding the minimum compression size), we wrote a program that

compresses the data. The Lossless source coding is used, and it has advantages of preserving source data, and removing redundancy.

Each pre-processed sample is then subject to k-th bit splitting, and this is done by taking the (Least Significant Bit) LSB (which is represented by k-bits) from the sample and storing it unchanged, and using a Fundamental Sequence codeword (FS code) to encode the remaining (Most Significant Bits) MSB.

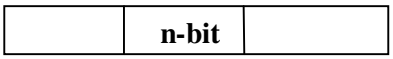
We decided to pack the data into a 32-bit bit stream by first filling the bit stream with k-bits, FS and then LSB in the following format, k, $FS_0$, $LSB_0$, $FS_1$, $LSB_1$, $FS_2$, $LSB_2$..., until the bit stream is completely filled.

LSB has length k-bits, and MSB has variable length. The bit stream is filled from left to right. The compression process involves bit shifting using the bitwise AND and OR operators, and the left and right shift bitwise operators. So ideally **k** is written to the output bit stream first, then MSB/FS code, and finally the LSB. We only code zeros for the MSB, as this simplifies the compression process. We add the 1 from the MSB/FS to the LSB, after which we write the LSB to the bit-stream.

In the compression process, a bit stream might be filled completely with only a fraction of the LSB or MSB. When this boundary problem occurs we have to continue the procedure by coding the remaining segment of the LSB or MSB to a new bit stream.

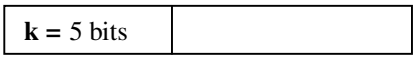
**Program details**

-First we define **n-bit** as a position in the bit stream where we will write a chunk of data.




-Defining n-bit is important, as MSB is of variable width, so we do not know the exact number of positions to fill for the FS code.

-We will initialize n-bit and update it.

-We first fill **k** 


out = k << n-bit; fill k

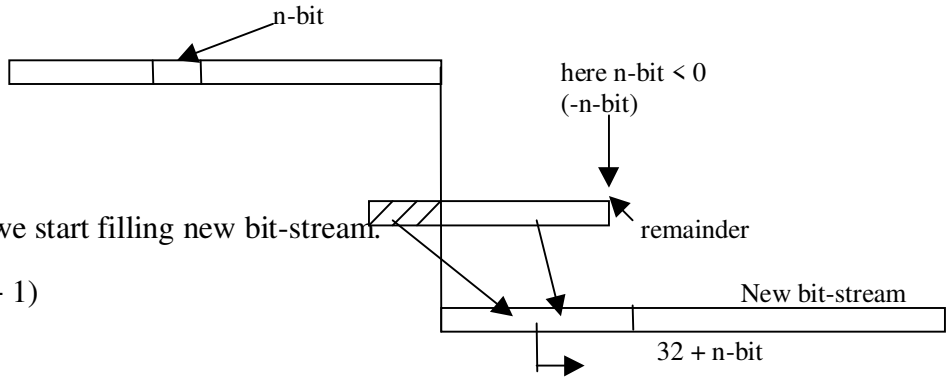
-We then initialize n-bit and update it;

n-bit = n-bit – 5 ; initialize n-bit

n-bit = n-bit – MSB; update n-bit

There might be a break in the FS at the end of a completely filled bit stream. When this occurs we continue filling a new bit-stream with zeros, as only zeros make up the modified MSB. However, if there is a break in the LSB;

- We start a new word, so we update n-bit.
- N-bit = n-bit + 32; 32 being the length of the bit stream.




So if n-bit < 0, we start filling new bit-stream.

nbit = nbit –(k + 1)

**Figure 1.9**

The remainder is put in a new position x1 so x1 = LSB1 >> (-n-bit)

-We then update the output; -out = x1| out; -update n-bit; -n-bit=n-bit + 32;

To complete the compression process we continue filling the bit streams with FS and LSB.

The compressed values in hexadecimal format match their corresponding input values, and correspond directly to the decompressed data

**MSB** = ivec[i] << k

**LSB** = ivec[i] & mask

**LSB1** = LSB| mask +1

**mask** = (1 << k) –1 that is 1 shifted by k-bits then subtract 1

Example-- 00000 11111 mask
00001 00000

**Decompression**

We wrote the compressed data values to a file, which we opened in a new program. The decompression process basically involves just reading each bit stream, and converting the compressed binary or hexadecimal values back to their original integer values.

