Needles in Haystacks

Final Year Project submitted by

Rohan Chabukswar
04026007

Under the direction of

Prof. Raghav Varma
And
Prof. V. M. Gadre

Indian Institute of Technology Bombay

April 22, 2008

Department of Physics
Indian Institute of Technology Bombay
April 2008

Acceptance Certificate

Department of Physics
Indian Institute of Technology Bombay

This is to certify that this project report titled **"Needles in Haystacks"** completed and submitted by **Rohan Chabukswar** towards partial fulfilment of the requirements for the degree of **Bachelor of Technology** in **Engineering Physics** is approved by this committee.

Prof. Raghav Varma Guide	Prof. V. M. Gadre Co-guide
Examiner	Chairman

Acknowledgments

I would like to thank Prof. Raghav Varma and Prof. Vikram M. Gadre for giving me the opportunity to undertake my B. Tech. Project under their esteemed guidance and in a subject concurrent with my academic interests. I also thank Prof. Basanta Kumar Nandi for acquiring the data and explaining its format in detail. This project was extremely stimulating, and the knowledge and experience gained by undertaking it will be useful in my future life.

Rohan Chabukswar
04026007
April 22, 2008

Abstract

The project investigates the search for Quantum Chromodynamic Jets in the huge data created from simulation of these events, using wavelet analysis. Multiresolution Analysis using Haar Wavelets is applied to the data to search for jets, and a Search Algorithm is constructed from it. An in-depth scrutiny of the algorithm, including complexity analysis is also presented.

Contents

1 Quantum Chromodynamics and Strong Interaction **4**
1.1 Basics of Quantum Chromodynamics 4
1.2 Hadronization And Jets 5
1.3 The Problem 5

2 Wavelet Analysis **7**
2.1 Why Wavelets? 7
2.2 Discrete Wavelet Transform 8
2.3 Wavelet Transform In Higher Dimensions 9

3 Preliminary Analysis **10**

4 The Search Algorithm **12**
4.1 The Algorithm 12
4.2 Complexity Analysis 13

5 Simulation Results **16**

6 Limitations **19**

7 Further Work **22**

List of Figures

1.1 Jet Event 6

2.1 Level 3 Filter Bank 9
2.2 Two Dimensional Wavelet Transform 9

3.1 Optimal Resolutions in 100 Events. 11

4.1 Algorithm Flowchart 14

5.1 Simulation Image 17

6.1 Circles In $(\theta,\ \phi)$ 20
6.2 Maximum-Valued Pixels At Each Resolution 21

List of Tables

5.1 Multiresolution Tree 18

Chapter 1

Quantum Chromodynamics and Strong Interaction

1.1 Basics of Quantum Chromodynamics

The strong force, detailed by the theory of Quantum Chromodynamics, is used today to represent the interactions at the most basic level. It is a fundamental force mediated by gluons acting upon quarks, antiquarks and gluons themselves. This force acts directly only on the elementary particles. Quarks are one of the two basic constituents of matter (the other are the leptons, which, not experiencing strong interaction, need not be considered here). There are six different flavors of quarks — up, down, charm, strange, top and bottom. Since quarks are fermions, the Pauli Exclusion Principle implies that the three quarks must be in an antisymmetric combination inside a baryon. However, there has to exist another internal quantum number to account for observations of some baryons. This quantum number, given the whimsical name "color", is the charge involved in the gauge theory of quantum chromodynamics (QCD).

The other colored particle is the gluon, which is the gauge boson of QCD. QCD being a non-Abelian gauge theory, the gluons (unlike photons from quantum electrodynamics) interact with one another by the same force that affects the quarks. Color is a gauged SU(3) symmetry. Quarks come in three colors (red, green, and blue), while gluons come in eight.

Quantum Chromodynamics displays two peculiar properties:

1. Confinement, which means that the force between quarks does not diminish as they are separated. Because of this, it would take an infinite amount of energy to separate two quarks. They are forever bound into hadrons such as the proton and the neutron.

2. Asymptotic freedom, which means that in very high-energy reactions, quarks and gluons interact very weakly.

Although analytically unproven, confinement is widely believed to be true because it explains the consistent failure of free quark searches.

Asymptotic freedom can be understood qualitatively as coming from the action of the field on virtual particles carrying the relevant charge. The screening effect observed

in QED is a consequence of virtual charged particle-antiparticle pairs in vacuum. In the vicinity of the charge the vacuum becomes polarized - virtual particles of opposing charge are attracted to the charge, and the virtual particles of like charge are repelled. The net effect is to partially cancel out the field at any finite distance. Closed to the central charge, less effect of the vacuum is seen and the effective charge increases. In QCD, the same thing happens with virtual quark-antiquark pairs. They tend to screen the color charge. However, gluons themselves are colored - each gluon carries a color charge and an anti-color magnetic moment. The net effect of polarization of virtual gluons in the vacuum is not to screen the field, but to augment it and affect its color. This effect is called antiscreening. Getting closer to the quark diminishes the antiscreening effect of the surrounding virtual gluons, and this weakens the effective charge. This implies that within nucleons, quarks move mostly as free, non-interacting particles, which is termed Asymptotic Freedom.

1.2 Hadronization And Jets

A consequence of antiscreening is that the color force experience by quarks remains constant regardless of their distance from each other (after a certain point). As a direct consequence, when two quarks become separated, as in particle accelerator collisions, at some point it becomes energetically more favorable for a new quark-antiquark pair to be created spontaneously in vacuum, than to allow the quarks to separate further. Thus the quarks can never be separated, leading to confinement. When such an interaction occurs, instead of individual quarks, many color-neutral particles are detected. This process is called *hadronization*, and is one of the least understood processes in particle physics.

The tight cone of particles created by the hydronization of a single quark is called a jet. These jets must be measured in a particle detector and studied in order to determine the properties of the original quark and other elementary particles. These are simulated using PYTHIA, which is a program for the generation of high-energy physics events, including collisions at high energies between elementary particles. It contains theory and models for a number of physics aspects, including hard and soft interactions, parton distributions, initial- and final-state parton showers, multiple interactions, fragmentation and decay. PYTHIA outputs an ASCII file of the particles generated, and their momenta. Figure 1.2 shows the simulation of a jet event.

1.3 The Problem

Each such event generates particles in thousands, and hundreds of such events need to be analyzed. The volumes of data involved are huge. Moreover, these volumes will increase many times, when actual jet events are observed instead of simulations. Analyzing these massive amounts of data, even for something as simple as figuring out direction of the jets, will take years for normal analysis.

This project attempts at applying Multiresolution Analysis (MRA) using wavelets, to find the proverbial needles in the haystack. This is achieved by implementing a kind of Binary Search using the outputs of a normal Multiresolution Analysis.

