Transverse momentum of Z^0 bosons and associated $W^{\pm}$ mass uncertainty

Daniel Beecher

Supervisor:
Dr. Mark Lancaster

March 26, 2004

Abstract

An investigation into the effect that quantum chromodynamics, such as gluon emission, has upon the transverse momentum of Z^0 bosons produced in $p\bar{p}$ collisions.

A simulation of the CDF's Central Electromagnetic (CEM) calorimeter was written. HERWIG events were run through the simulation to investigate how the energy resolution of the CEM affect the production of Z^0 bosons. It was found that with increased κ the peak of the $P_T(Z)$ increased. The affect of κ was found to be significant, and the value of κ was derived by means of fitting mass distributions to M_Z data from the CDF. κ was calculated at 1.5065 ± 0.2368; the χ^2 for the fit was 123.383 with 138 degrees of freedom, corresponding to passing a 80% significance test.

A functional form was chosen to represent $P_T(Z)$, run through the detector simulation, and then compared with $P_T(Z)$ data from the CDF. The parameters of the best fit were: P_1 equal to 0.1311 ± 0.0103; P_2 being 0.0302 ± 0.0262; P_3 equal to 6.8299 ± 0.4895; P_4 being 0.6070 ± 0.0607. The χ^2 of the fit was 38.9175 with 46 degrees of freedom, passing a 76% significance test.

Contents

List of Figures

List of Tables

Project Aims

The aims of the project were as follows:

- Understand the effect the gluon radiation has upon the kinematics of $W^{\pm}$ and Z^0 production.
- Derive a form for the transverse momentum of Z^0 by fitting experimental data with that of a detector simulation, and comparing it with the latest theoretical models.
- Estimate how the uncertainty in the $W^{\pm}$ mass is affected by the transverse momentum of Z^0

Acknowledgements

I must give thanks to my supervisor, Dr. Mark Lancaster, who's knowledge and help proved invaluable.

1 Background Theory

1.1 $W^{\pm}$ and Z^0 bosons

1.1.1 The Standard Model

The Standard Model is the currently accepted theory explaining the nature of sub-atomic particles, and their interactions. It allows for three types of particle: leptons, quarks and force mediators (gauge bosons). The model came into being during the late 1970's as a combination of all the previous work into elementary particle physics during the last century.

As a result of their experiments, physicists believe that there are four fundamental forces, three of which can be explained using the Standard Model: the electromagnetic force, the strong force, and the weak force. The mediators of the weak force are the $W^{\pm}$ bosons and the Z^0 boson.

The Standard Model does not account for gravity, however, but it is hoped that with greater research that it may one day be tied with the other three fundamental forces.

1.1.2 Importance of $W^{\pm}$ mass

The mass of the $W^{\pm}$ is an important test of the validity of the Standard Model. Yet, an *a priori* prediction of the $W^{\pm}$ mass is not possible, it must be inferred from the masses of other particles such as top quark. Accurate measurements of the top mass and $W^{\pm}$ mass would make the constraints upon things such as the Higgs mass, smaller. The current measurement of the $W^{\pm}$ mass is 80.412 ± 0.0060 GeV, yet the particle physicists want to reduce the uncertainty to less than 30 MeV; such accuracy is not currently possible with available detectors. An alternative approach is to model the $W^{\pm}$ mass in a detector simulation, and then compare this with the experimental results.

Figure 1 shows how the $W^{\pm}$ and top masses affect the Higgs mass predictions. Should the uncertainty in the masses of the the $W^{\pm}$ and top be reduced, then the Higgs mass bands would narrow. Thus, the more precise the measurement of these masses, the more precise we hope we can be about Higgs mass predictions. There are various Higgs mass predictions as a result of work conducted by various collaborations and experiments. The DØ and CDF experiments, both at the Fermilab, collide protons and anti-protons at the Tevatron. NuTeV is another Fermilab experiment, which investigates neutrino-neutron scattering. LEP2 was an accelerator at CERN, which collided electrons and positrons; one of the results from the its OPAL experiment was a good measurement of the $W^{\pm}$ mass, which is indicated. The Minimal Supersymmetric Standard Model (MSSM) predictions are also shown, this is the Standard Model prediction, but modified with the assumption that the hypothetical supersymmetric particles exist.