We used the fscanf function to read the bit stream one by one.

So as in the compression process, we first read k, and fill the output with k.
n-bit is initialized to -5; n-bit=0; n-bit = n-bit-5; -shift n-bit by 5 bits (k is 5 bits).

Our goal here is to move k from its position in the bit stream originally, to its position before it was compressed. As a consequence we shift in the opposite direction.

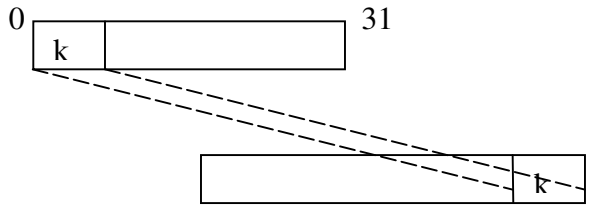



**Figure 2.0**

If k is at some random position in the bit stream, we have to devise a way of reading it to the new output bit stream to be decompressed as well. This involves masking the data around k by k-bits (5 in binary = 31).

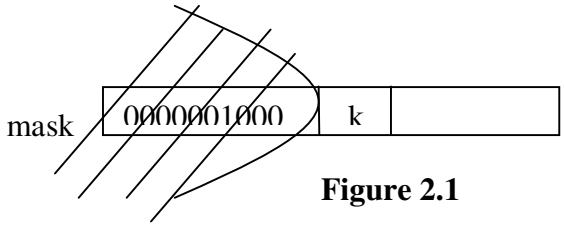



**Figure 2.1**

k = (in >> n-bit ) & 31 (& 31 picks out the value of k, but eliminates the data in front of it)

Our next step is to scan the MSB/FS values consecutively until we find a 1. This signifies the beginning of the LSB, and the end of the FS. When a 1 is found, the MSB, then the LSB is decompressed. The reverse of the k-th bit splitting is done by adding the MSB and LSB.

We then need to re-map all the decompressed data back to their original values (positive or negative numbers). This is the reverse of the mapping process ([2 x I + 1] if I <0 otherwise [2xI], where I is the input value). Hence re-mapping is as follows;

-[I/2 + 1] if I is odd
I/2 if I is even.

Re-mapping

| out1<br>output | out<br>input |
|---|---|
| -3 | 5 |
| -2 | 3 |
| -1 | 1 |
| 0 | 0 |

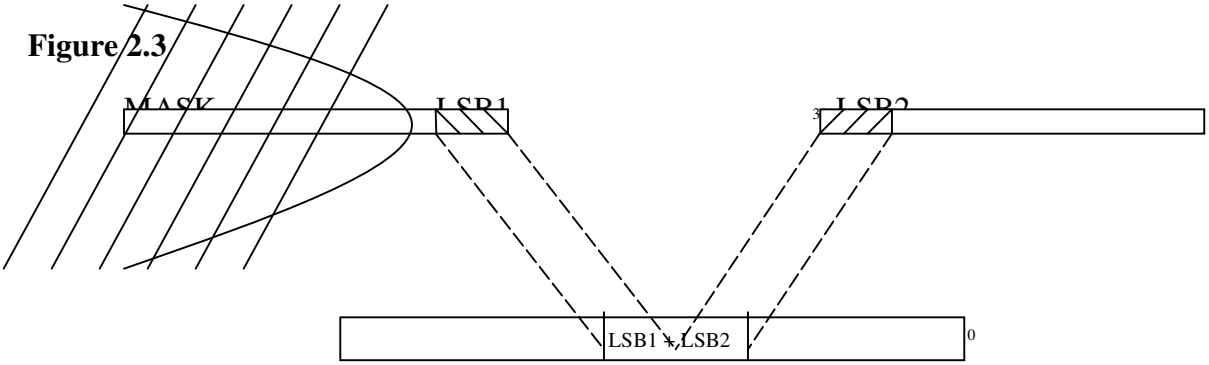
| | |
|---|---|
| 1 | 2 |
| 2 | 4 |

**Figure 2.2**

Re-map the output data to positive and negative values
if((out & 1)=0) If value is even
out1 = out >> 1; Divide output by 2
out1 = -((out >> 1)+1); If value is odd divide negative out by 2 and add 1

We also have to account for boundary problems when decompressing data as well. When extracting the LSB or k, that has crossed a boundary, we extract both segments, and add them in the output bit stream. The data values in front of the bit stream on the left, will be masked, and LSB1 extracted. We do not need the data in front of LSB1.