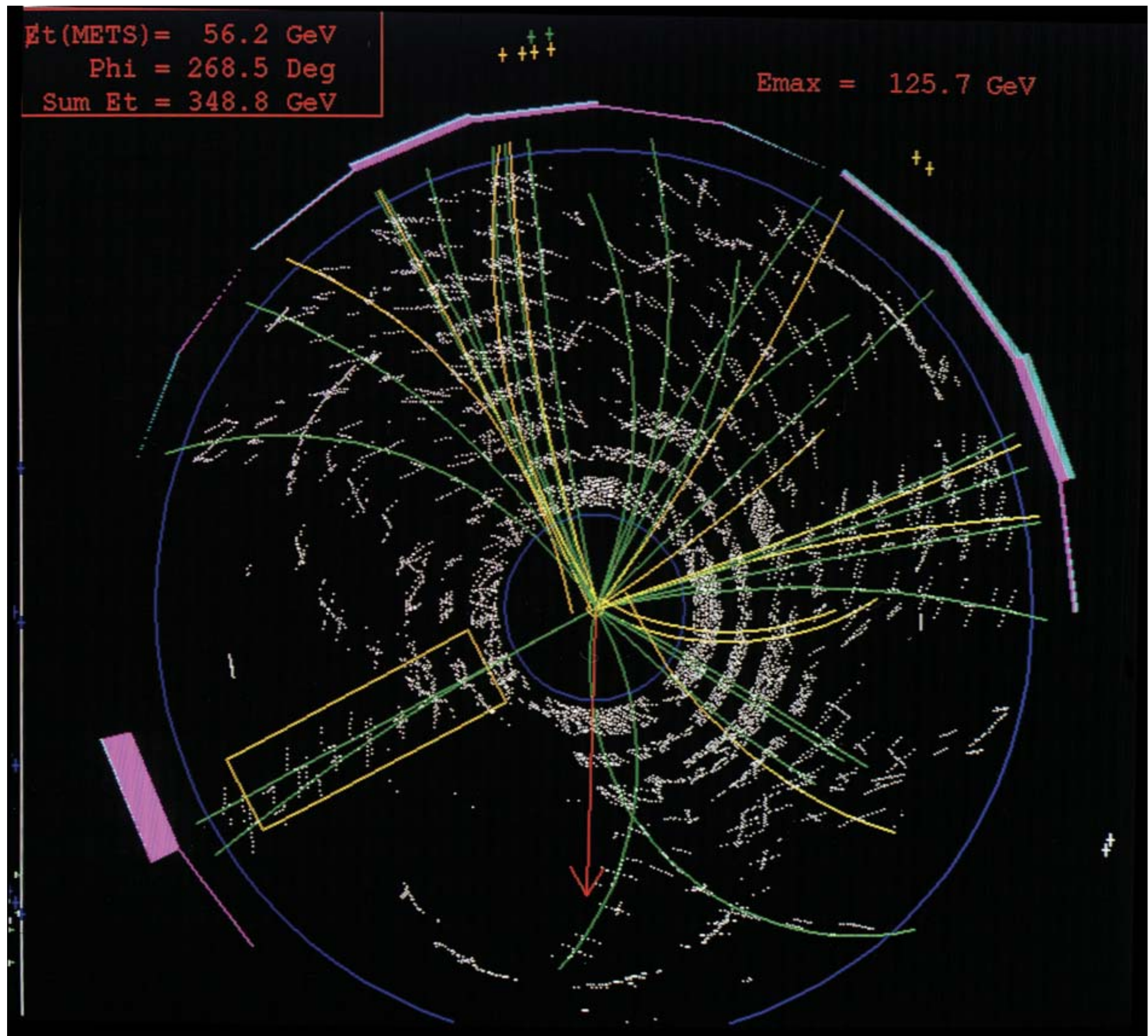


Figure 1.1: Top quark and anti-top quark pair decaying into jets, visible as collimated collections of particle tracks, in the CDF detector at Tevatron.

Chapter 2

Wavelet Analysis

A wavelet is a kind of mathematical function used to divide a given function into different frequency components and study each component with a resolution that matches its scale. A wavelet transform is the representation of a function by wavelets. The wavelets are scaled and translated copies (known as "daughter wavelets") of a finite-length or fast-decaying oscillating waveform (known as the "mother wavelet"). Wavelet transforms have advantages over traditional Fourier transforms for representing functions that have discontinuities and sharp peaks, and for accurately deconstructing and reconstructing finite, non-periodic and/or non-stationary signals. In formal terms, this transform is a wavelet series representation of a square-integrable function with respect to either a complete, orthonormal set of basis functions, or an overcomplete set of frame of a vector space (also known as a Riesz basis), for the Hilbert space of square integrable functions. Wavelet transforms are classified into discrete wavelet transforms (DWTs) and continuous wavelet transforms (CWTs). CWTs operate over every possible scale and translation whereas DWTs use a specific subset of scale and translation values.

Wavelet analysis studies each spectral component of the data with a matched resolution. Its advantage over Fourier analysis is seen analyzing physical signals with discontinuities and sharp spikes. Developed independently in mathematics, quantum physics, electrical engineering and seismic geology, wavelets now find application in image compression, turbulence, human vision, radar and earthquake prediction.

A particular set of wavelets is specified by a set of wavelet filter coefficients. The most widely used wavelet filters are the Haar Filter and the Daubechies Filters.

2.1 Why Wavelets?

Multiresolution Analysis or multiscale modeling is used in solving physical problems which have important features at multiple scales, particularly multiple spatial or temporal scales. This analysis can only be achieved by using bases which have finite spread in both time and frequency domains. Unlike sines and cosines used in Fourier Analysis, individual wavelets are quite localized in space and characteristic scale. This dual localization achievable by wavelet analysis renders many functions and operators sparse to a high accuracy. Thus there is a large class of computations, those which take advantage of sparsity, that become computationally fast in the wavelet domain.

In our example, we do not know the extent or the strengths of the jets apriori. In a

real experiment, we might not even be able to choose the specific resolution of the data. The amount of data gathered being huge, a thorough scan of the entire data will involve massive amounts of computations and data storage. Multiresolution analysis can assist in this search by searching on all resolutions at the same time.

There is no reason for choosing a particular wavelet at this stage. The Haar wavelet has been chosen for this analysis because of its conceptual simplicity and easy visualization. Other members of the Daubechies family might work just as well, as might other families.

2.2 Discrete Wavelet Transform

In discrete wavelet transforms, a given signal of finite energy is projected on a discrete subset of the upper halfplane of frequency bands (or similar subspaces of the function space $L_2(\mathbb{R})$). Then, the original signal can be reconstructed by the corresponding wavelet coefficients. One such system is the Affine system for some real parameters $a > 1$, $b > 0$. The corresponding discrete subset of the halfplane consists of all the points $(a^m,\ na^m b)$ with $m,\ n \in \mathbb{Z}$. The corresponding daughter wavelets are now given as

$$\psi_{m,n}(t) = a^{-m/2}\psi\left(a^{-m}t - nb\right). \tag{2.1}$$

A sufficient condition for the reconstruction of any signal x of finite energy by the formula

$$x(t) = \sum_{m\in\mathbb{Z}}\sum_{n\in\mathbb{Z}} \langle x\ , \psi_{m,n}\rangle \cdot \psi_{m,n}(t) \tag{2.2}$$

is that the functions $\{\psi_{m,n}|m,\ n \in \mathbb{Z}\}$ form a tight frame of $L^2(\mathbb{R})$.

The DWT of a signal is calculated by passing it through a series of filters. The signal is simultaneously passed through a low-pass and a high-pass filter, which together form a quadrature mirror filter. The filter outputs are then downsampled by 2, which can be done without data loss due to Nyquist Rule (since half the frequencies have been removed). This decomposition has halved the time resolution since only half of each filter output characterizes the signal. However, each output has half the frequency band of the input so the frequency resolution has been doubled. This decomposition is repeated to further increase the frequency resolution and the approximation coefficients decomposed with high and low pass filters and then down-sampled. This is represented as a binary tree with nodes representing a sub-space with a different time-frequency localization. Figure 2.1 shows a level 3 filter bank.

The Discrete Wavelet Transform (DWT) as an algorithm consists of applying a wavelet coefficient matrix hierarchically, first to the full data, then to the smooth part of half the length, then the smooth-smooth part of quarter the length, and so on. For example, the Haar wavelet can be considered as replacing every two consecutive data values by their average and difference (neglecting a numerical factor), and then rearranging the resulting data to have all the averages first and the differences later. The first half of the data is the smooth part, and the latter half is the detail part. This algorithm is again applied to the smooth vector to get smooth-smooth and detail-smooth parts. To invert the transform, the process is reversed, with the inverse of the matrix being used.