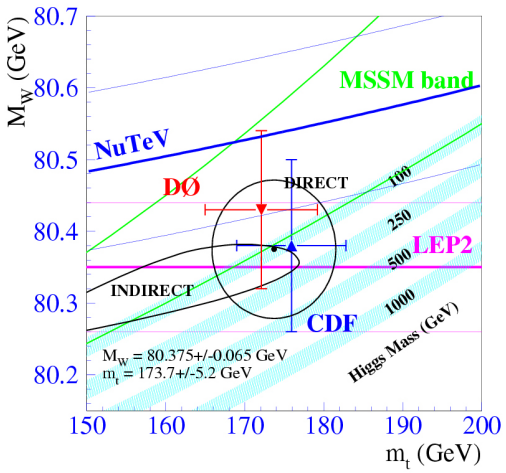


Figure 1: Predicted Higgs mass for values of the top and $W^{\pm}$ mass. Limits and predictions as a result of various experiments are also shown (Image credit: Fermi National Accelerator Laboratory).

1.1.3 Glashow-Weinberg-Salam Theory

The Glashow-Weinberg-Salam (GWS) theory is the current model for explaining the nature of weak interactions. One crucial result of the GWS scheme is the ratio of the masses of the weak bosons:

$$\frac{M_W}{M_Z} = \cos\theta_W \tag{1}$$

In the above equation θ_W is the Weinberg angle, where $\sin^2\theta_W$ has been experimentally measured to be 0.23101 ± 0.00027. This corresponds to a θ_W of $28.726 \pm 0.0014°$. During the 1990's, many experiments went into determining the mass of the Z^0, such as OPAL at CERN's LEP. It has been found to be 91.1875 ± 0.0021 GeV. But, if one uses (1) then the mass of the $W^{\pm}$ should be 79.964 ± 0.0022 GeV. This contradicts what has been found experimentally, and indicated that there is something amiss.

One reason for the difference is that these calculations are only to leading order in perturbation theory. There are many higher order processes, examples of which are illustrated in figure 2. These contribute to the measured mass of the $W^{\pm}$. Therefore, a precise measurement of the $W^{\pm}$ mass is crucial to our understanding of what may lie beyond the Standard Model. It would also be useful in attempting verify the Standard Model beyond leading order.

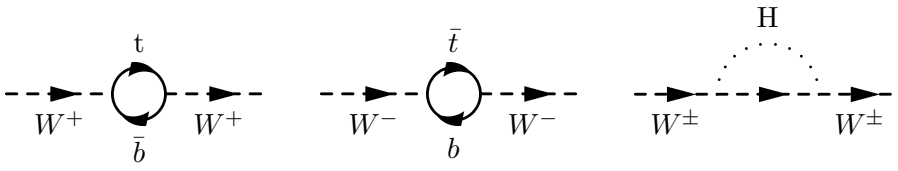


Figure 2: Feynman diagrams showing some loop corrections to leading $W^\pm$ propagation. The top-bottom quark loops dominate the predicted Higgs loop.

1.2 $W^\pm$ and Z^0 Production

1.2.1 Feynman viewpoint

One method of producing Z^0 and $W^\pm$ bosons is via $p\bar{p}$ collisions in a particle accelerator, such as the Collider Detector at Fermilab (CDF).

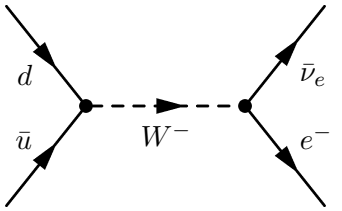
The Feynman representations of the leading order boson production, figure 3, show the basic weak force interactions vertices encountered. This is, however, a naïve view of the processes, a basic principle of quantum chromodynamics (QCD) is that all free particles must be colourless; free, or unbound quarks cannot exist

Figure 4 is a more realistic case: the quarks inside the proton and anti-proton collide with one another, but they may be gluon exchanges between the interacting quark and those left in the proton remnant to conserve colour. The quarks not involved in the $q\bar{q}Z^0$ vertex decay, those in the proton remnant, fragment into jets of hadronic particles. The decay of these extra quarks produces even more gluons. In addition to this process, there may be QCD radiation of gluons from the interacting quarks before they collide. This is far removed from the simple picture as depicted in figure 3(c). The added complexity as a result of the gluon interference brings added experimental problems with it when trying to determine which jet corresponds to which gluon decay.