**Figure 2.3**




We then extract the data from LSB2 and add it to LSB1.

Code

n-bit = n-bit – n, 31 for k (n=5) and 2^k for LSB (n=k)

So if n-bit < 0: LSB1 = in << (-n-bit); (-n-bit is n-bit after we add 32- we're shifting to bit zero (<<-left shift).

n-bit = n-bit + 32 extract from new bit stream

LSB2 = in >> n-bit;

Hence LSB = LSB1 + LSB2 (combining both segments of the LSB to form the original input values)

Otherwise k or LSB is extracted and filled by (k = ((in >> n-bit) & 31), or LSB by LSB = (in >> n-bit)& mask.

**in** is the input data to be decompressed.

**Zero Suppression**

The Standard Model proposes that the Higgs field (not yet observed), fills all of space and that by interacting with this field, particles acquire their masses.

The Higgs field has at least one new particle associated with it, the Higgs particle (or Higgs boson). One of our goals to identify the Higgs boson, would help us explain more clearly the origin of mass.

The Higgs boson decays into two separate photons ($H \rightarrow \gamma\gamma$). $\pi^\circ$ decay ( $\pi^\circ \rightarrow \gamma\gamma$) also has similar products. As a result of this, we need to differentiate between pions and photons in the electromagnetic calorimeter. Identification of these photons (the products of a Higgs boson decay), will be one of our first steps to finding the Higgs boson.

The EM calorimeter consists of four layers, namely the pre-sampler, the strips, the middle and back. Analysis of the $\gamma/\pi^\circ$ separation using information of the $\eta$-strips in the EM calorimeter layout was done using samples of single photons and single $\pi^\circ$'s. The goal of the study was to investigate the impact of $\gamma/\pi^\circ$ separation of zero suppression.

Firstly the total energy in each of the four layers was computed. We then found the total energy for all events, from which the fraction $F_1$ of the shower energy deposited in the strips was calculated **(Figure 2.4).** There is a greater probability for a $\pi^\circ$ decay (two photons) to deposit significant energy in the strips, than a single photon. As a consequence we flag all showers with $F_1 < 2\%$ as photons, by performing a cut on the distribution. This cut serves to eliminate the small energies below the threshold we want to analyze. The following variables were computed for the remaining events.

(1) A search was done for the most energetic strip. The sum of energy (E3) of this strip along with its two adjacent strips was calculated. The energy contained in seven strips centered about the most energetic strip (E7) was also computed. From this the quantity Fs= (E7-E3)/E3 is found, from which a distribution showing the shower shape is plotted **(Figure 2.5).** Energies below 0.2 were flagged as photons in this case. So a cut of Fs < 0.2 was performed. This results in the elimination of more pions.

(2) Using the most energetic strip, and its two neighbors, the shower position $\eta$ (in strip units) was calculated from the formula $\eta = \sum_{i=1}^{3} \frac{E_i \eta_i}{\sum_{i=1}^{3} E_i}$. The shower width was also found for these three strips from the formula $W = \sqrt{\frac{\sum_{i=1}^{3} E_i (\eta_i - \eta)\text{^}2}{\sum_{i=1}^{3} E_i}}$. Graphs of the shower width as a function of shower position (strip units) inside the hit strip, were plotted for photons and $\pi^\circ$'s **(Figure 2.6).**

We now have a representation of the particles in the strips. Our goal is to eliminate as many pions as possible, while keeping most of the photons. To accomplish this, we got rid of all the points above the line representing the intensity of the shower of both our 25 GeV gamma and pion plots.

Next we tested the effect on photon/pion separation of zero suppression. All strip energies below a cut (ecut) were set to zero for ecut = 0, 0.025, 0.05 and 0.075 GeV **(Figure 2.7).**

The plots show that the efficiency of photon selection is greater for single photons than $\pi^{\circ}$s. As ecut is increased, there is a slight increase in photon efficiency. The increase in $\pi^{\circ}$ is larger, showing that $\pi^{\circ}$ rejection is degraded by zero suppression.