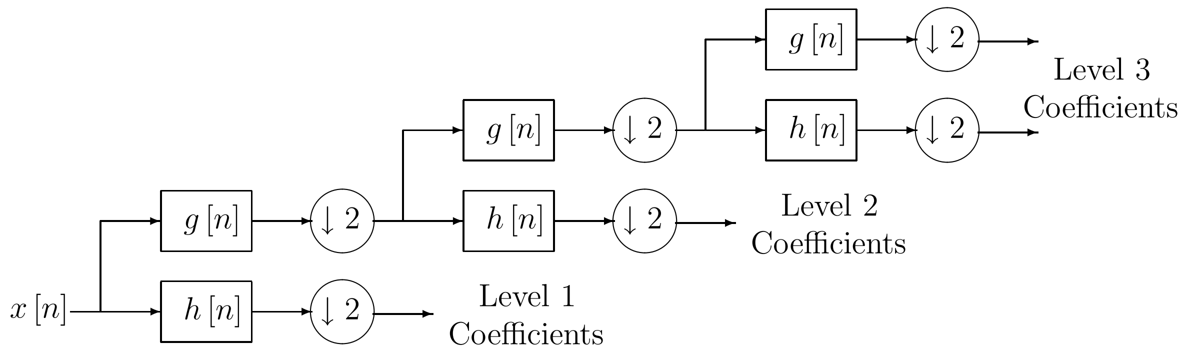


Figure 2.1: Level 3 Filter Bank

2.3 Wavelet Transform In Higher Dimensions

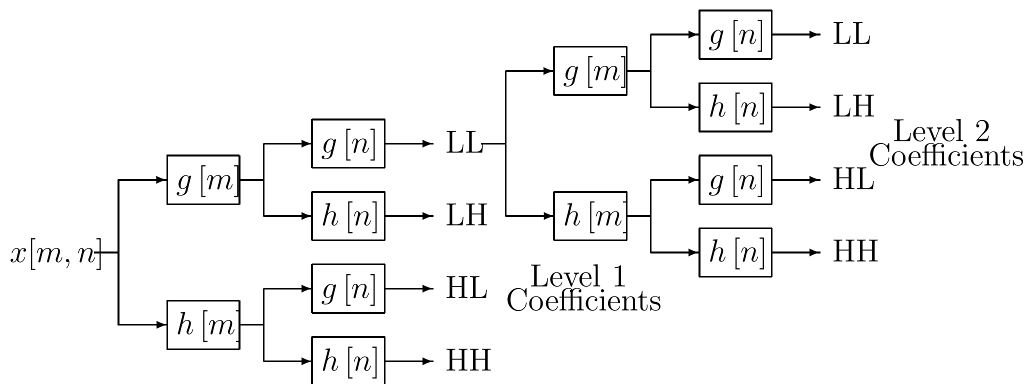


Figure 2.2: Two Dimensional Wavelet Transform

For an essentially one-dimensional wavelet, the wavelet transform of a d-dimensional array is most easily obtained by transforming the array sequentially on its first index for all values of its other indices, then on its second, and so on. Each transformation corresponds to a multiplication by an orthogonal matrix, hence by matrix associativity, the result is independent of the order in which the indices are transformed. The situation is exactly like multidimensional Fourier transforms. If the second level analysis is performed only on the LL component, the transform is called a multidimensional wavelet transform. If however, all the components are subjected to second level analysis, the transform is called the wave packet transform. Again, the levels of analysis and the indices can be interchanged without affecting the input, i.e., one particular index can be analyzed to the highest level before starting on the next one. The method of obtaining the first two levels of coefficients for two dimensional wavelet transform is shown in Figure 2.2. Again, the levels of analysis and the indices can be interchanged without affecting the input, i.e., one particular index can be analyzed to the highest level before starting on the next one.

This is not true for essentially multidimensional wavelets. However, this is not a very great concern for this application, since only one-dimensional wavelets will be used.

Chapter 3

Preliminary Analysis

The first step of the analysis is to choose the optimum resolution at which to search for the jets. In spherical co-ordinates, the jet will be represented by a higher concentration of particles in a particular area. So, we have to carry out two preliminary steps:

1. Convert the data to spherical (r, θ, ϕ) co-ordinates
2. Bin the data into appropriately sized parts and get a count of the number of particles per unit solid angle in each part, like a two dimensional histogram.

The initial resolution of the histogram should be high enough so that we can still figure out the optimum resolution for the analysis. This is arbitrarily taken to be 1024 in both θ and ϕ. However, this does not divide the space into areas subtending equal solid angles, hence while creating the histogram, the actual number of particles in that area has to be divided by the solid angle it subtends at the center. If the bin is between θ_1 and θ_2 and has a width of $\Delta\phi$, the solid angle subtended can be easily calculated to be

$$\Delta\Omega = (\cos\theta_1 - \cos\theta_2) \cdot \Delta\phi. \tag{3.1}$$

The optimum resolution can be taken to be the one where the histogram shows the most uneven distribution, or the most entropy. This can be easily implemented by using Haar wavelet analysis. First we carry out a two-dimensional wavelet transform on the histogram, down to the lowest level. This will give us, for a resolution of 1024 in each direction, 1 average component (LL), and 9 different levels with three detail components (LH, HL and HH) each. Now, the energy in each of the detail components is a measure of the entropy of the histogram at the corresponding resolution. The energy is defined as the average of the squares of all the detail components. This energy is calculated for each component, and the level with the maximum energy is outputted for each event.

The first part of the analysis is done in a separate program, as the (r, θ, ϕ) co-ordinates will have to be used for further analysis as well. Thus it is better to calculate them only once. The wavelet transform is done in another program, which also calculates the energy of each component, and the component with the maximum energy. This analysis was carried out for a random set of 100 events, and a histogram of the optimum resolutions was plotted. This histogram is given in Figure 3.1.

96% of the events showed a maximum energy at level 2, i.e., at a resolution of 4 in each direction, which gives us the native resolution of the data, or the resolution at which the data has maximum information. This will be used for further analysis later.

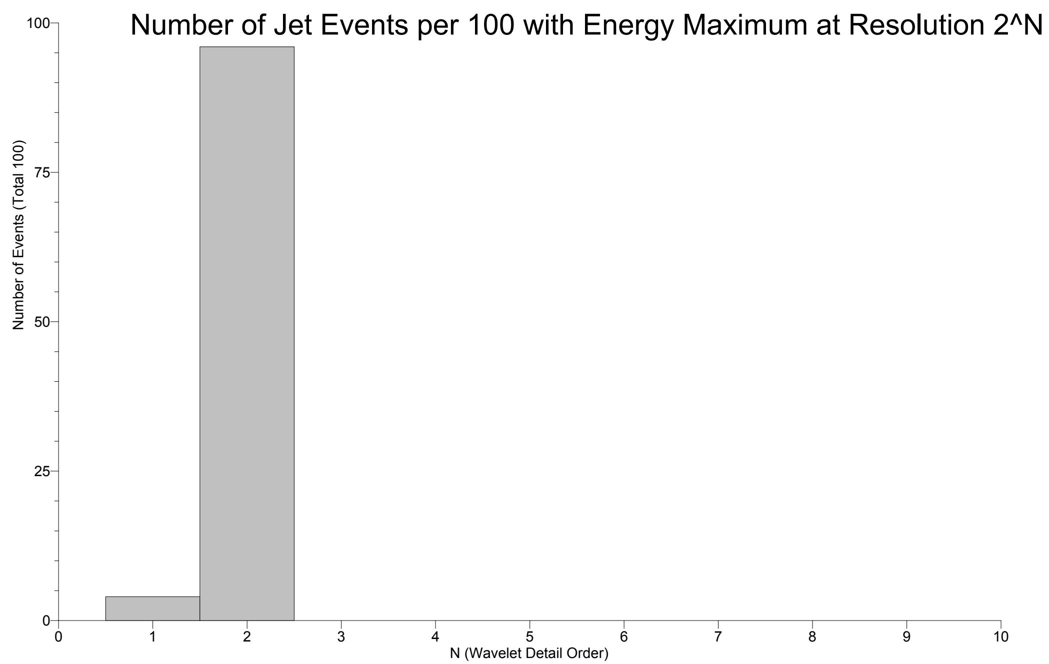


Figure 3.1: Optimal Resolutions in 100 Events.