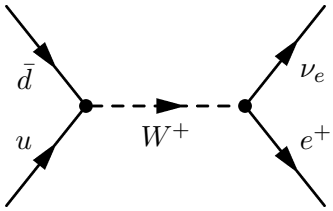
1.2.2 Schematic viewpoint

Schematically, the $p\bar{p}$ are fired at one another in a straight line, with the decay products being scattered off at an angle, the scattering angle, θ, to the incident beams. The quarks that fragment into the jets do not collide, thus the jets are projected along the path of the incident quark, rather than being deflected. The problem with the gluon interference is that it affects the momenta of the incoming quarks in the transverse plane. If one arbitrarily chooses the path of the incoming quarks to be the $\hat{z}$-axis, then the transverse plane exists in the $\hat{x}$ and $\hat{y}$ directions. In the naïve regime, all the incoming quarks only have all their momenta projected in the z-direction; this is not the case when gluon radiation is considered.

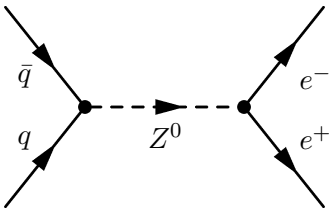
Highly energetic quarks are prone to QCD radiation of gluons, affecting the momenta of the quarks. Whereas before one could assume that the quarks had no momentum in the transverse plane, the emission of a gluon would cause the



(a) The formation of a W^- boson.



(b) The formation of a W^+ boson.



(c) The formation of a Z^0 boson ($q\bar{q}$ represents either a $u\bar{u}$ or $d\bar{d}$ pair).

Figure 3: The Feynman representation (to leading order) of the formation of the weak force bosons from a $p\bar{p}$ collision. It is assumed that the force mediators are travelling forward in time, and that time runs from left to right.

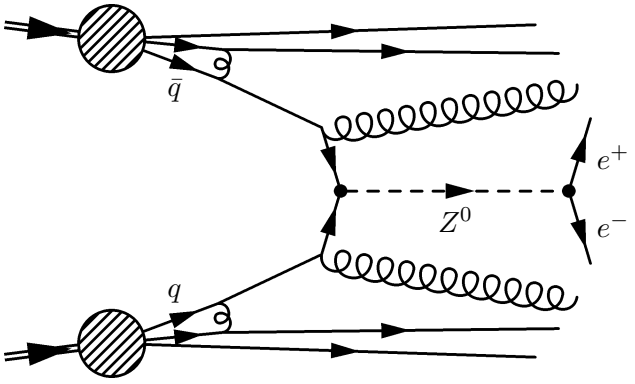


Figure 4: A more realistic process for the production of a Z^0 from $p\bar{p}$ collision. The quarks not involved in the collision fragment into hadronic jets (while a $u\bar{u}$ process is depicted, it could easily be a $d\bar{d}$ collision). QCD radiation of the colliding quarks is also shown.

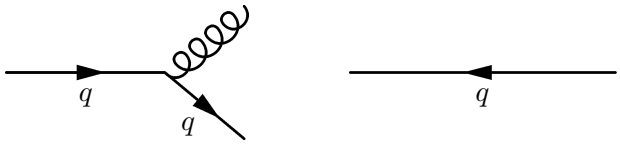


Figure 5: The effect of a quark emitting a gluon prior to collision. Whereas quark on the right is still travelling in the same path as before, the emission of a gluon from quark on the left has affected the momentum in the transverse plane.

quark's trajectory to change. These emitted gluons then decay into hadronic jets, in a direction opposing that of the formed boson.

1.3 Transverse momentum uncertainty

The uncertainty in the initial transverse momentum has a sizable effect upon the error estimates of $W^{\pm}$ calculations. One cannot directly observe the bosons, they are virtual particles; one can only observe the products from the decay of the boson. For the $W^{\pm}$ bosons, these are electrons, positrons, and their respective neutrinos and anti-neutrinos. However, neutrinos are deeply penetrating and cannot be easily detected. For this reason, in order to work out the fraction of momentum being carried by the neutrino, one has to assume the initial momentum of the $W^{\pm}$, the quantity one wishes to calculate. This is a problem, for M_W is derived from calculations involving the recorded mass in the transverse plane, the transverse mass.

No such problem exists for the decay of the Z^0, its decay products are electrons and positrons; the properties of the initial boson can be easily reconstructed from experimental signatures. The kinematics of $W \to e\nu$ are very similar to $Z \to e^- e^+$, therefore one may be able to estimate the transverse momentum distribution for $W^{\pm}$ from corresponding Z^0 distribution.