**Graphs**

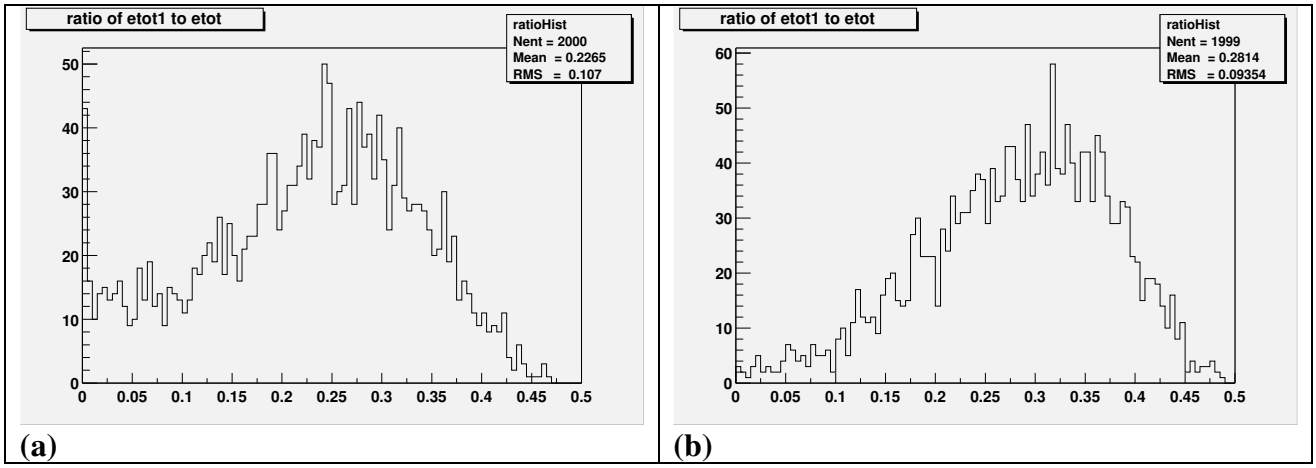



**Figure 2.4** Fraction of energy deposited in the strips of the Electromagnetic calorimeter by 25 GeV photons (a) and $\pi^{\circ}$'s (b)).

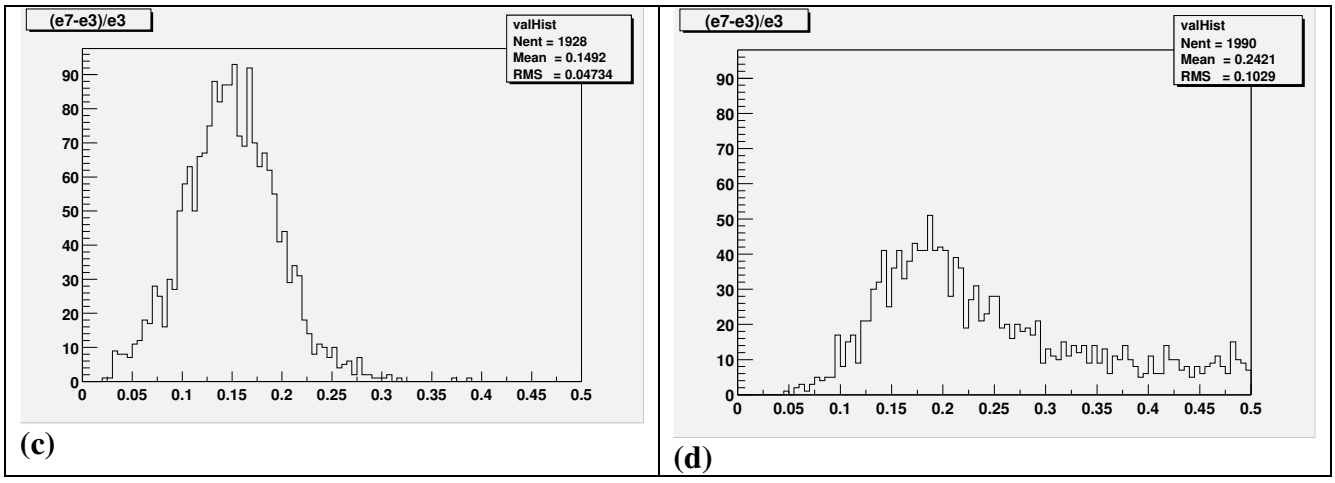



**Figure 2.5** Plots of (E7-E3)/E3 for 25 GeV photons (c) and $\pi^{\circ}$'s (d). E3 and E7 are the energies contained in three and seven strips centered on the strips with largest energy (respectively.