Chapter 4

The Search Algorithm

The histogram of the data can be considered as an image, where the gray value of each pixel is the either the number of particles observed near those co-ordinates, or the energy content near those co-ordinates, which is basically the sum of the energies of the particles observed. As the algorithm can be carried out for any one or both of particles or energy, the actual quantity can be called as the gray value without being specific. We assume that the jet is conical, with the gray value being a Gaussian distribution from the center (θ_0, ϕ_0) of the jet. The exact equations for the jets will be considered later. The two Gaussians for the two antipodal jets will be overlaid on a random white gaussian noise, representing the particles detected which are not included in the jet, and for random instrumental noise.

The random white gaussian noise can have widely distributed intensities. In actual experiments, the non-jet particles will be few and far in-between, which leads to a large group of zero gray value pixels. This noise represents itself as randomly occurring white and black pixels. This noise is a form of noise typically seen on images, commonly referred to as intensity spikes, speckle or salt and pepper noise. This noise interferes with peak-detection algorithms which use differences to find local extrema. Usual and effective noise reduction method for this type of noise involves the usage of a 2-dimensional median filter, or alpha-quantile averaging. Such filters have a significant time complexity, which in fact will govern the time complexity of the entire algorithm.

These filters are a necessity to practical peak detection. The key idea here is to use the the outputs of the filters themselves to implement a sort of binary search on the data. A binary search algorithm (or binary chop) is a technique which makes progressively better guesses, and closes in on the sought value. Pursuing the strategy iteratively, it narrows the search by a factor of two each time, and finds the target value. A binary search is an example of a dichotomic divide and conquer search algorithm. While it is usually used for finding a particular value in a sorted list, its combination with a multiresolution analysis can be used to detect peaks in the image.

4.1 The Algorithm

The main difference between the algorithm and a normal MRA is that instead of keeping the differences at each tier of the analysis, the difference coefficients are discarded but the smooth components are saved instead. The output of the multiresolution analysis, which

is the first step in the algorithm, can be represented as a tree — each pixel in a lower tier corresponds to four pixels in the tier immediately above it. Thus the tree is a collection of successively poorer resolutions of the original image. The lowest element is a 2×2 image, or four pixels. The highest element is the original image itself, at whatever resolution was first selected. It is necessary that the resolution in each direction be a power of two. If the resolution cannot be chosen for any reason, it should be appropriately padded by zeros upto the next power of two. In this implementation the resolution is assumed to be 1024×1024, or 1048576 pixels.

Since we are considering only two-jet events, theoretically for conservation of momentum, the two jets have to be antipodal. If one jet is assumed to be at $(\theta,\ \phi)$, the antipodal jet will occur at $(\pi - \theta,\ \phi \pm \pi)$. For a resolution of 2×2, where the $(\theta,\ \phi)$ values can only be $(0,\ 0)$, $(0,\ 1)$, $(1,\ 0)$ and $(1,\ 1)$, the antipodal pixel is given by $(1 - \theta,\ 1 - \phi)$. Thus for the first step the pixel with the highest value is chosen, along with its antipodal pixel. The further steps are run iteratively on each of the two pixels independently.

For each step in the further analysis, chosen pixel is taken, and the four pixels it corresponds to in the next higher tier are considered. The pixel with the highest gray value in this set is the chosen pixel for the next iteration. The resolution of the location of the peak is successively doubled, down to the resolution of the original image.

The algorithm has the following advantages. The number of iterations is fixed for a particular maximum resolution. For an image of size $2^N \times 2^N$, the number of iterations is $(N - 1)$ for each jet. In each iteration, there are only a few comparisons which need to be undertaken, to find which the pixel with the maximum gray value is. As the wavelet transform itself involves a low pass filter, the noise is automatically taken care of at each resolution.

The algorithm was found to be extremely resistant to noise with signal-to-noise ratio (SNR) as low as 4. The amount of noise as such is large, but such a large noise may be experienced around $\theta = 0°$ or $\theta = 180°$, where the $(\theta,\ \phi)$ area factor is very small.

The flow chart for the algorithm is given in Figure 4.1.

4.2 Complexity Analysis

The transform implemented in the algorithm is quite similar to the wavelet transform. In fact, it is the wavelet transform exactly, with the intermediate results stored. Thus this part of the algorithm runs with a complexity same as that of a 2-dimensional wavelet transform, $\mathcal{O}(N^2)$. The complexity of the k_t jet search algorithm is $\mathcal{O}(n^3)$, where n is the number of particles. Using two-dimensional nearest neighbor location, the complexity can be reduced to $\mathcal{O}(n \ln n)$.

In a real experiment, the observations of the particles will be accrued from a fixed collection of detectors surrounding the event. This means that the output of the detectors will directly be the histogram (in numbers or energy content). Thus, one of the computationally intensive components (binning of the observed particles individually on the basis of their momenta) will drop out. The complexity of this algorithm does not depend upon the number of particles observed, but on the resolution required.

The search part of the algorithm implements a constant number of comparisons in each iteration. The number of iterations was calculated in the previous section to be

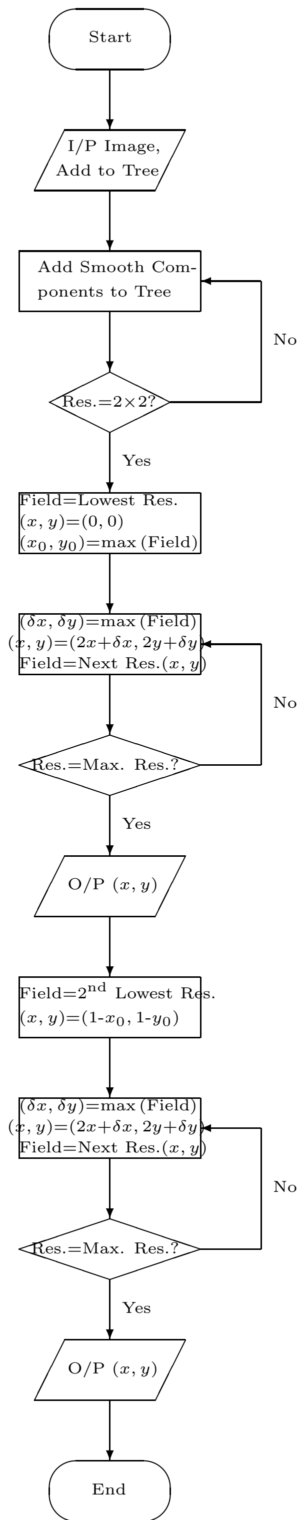


Figure 4.1: Algorithm Flowchart

$2\,(N-1)$ for an image of size $2^N \times 2^N$. Thus the complexity of the search is $\mathcal{O}\,(\ln N)$.

The total complexity of the complete algorithm will be governed by the wavelet transform and will go as $\mathcal{O}\,(N^2)$. However, in situations where the wavelet transform can be computed separately prior to analysis, say by using hardware Finite Impulse Response filters after the detectors, the search algorithm will only take $\mathcal{O}\,(\ln N)$ computations.

Chapter 5

Simulation Results

The algorithm could not be tested out on actual PYTHIA data, as no results from other algorithms on the *same* data were available for comparison of results or time complexity. To test whether the jets were actually found, all that could be done was to generate a grayscale image which resembles two antipodal jets with speckle noise, and run the algorithm on this image. The results of the algorithm could then be compared with the parameters which generated the image. This gives us an advantage in the sense that we can test the limits of the algorithm in terms of noise resistance and accuracy, independently of whether that noise would ever be present in a real image.