1.4 Comparison of the weak bosons

The Z^0 and $W^{\pm}$ are predicted, by the Standard model, to have similar decays. An inspection of the Feynman diagrams leads one to appreciate the similarities between the processes. Both decay into leptonic products, with amplitudes akin to one another arising from the vertices. The same can be said of the quark-boson vertices, both introduce similar contributions to the observed properties.

Both bosons are described by the massive propagator, albeit with slightly different masses involved. However the general form of the propagation term is the same in both cases.

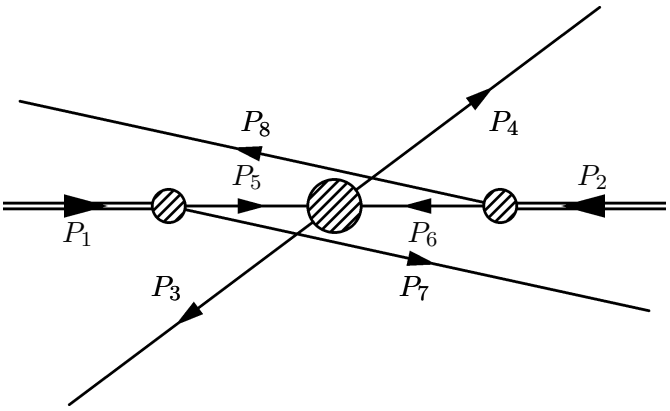


Figure 6: The kinematics of a two parton scattering process with two decay products. P_1 and P_2 are the incident $p\bar{p}$, P_3 and P_4 are the leptonic decay products. The interacting quarks are P_5 and P_6. The quarks from the proton remnant are P_7 and P_8. The QCD radiation gluons are not depicted, but they would affect P_5 and P_6.

2 Kinematics

2.1 Two-body processes

The properties of a particle are generally represented using 4-vector momentum, ie. (E, P_x, P_y, P_z). The benefits are apparent when attempting to reconstruct the properties of hidden particles from their decay products; adding all the 4-momenta of the decay particles together allows one to reproduce the 4-vector of the particle that produced them.

In the example above, figure 6, the two decay products, P_3 and P_4 were produced from the decay of the virtual particle involved in the process. Therefore, adding P_3 and P_4 together will produce a 4-vector momentum that describes the unseen particle. The $W^{\pm}$ decay produces neutrinos, so only P_3 or P_4 may be known, not both; both are observed in the decay of the Z^0. However, the properties of the hidden boson may be found by considering the particles that formed it, namely P_5 and P_6. Yet, these particles are never observed, but the other quarks from in the incoming protons may be detected. They are likely to disappear down the beam pipe of a detector, but they may miss, and strike part of the detector. Thus, if fortunate, P_7 and P_8 may be inferred from the properties of low angle jets, produced by the quarks fragmenting into hadrons. Using these, and knowledge of the incoming protons, P_1 and P_2, the properties of the quarks, and thus the properties of the boson, may be derived. This method is used to observe the $W^{\pm}$, but it is severely limited by how much data is lost by the remaining quarks not being measured by the detector.

2.2 Co-ordinate systems

The decay particles from a collision are better described using a spherical coordinate system, thus the following conversion into Cartesian is useful:

$$P_x = P \sin\theta \cos\phi \tag{2}$$
$$P_y = P \sin\theta \sin\phi \tag{3}$$
$$P_z = P \cos\theta \tag{4}$$

The total momentum may be calculated from the recorded energy and knowledge of the rest mass of the decay product, or, trivially, from the Cartesian components:

$$p = \sqrt{E^2 - m_o^2} = \sqrt{P_x^2 + P_y^2 + P_z^2} \tag{5}$$

2.3 Transverse plane

In collision physics, at the point of interaction, particles are directed toward one another in a straight line. The transverse plane can be defined as the plane perpendicular to the direction of the incoming particles. In general, the direction of the colliding beams is designated as $\hat{z}$, thus, the transverse plane is that in the $\hat{x}$ and $\hat{y}$ dimensions. The hadrons that result from the decay of the proton remnant are usually lost down the beam pipe. It is for this reasons that measurements are made in this plane.