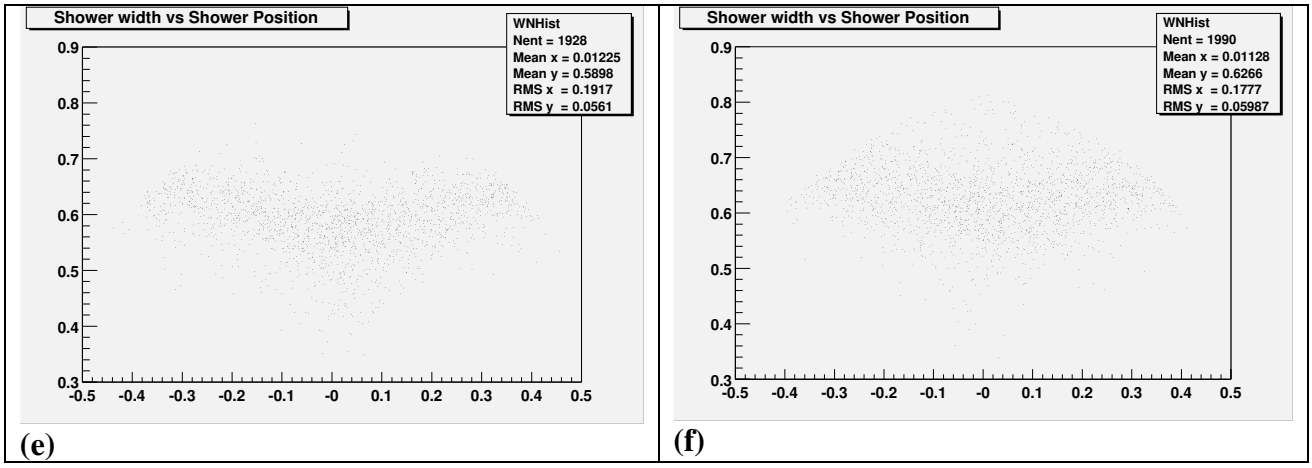



**Figure 2.6** Graphs of the shower width (W) as a function of shower position ($\eta$ in strip units) inside the hit strip for photons (e) and $\pi^{\circ}$'s (f) at 25GeV.

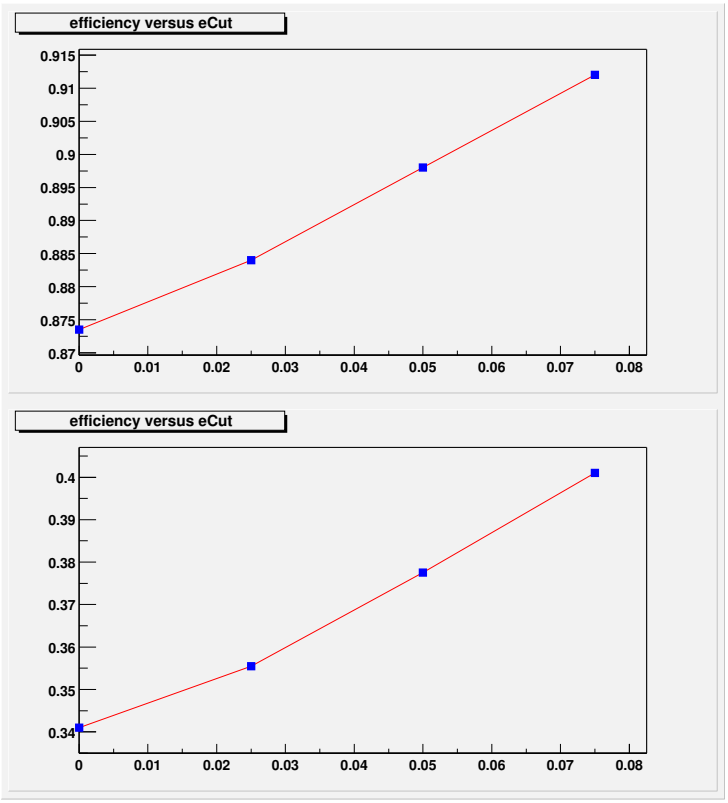



**Figure 2.7** Shows efficiency for collecting all the photons from the strips for 25GeV gamma and 25 GeV $\pi^{\circ}$'s. The process was more efficient in collecting photons.