The data was generated as a 1024×1024 image. Thus 180° of θ are divided into 1024 parts, and 360° of ϕ into another 1024. Each pixel can have a gray value ranging from 0 through 255, which can represent particles and/or energy content. The jet is generated as a gaussian, $192 \pm (64 \cdot \text{random})$ in height, $7.5 \pm (2.5 \cdot \text{random})$ in width, located at a random point $(\theta_1,\ \phi_1) = (1024 \cdot \text{random},\ 1024 \cdot \text{random})$, where random denotes a random number between 0 and 1, with uniform distribution. Though the other jet should theoretically be antipodal, practically, it might deviate from the exact antipode due to instrumental and other errors. Thus in the simulation, the antipodal jet is generated at the approximately antipodal point $(\theta_2,\ \phi_2) = (1024 - \theta_1 \pm 64 \cdot \text{random},\ 512 + \phi_1 \pm 64 \cdot \text{random})$. There is also random white gaussian speckle noise of signal-to-noise ratio around 4, i.e. maximum strength is 25% of the average strength 192, or 48.

A sample grayscale image is show in Figure 5.1. In this, the first jet has a randomly chosen strength 175.76, random width 9.39941, and is located at (622, 614). The antipodal jet is located at (373, 89) with strength 147.86, width 5.17551.

After a negligible time interval, the algorithm calculated the jet locations to be (623, 614) and $(373, 88)$, which is very accurate for such a low SNR. These points were the highest in the jet, deviating from the actual center due to the random noise. This simulation was run for many such images, and all of the results were within ± 3 of the actual center, and a large fraction were within ± 2. Table 5.1 shows the images in the tree that were generated for the sample simulation described above.

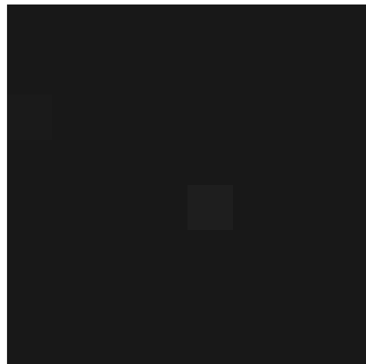
Figure 5.1: An Example Simulation Image Generated, 1024×1024.

512×512

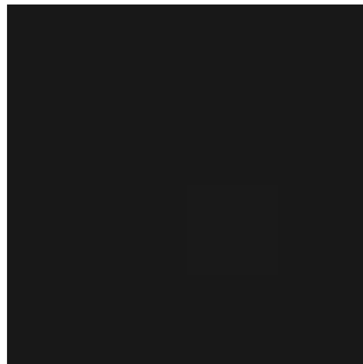
256×256

128×128

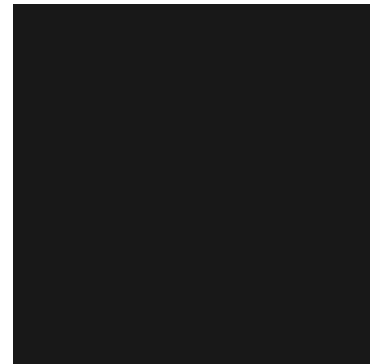
64×64 32×32 16×16



8×8



4×4



2×2

Table 5.1: The Multiresolution Tree Generated By The Search Algorithm.

Chapter 6

Limitations

One serious limitation of the algorithm is the fact that it does not delineate the jet completely or count the number of particles in the jet or their energy. This is rendered very difficult by the use of polar co-ordinates $(\theta,\ \phi)$. The jet will be a gaussian, that is, the strength of the pixel will be of the form e^{-r^2}, where r is the great circle distance of the pixel from the center of the jet, $(\theta_0,\ \phi 0)$. We can assume that the exact equation of the jet is given by

$$\text{gray value} = e^{-r^2} \cdot \sin r dr d\eta \tag{6.1}$$

where η is the angle formed by the great circle from the pixel to the center with respect to the great circle passing through the center and $\theta = 0$ or "north pole". This distribution has a circular when observed on the sphere, but however, will no longer remain circular in the $(\theta,\ \phi)$ coordinates. As an illustration, consider the circle $r = 15°$ for various θ_0 from 0 through 90. ϕ_0 can be zero without any loss of generality. Using standard equations of spherical trigonometry, it can be shown that

$$\begin{aligned} r &= \cos^{-1}\left(\cos\theta_0 \cos\theta + \sin\theta_0 \sin\theta \cos\left(\phi - \phi_0\right)\right) \\ \eta &= \tan^{-1} \frac{\sin\left(\phi - \phi_0\right)\sin\theta}{\sin\theta_0 \cos\theta - \cos\theta_0 \sin\theta \cos\left(\phi - \phi_0\right)} \end{aligned} \tag{6.2}$$

and

$$\begin{aligned} \theta &= \cos^{-1}\left(\cos r \cos\theta_0 + \sin r \sin\theta \cos\eta\right) \\ \phi &= \tan^{-1} \frac{\sin\eta \sin r}{(\sin\theta_0 \cos r - \cos\theta_0 \cos\eta \sin r} \end{aligned} \tag{6.3}$$

Thus the circle $r = 15°$ can be transformed into $(\theta,\ \phi)$ coordinates. These curves will be circles when viewed on a sphere, but as seen in Figure 6.1, in $(\theta,\ \phi)$ they do not resemble circles, except perhaps when $\theta_0 = 90°$. Indeed, near $\theta_0 = 0°$, they do not even close in on themselves. This means that the boundaries cannot be defined easily for an arbitrary jet, which excludes the usual boundary finding algorithms like the Hough Transform.

A different approach was tried for trying to find atleast approximately the extent of the jets. The idea used was as follows. As long as the jet extent is less than the current pixel size, the value of the maximum pixel, which is the average of all the (maximum resolution) pixels containing the jet, will continue to increase by an approximate factor of 4. As soon as the jet extent exceeds one pixel, the average will remain approximately

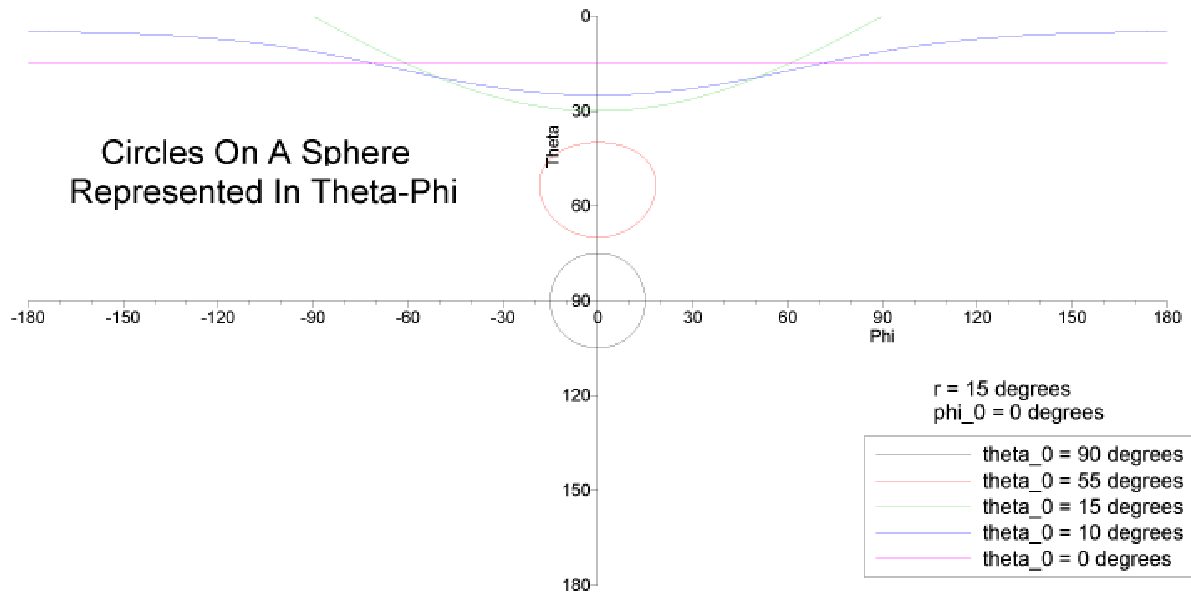


Figure 6.1: Circles On A Sphere Represented In $(\theta,\ \phi)$.

constant. Thus plotting the gray values of the maximum-valued pixels at each resolution will give us the resolution at which the jet starts exceeding 1 pixel. This will give the extent of the jet to the nearest power of two. While not terribly useful, further analysis can be carried out at a finer resolution to find the jet boundary. The results for the above simulation are shown in Figure 6.2. It can be figured out that the extent of the first jet is roughly 32×32 pixels and that for the second it is roughly 16×16.