Using Cartesian coordinates, and taking the path of the incoming particles to be $\hat{z}$, then the transverse momentum, energy, and mass can be defined:

$$E_T = \sqrt{E_x^2 + E_y^2} \tag{6}$$
$$P_T = \sqrt{P_x^2 + P_y^2} \tag{7}$$
$$M_T = \sqrt{E_T^2 - P_T^2} \tag{8}$$

2.4 Rapidity and pseudorapidity

The rapidity, y, is an important quantity in the kinematics of particle collisions. It is defined as:

$$\tanh y = \frac{P_z}{E} \tag{9}$$

The rapidity is a useful because Δy is Lorentz invariant, and may be used to convert between frames of reference. If one takes the limit where the energy of the particle is much larger than the mass of the particle, then the pseudorapidity,

η, may be used as an approximation (where θ is the scattering angle between the incident beam and the scattered products):

$$\eta = -\ln \tan \frac{\theta}{2} \tag{10}$$

In general, the pseudorapidity is a more useful measure of angle than the scattering angle. Given that particle detectors are generally cylindrical about $\hat{z}$, then any point on this cylinder may be described by (η, ϕ).

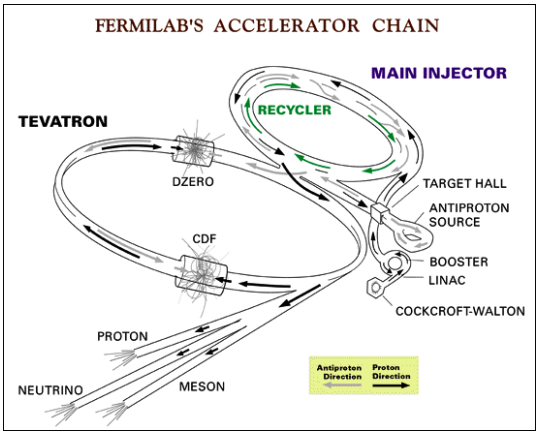


Figure 7: Outline of the cyclotrons at Fermilab. (Image credit: Fermi National Accelerator Laboratory)

3 The CDF detector

3.1 Overview

The Collider Detector (CDF) is one of the detectors at the Fermi National Accelerator Laboratory (Fermilab), a high-energy particle physics laboratory in Batavia, Illinois (45 miles from Chicago). The main accelerator at Fermilab is the Tevatron, which could accelerate protons and anti-protons up to energies of 1.8 TeV during its first run, the CDF Run I experiment, from which the data in this project was obtained.

Like all cyclotrons, the Tevatron accelerates protons and anti-protons around in loops, gaining energy, before directing the particle beams toward one another inside a detector, figure 7.

Similar to most detectors in high energy particle physics, the CDF is not one detector, but a combination of several different detection systems. A majority of the detection systems are orientated with a cylindrical geometry around the beam pipe.

The innermost systems are tracking systems, used to determine the trajectories of charged particles. The first of these systems in the CDF is a silicon vertex detector (SVX). This type of tracking detectors has a very high spatial resolution, and is used to make precise measurements of very short-lived particles close to the beam pipe. The SVX has a spatial resolution of 30–50μm. Silicon vertex detectors are expensive, and it is not feasible to have an entire tracking system using the material. Therefore, there is a cheaper drift chamber, the Central Outer Tracker (COT), surrounding the SVX. The COT has a spatial resolution of 150–200μm. Again, like the SVX, drift chambers can only be used to observe the paths of charged particles.

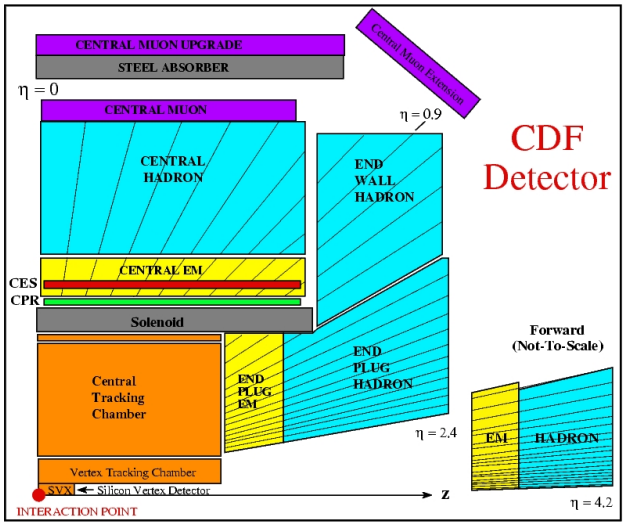


Figure 8: The different detection systems in the CDF. The systems of note are the central and plug electromagnetic calorimeters, which have detection ranges of $|\eta| \leqslant 1$ and $1 < |\eta| \leqslant 2.4$ respectively (Image credit: Fermi National Accelerator Laboratory)

The electrons and positrons given off from the decay of $W^{\pm}$ and Z^0 can be observed by these systems, but the neutral neutrinos and anti-neutrinos cannot, one reason why neutrinos are difficult to observe in experiments. Neutrinos can only interact with matter via the weak force, therefore they only leave a tiny energy loss as they leave the detector. It is estimate that one interaction in every 10^{10}–10^{21} is a neutrino.