Currently a brute force algorithm to find the boundary, which finds the pixels with a specific strength (with respect to the peak) nearest to the peak, has not been implemented in the hope that a better algorithm can be figured out. This algorithm will be prone to noise, but the native resolution found out in stage one can be used here to apply a low pass filter of size 4×4, which will reduce the noise level.

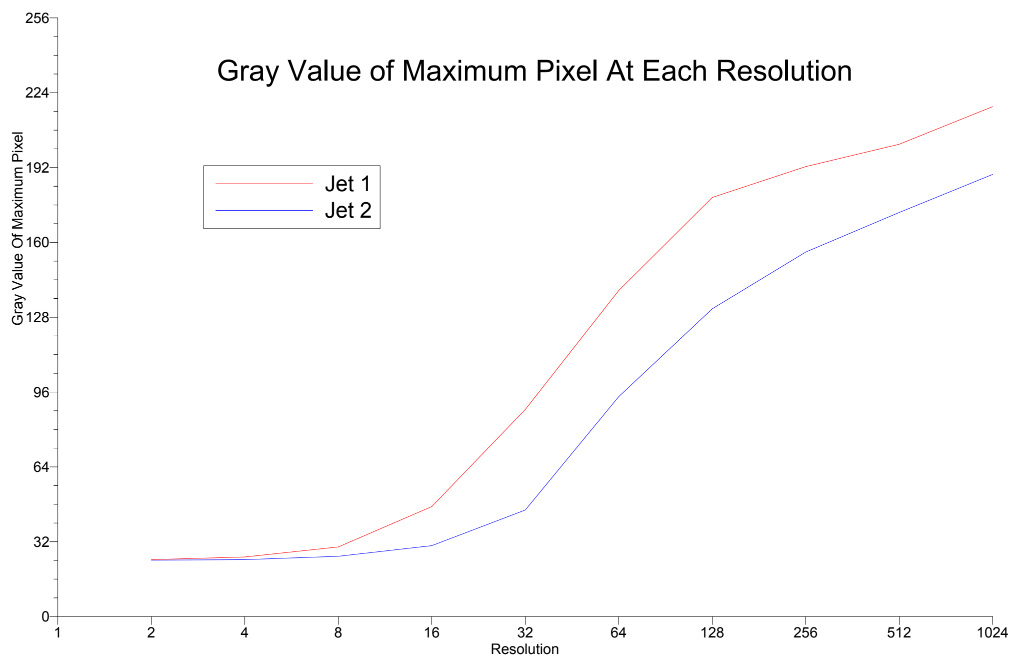


Figure 6.2: Gray Values Of Maximum-Valued Pixels At Each Resolution.

Chapter 7

Further Work

Further work will hopefully come up with a good algorithm to find the boundaries of the jets. Once the boundaries are figured out, it is easy to find the number of particles in each jet, along with the total energy of the jet.

Another direction for work will be using a wavelet other than the one used, say DAUB4 or DAUB20, or a different family like Morlet. The use of Biorthogonal and Continuous Wavelet Transforms might also be explored, though they will almost certainly require extensive computation.

Appendix

Codes

The analysis was done using C++ Codes. MATLab was also used for some trials.

The final search algorithm code is given below. It also includes the generation of the simulation image as detailed above. The output is given on standard output.

```
#include<iostream>
#include<fstream>
#include<vector>
#include<cmath>
#include<ctime>
using namespace std;

vector<double> daub1(vector<double> a, int n, int isign)
{
  double C=0.5;
  int nh,i,j;
  vector<double> wavelet=a;
  if(n < 2)
    return wavelet;
  nh=n>>1;
  if(isign>=0)
    for(i=0,j=0;j<n-1;j+=2,i++)
      {
        wavelet[i]=C*a[j]+C*a[j+1];
        wavelet[i+nh]=C*a[j]-C*a[j+1];
      }
  else
    {
      for(i=0,j=0;i<nh;i++)
        {
          wavelet[j++]=C*a[i]+C*a[i+nh];
          wavelet[j++]=C*a[i]-C*a[i+nh];
        }
    }
  return wavelet;
}
```

```
int swap(double &a, double &b)
{
  double temp=a;
  a = b;
  b = temp;
  return 0;
}

vector<vector<vector<double> > > binarysearch
(vector<vector<double> > image,int resolution)
{
  vector<vector<vector<double> > > answer;
  for(;resolution>2;resolution>>=1)
    {
      answer.push_back(image);
      vector<vector<double> > temp=image;
      vector<double> dummy;
      for(int i = 0;i < resolution;i++)
temp[i]=daub1(image[i],resolution,1);
      for(int i=0;i<resolution;i++)
for(int j=0;j<i;j++)
  swap(temp[i][j],temp[j][i]);
      for(int i=0;i<resolution;i++)
temp[i]=daub1(temp[i],resolution,1);
      for(int i=0;i<resolution;i++)
for(int j=0;j<i;j++)
  swap(temp[i][j],temp[j][i]);
      image.clear();
      for(int i = 0;i < resolution/2;i++)
{
  dummy.clear();
    for(int j = 0;j < resolution/2;j++)
      dummy.push_back(temp[i][j]);
  image.push_back(dummy);
}
      temp.clear();
    }
  answer.push_back(image);
  string filename = "tree";
  string number = "0123456789";
  string extension = ".txt";
  for(int i = 0;i < answer.size();i++)
    {
      ofstream fout((filename + number[i] + extension).c_str());
      for(int j = 0;j < answer[i].size();j++)
```

```
{
  for(int k = 0;k < answer[i][j].size() - 1;k++)
    fout<<answer[i][j][k]<<"\t"<<flush;
  fout<<answer[i][j][answer[i][j].size() - 1]<<endl;
  fout<<endl;
}
      fout.close();
    }
  return answer;
}

int max(vector<vector<double> > field, int &x, int &y)
{
  if(field[0][0]>=field[0][1]&&field[0][0]>=field[1][0]
&&field[0][0]>=field[1][1])
    {
      x = 0;
      y = 0;
    }
  else
    {
      if(field[0][1] >= field[1][0] && field[0][1] >= field[1][1])
{
  x = 0;
  y = 1;
}
      else
{
  if(field[1][0] >= field[1][1])
    {
      x = 1;
      y = 0;
    }
  else
    {
      x = 1;
      y = 1;
    }
}
    }
  return 0;
}

int main()
{
  int resolution = 1024;
```

```
  double shiftmax = 64;
  double signal = 192;
  double variance = 64;
  double width = 7.5;
  double widthvariance = 2.5;
  double snr = 0.25;
  vector<int> dummy;
  vector<double> dummydouble;
  vector<vector<double> > image;
  vector<vector<vector<double> > > tree;
  srand(time(0));
  int x1 = (int) resolution*((double) rand()/RAND_MAX);
  int y1 = (int) resolution*((double) rand()/RAND_MAX);
  int x, y, x2, y2;
  vector<double> value;
  x2 = resolution - x1 - 1;
  y2 = (resolution / 2 + y1) % resolution;
  x2 += ((int) 50*(1 - 2 * (double) rand() /  RAND_MAX));
  x2 %= resolution;
  y2 += ((int) 50*(1 - 2 * (double) rand() /  RAND_MAX));
  y2 %= resolution;
  double signal1=signal+(1-2*(double)rand()/RAND_MAX)*variance;
  double signal2=signal+(1-2*(double)rand()/RAND_MAX)*variance;
  double width1=width+(1-2*(double)rand()/RAND_MAX)*widthvariance;
  double width2=width+(1-2*(double)rand()/RAND_MAX)*widthvariance;
  for(int i = 0;i < resolution;i++)
    {
      for(int j = 0;j < resolution;j++)
dummydouble.push_back((double)rand()/RAND_MAX*signal*snr
+signal1*exp(-(double)((i-x1)*(i-x1)+(j-y1)*(j-y1))/(2*width1*width1))
+signal2*exp(-(double)((i-x2)*(i-x2)+(j-y2)*(j-y2))/(2*width2*width2)));
      image.push_back(dummydouble);
      dummydouble.clear();
    }
  cout<<x1<<"\t"<<y1<<"\t"<<signal1<<"\t"<<width1<<endl;
  cout<<x2<<"\t"<<y2<<"\t"<<signal2<<"\t"<<width2<<endl;
  cout<<endl;
  tree = binarysearch(image, resolution);
  dummydouble.clear();
  dummydouble.push_back(0);
  dummydouble.push_back(0);
  vector<vector<double> > field;
  field.push_back(dummydouble);
  field.push_back(dummydouble);
  x = 0;
  y = 0;
```

```
  double x0,y0;
  int dx,dy;
  value.clear();
  for(int i = tree.size() - 1;i >= 0;i--)
    {
      x *= 2;
      y *= 2;
      field[0][0] = tree[i][x][y];
      field[0][1] = tree[i][x][y+1];
      field[1][0] = tree[i][x+1][y];
      field[1][1] = tree[i][x+1][y+1];
      max(field,dx,dy);
      value.push_back(field[dx][dy]);
      x += dx;
      y += dy;
      if(i == tree.size() - 1)
{
  x0 = x;
  y0 = y;
}
    }
  for(int i = 0;i < value.size();i++)
    cout<<value[i]<<endl;
  cout<<endl;
  value.clear();
  cout<<x<<"\t"<<y<<endl;
  x = 1 - x0;
  y = 1 - y0;
  value.push_back(tree[tree.size()-1][x][y]);
  for(int i = tree.size() - 2;i >= 0;i--)
    {
      x *= 2;
      y *= 2;
      field[0][0] = tree[i][x][y];
      field[0][1] = tree[i][x][y+1];
      field[1][0] = tree[i][x+1][y];
      field[1][1] = tree[i][x+1][y+1];
      max(field,dx,dy);
      value.push_back(field[dx][dy]);
      x += dx;
      y += dy;
    }
  for(int i = 0;i < value.size();i++)
    cout<<value[i]<<endl;
  cout<<endl;
  cout<<x<<"\t"<<y<<endl;
```

```
  return 0;
}
```

The output of the above program for the sample simulation is given below.

```
622 614 175.76 9.39941
373 89 147.86 5.17551

24.3478
25.4744
29.7486
47.0273
88.5513
139.412
179.234
192.402
201.969
218.072

623     614

24.1052
24.3333
25.7749
30.3157
45.5425
94.0369
131.601
155.862
172.805
189.067

373     88
```

The next code converted the data from the file from Cartesian to spherical co-ordinates. The program converts the co-ordinates, and adds r, θ and ϕ values as three separate columns, while maintaining the previous data. The data file has to be redirected into the program by standard input and the processed data is given on standard output, which should be redirected into another file. The program gives its status on standard error.

```
#include<iostream>
#include<fstream>
#include<string>
#include<vector>
#include<cmath>
using namespace std;

vector<double> parse(string line)
```

```
{
  vector<double> xyz;
  int i;
  string temp;
  for(i = 0;i != -1;)
    {
      i=line.find("\t",0);
      temp=line.substr(0,i);
      xyz.push_back(atof(temp.c_str()));
      line=line.substr(i+1);
    }
  xyz.push_back(atof(line.c_str()));
  return xyz;
}

vector<double> convert(vector<double> xyz)
{
  vector<double> rthetaphi;
  rthetaphi.push_back(xyz[0]);
  double x=xyz[1];
  double y=xyz[2];
  double z=xyz[3];
  double pi=4*atan(1);
  rthetaphi.push_back(sqrt(x*x+y*y+z*z));
  rthetaphi.push_back(pi/2-asin(z/rthetaphi[1]));
  if(x != 0)
    rthetaphi.push_back(atan2(y,x));
  else
    if(y < 0)
      rthetaphi.push_back(-pi/2);
    else if(y > 0)
      rthetaphi.push_back(pi/2);
    else
      rthetaphi.push_back(0);
  return rthetaphi;
}

int main()
{
  cerr.precision(4);
  string line;
  vector<double> xyz;
  vector<double> rthetaphi;
  for(unsigned int i = 0;i < 127792;i++)
    {
      cerr<<"\b\b\b\b\b\b\b\b"<<double(i)/127792 * 100<<flush;
```

```
      cout<<"% done"<<flush;
      getline(cin, line);
      if(line.substr(0,5)=="Event")
        {
          cout<<endl<<line<<endl;
          getline(cin, line);
          getline(cin, line);
          continue;
        }
      else if(line=="")
        continue;
      else
        {
          xyz=parse(line);
          rthetaphi=convert(xyz);
          cout<<rthetaphi[0]<<"\t"<<flush;
          cout<<xyz[1]<<"\t"<<xyz[2]<<"\t"<<flush;
          cout<<xyz[3]<<"\t"<<flush;
          cout<<rthetaphi[1]<<"\t"<<rthetaphi[2]<<flush;
          cout<<"\t"<<rthetaphi[3]<<endl;
        }
    }
  cerr<<"\b\b\b\b\b\b\b\b"<<"Done!"<<endl;
  return 0;
}
```

An excerpt from the output of this program is given below.

```
Event No.:14 No. of particles:1355 No. of Jets:2
21 1.35196 -0.679553 828.338 828.339 0.00182671 -0.46576
1 -1.03581 -0.878458 -1158.27 1158.27 3.14042 -2.43821
21 29.7538 -16.7666 64.5092 72.9921 0.486909 -0.513157
21 8.7499 -4.66825 -187.677 187.939 3.0888 -0.490103
21 -38.761 -77.7077 -139.146 164.02 2.58365 -2.03349
21 77.2648 56.2728 15.978 96.9112 1.40517 0.629475
1 -5.7006 1.59142 -702.788 702.813 3.13317 2.86936
21 0.155065 -1.37802 -3.30233 3.58167 2.74403 -1.45874
:
22 0.00130509 -0.00443941 -0.00104797 0.00474446 1.79352 -1.28487
11 0.000759893 -0.00151122 0.000210186 0.00170452 1.44717 -1.10488

Event No.:26 No. of particles:774 No. of Jets:2
2 -0.144125 -0.0830878 3357.7 3357.7 4.95458e-05 -2.61863
2 1.44665 -1.00189 -2295.66 2295.66 3.14083 -0.605714
21 13.6096 -4.8431 58.302 60.065 0.242881 -0.341885
2 -20.1933 23.6538 -985.371 985.862 3.11004 2.27744
21 -75.1177 70.3608 11.3948 103.553 1.46053 2.38888
```

```
2 68.5341 -51.5501 -938.464 942.374 3.05047 -0.644896
2 0.595598 0.850352 4.98365 5.09064 0.205382 0.959793
21 1.47263 5.41869 -6.41188 8.52309 2.42234 1.30544
:
```

The next code written was to analyze the data processed by the above program. This program creates a histogram of the data in θ and ϕ at a resolution of 1024 in each angle, and does a Haar Multi-Resolution Analysis, separately for each event. The energies in each of the nine the detail components of the MRA are evaluated, and are outputted for further analysis. Also, the level where the maximum energy occurs is given in the last column. Again, the file is given to the program through standard input, and the output and status are available on standard output and standard error respectively.

```
#include<iostream>
#include<fstream>
#include<vector>
#include<cmath>
using namespace std;

vector<double> parse(string line)
{
  vector<double> data;
  int i;
  string temp;
  for(i = 0;i != -1;)
    {
      i=line.find("\t",0);
      temp=line.substr(0,i);
      data.push_back(atof(temp.c_str()));
      line=line.substr(i+1);
    }
  data.push_back(atof(line.c_str()));
  return data;
}

vector<double> daub1(vector<double> a, int n, int isign)
{
  double C=sqrt(0.5);
  int nh,i,j;
  vector<double> wavelet=a;
  if(n < 2)
    return wavelet;
  nh=n>>1;
  if(isign>=0)
    for(i=0,j=0;j<n-1;j+=2,i++)
      {
        wavelet[i]=C*a[j]+C*a[j+1];
```

```
        wavelet[i+nh]=C*a[j]-C*a[j+1];
      }
  else
    {
      for(i=0,j=0;i<nh;i++)
        {
          wavelet[j++]=C*a[i]+C*a[i+nh];
          wavelet[j++]=C*a[i]-C*a[i+nh];
        }
    }
  return wavelet;
}

vector<double> wavetrans(vector<double> a, int isign)
{
  int nn;
  int n=a.size();
  vector<double> transform=a;
  if(isign>=0)
    for(nn=n;nn>=2;nn>>=1)
      transform=daub1(transform,nn,isign);
  else
    for(nn=2;nn<=n;nn<<=1)
      transform=daub1(transform,nn,isign);
  return transform;
}

int swap(double &a, double &b)
{
  double temp=a;
  a = b;
  b = temp;
  return 0;
}

int max(vector<double> energy)
{
  int currentmax = 0;
  for(int i = 1;i < energy.size();i++)
      if(energy[i] >= energy[currentmax])
currentmax=i;
  return currentmax+1;
}

int main()
{
```

```
string line;
int bintheta=1024,binphi=1024,particles=0;
double pi=4*atan(1),factor,sum=0;
double binunittheta=pi/bintheta,binunitphi=2*pi/binphi;
vector<int> dummy;
vector<vector<int> > histogram;
vector<double> dummydouble, energy, data;
vector<vector<double> > histdoub, transform;
transform.clear();
for(int i = 0;i < binphi;i++)
  dummy.push_back(0);
for(int i = 0;i < binphi;i++)
  dummydouble.push_back(0);
for(int i = 0;i < bintheta;i++)
  histogram.push_back(dummy);
for(int i = 0;i < bintheta;i++)
  histdoub.push_back(dummydouble);
for(int count = 0;count < 100;)
  {
    getline(cin,line);
    if(line=="")
      {
        count++;
        particles=0;
        for(int i = 0;i < bintheta;i++)
          for(int j = 0;j < binphi;j++)
            particles+=histogram[i][j];
        for(int i = 0;i < bintheta;i++)
          {
            factor=cos(i*binunittheta);
            factor-=cos((i+1)*binunittheta);
            factor*=(2*pi);
            for(int j = 0;j < binphi;j++)
              {
                double temp=factor*particles;
                histdoub[i][j]=(double)histogram[i][j];
histdoub[i][j]/=temp;
              }
          }
        for(int i=0;i<1024;i++)
          transform.push_back(wavetrans(histdoub[i],1));
        for(int i=0;i<1024;i++)
          for(int j=0;j<i;j++)
            swap(transform[i][j],transform[j][i]);
        for(int i=0;i<1024;i++)
          transform[i]=wavetrans(transform[i],1);
```

```
          for(int i=0;i<1024;i++)
            for(int j=0;j<i;j++)
              swap(transform[i][j],transform[j][i]);
          for(int k = 2;k<=1024;k*=2)
            {
              sum = 0;
              for(int i = 0;i < k;i++)
                for(int j = 0;j < k;j++)
                  if((i >= k / 2) || (j >= k / 2))
                    sum += transform[i][j] * transform[i][j];
              sum /= ((double) 3*k*k/4);
              energy.push_back(sum);
            }
          for(int i = 0;i < 9;i++)
            cout<<energy[i]<<"\t"<<flush;
          cout<<max(energy)<<endl;
          energy.clear();
          transform.clear();
          histogram.clear();
          for(int i = 0;i < bintheta;i++)
            histogram.push_back(dummy);
          histdoub.clear();
          for(int i = 0;i < bintheta;i++)
            histdoub.push_back(dummydouble);
          cerr<<"\b\b\b\b\b\b\b\b\b"<<count<<"% done"<<flush;
        }
      else if(line[0]=='E')
        continue;
      else
        {
          data=parse(line);
          int x = (int) (data[5]/binunittheta);
          int y = (int) ((pi+data[6])/binunitphi);
          histogram[x][y]++;
        }
    }
  cerr<<"\b\b\b\b\b\b\b\b\b100% done."<<endl;
  return 0;
}
```

An excerpt of the output from this program is given below. The nine columns give the energies in each of the detail components, smoothest first.

```
0.0361 0.1633 0.0787 0.0439 0.0294 0.0225 0.0163 0.0172 0.0148 2
0.5158 0.5584 0.2988 0.1590 0.1225 0.0878 0.0630 0.0632 0.0487 2
0.0806 0.4331 0.2459 0.1451 0.0815 0.0479 0.0364 0.0375 0.0368 2
0.2353 0.5628 0.2988 0.1591 0.0949 0.0637 0.0467 0.0451 0.0408 2
```

```
0.0046 0.1882 0.0951 0.0543 0.0324 0.0237 0.0203 0.0191 0.0192 2
0.0112 0.1309 0.1025 0.0615 0.0398 0.0315 0.0278 0.0278 0.0236 2
0.0763 0.1215 0.0826 0.0444 0.0145 0.0092 0.0060 0.0055 0.0052 2
0.1333 0.2540 0.1366 0.0886 0.0497 0.0387 0.0350 0.0254 0.0278 2
:
0.0086 0.1583 0.0818 0.0444 0.0275 0.0170 0.0148 0.0146 0.0147 2
0.1564 0.3638 0.2685 0.1525 0.1014 0.0730 0.0635 0.0626 0.0529 2
```

Data Files

The data is given as an ASCII file with four columns. The first column contains the particle ID, unique for each particle, from which we can determine the physical properties of the particle like mass, charge, etc. The remaining three columns contain the momentum of the said particle in X, Y and Z directions respectively. The data file contains one hundred random events, each given a number, and each with a different number of particles, but all with only 2 jets. The event number, the number of particles and the number of jets is indicated for each event, just before the data for the event begins. An excerpt from the data file is shown below.

```
Event No.:14 No. of particles:1355 No. of Jets:2
21 1.35196 -0.679553 828.338
1 -1.03581 -0.878458 -1158.27
21 29.7538 -16.7666 64.5092
21 8.7499 -4.66825 -187.677
21 -38.761 -77.7077 -139.146
21 77.2648 56.2728 15.978
1 -5.7006 1.59142 -702.788
21 0.155065 -1.37802 -3.30233
:
22 0.00130509 -0.00443941 -0.00104797
11 0.000759893 -0.00151122 0.000210186

Event No.:26 No. of particles:774 No. of Jets:2
2 -0.144125 -0.0830878 3357.7
2 1.44665 -1.00189 -2295.66
21 13.6096 -4.8431 58.302
2 -20.1933 23.6538 -985.371
21 -75.1177 70.3608 11.3948
2 68.5341 -51.5501 -938.464
2 0.595598 0.850352 4.98365
21 1.47263 5.41869 -6.41188
:
```

Bibliography

[1] Graps, Amara, An Introduction to Wavelets, *IEEE Computational Science and Engineering*, **vol. 2, num. 2**, 1995.

[2] *Numerical Recipes in C++ - The Art of Scientific Computing*, 2nd Ed., William H. Press, Saul A Teukolsky, William T. Vatterling, Brian P. Flannery, Cambridge University Press, 2002.

[3] HyperPhysics Concepts `http://hyperphysics.phy-astr.gsu.edu/hbase/particles/parcon.html`.

[4] Wikipedia `http://en.wikipedia.org/`.

[5] PYTHIA Website `http://www.thep.lu.se/~torbjorn/Pythia.html`