Measurement of the trajectories is very important in particle identification when the data is compared with the energy signatures observed by the calorimeters.

In between the tracking chambers and the calorimeters is a solenoid. This subjects the entire CDF apparatus to a magnetic field, allowing one to derive the particle momentum from the curvature of the paths of charged particles.

The tracking systems are surrounded by electromagnetic and hadronic calorimeters, used to measure the energy and position of particles. The calorimeter data is used with the tracking data in aid in particle identification. The curvature of the charged particles in the SVX and COT is used to determine where the particle should strike the calorimeters. The two calorimeters in the central region are the Central Electromagnetic (CEM) and the Central Hadron (CHA) calorimeters.

The detection systems beyond the calorimeters are used to detect muons. Muons are deeply penetrating and they would not be stopped by the previous apparatus. The layers of metal in the calorimeters sometimes reduces their energy to levels that allow detection, but this has no effect upon the neutrinos which will

escape the detector unobserved.

The systems already discussion were all in the main detection area, where $|\eta| \leqslant 1$; there are extra detectors at greater angles, increasing the angular resolution of the CDF. In the $1 < |\eta| \leqslant 2.4$ region there are extra tracking systems. There is smaller tracking chamber, the ISL, and behind that is the Plug. The Plug contains two calorimeters, one electromagnetic (PEM) and the other is hadronic (PHA).

3.2 Calorimeters

The two calorimeters of importance in the scope of the investigation are the CEM and PEM. The CEM is not one, but several hundred modular calorimeters, arranged in a cylindrical configuration around the beam pipe, figure 9.

Calorimeters usually consist of alternating layers of an absorbing material and a detector. When an electron enters the calorimeter, it passes heavy nuclei, radiating photons as it scattered off them. As a result, the energy of the incident electron is shared out. The electron and photon generate many secondary particles, forming an electromagnetic shower. The electron and positron shower because they have a small mass; heavier muons do not reach relativistic velocities, and will not radiate photons as they pass through the absorbing material.

Various processes can generate secondary particles, electrons and positrons can scatter off the lead nuclei, emitting photons in the process. The photons may produce electron-positron pairs, which can then radiate yet again. While all these processes produced fluctuations in the signal, they are random, which is important when the statistics of the electromagnetic shower are considered.

The CEM uses lead as the absorbing material and a scintillator as the active detection region. The PEM uses a drift chamber in the detection region and lead as the absorber.

3.2.1 Calorimeter resolution

Measurements during the testing of the CDF gave energy resolutions of 2% for the CEM and 4% for the PEM [1]. These resolutions refer to the limit of each module in the calorimeter, but each module will have its own characteristic error, brought about by differences such as the electronics. Thus, there is an added uncertainty on top of the error in each module, it is denoted as κ.

A useful feature of κ is that it can account for all uncertainties that affect the CEM and PEM, even if an source of error has not been identified, some other sources of error are depicted in figure 10.

The energy resolutions:

$$\left(\frac{\Delta E}{E}\right)_{CEM} = \sqrt{\frac{(0.135)^2}{E_T} + \kappa_{CEM}^2} \tag{11}$$

$$\left(\frac{\Delta E}{E}\right)_{PEM} = \sqrt{\frac{(0.16)^2}{E_T} + \kappa_{PEM}^2} \tag{12}$$

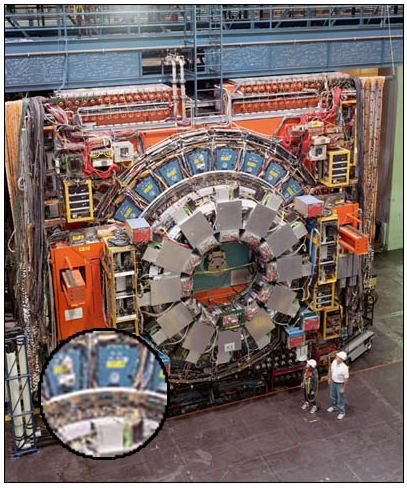


Figure 9: End view of the CDF showing several of the detection systems. Inset: Close up view of the modular system used for the calorimetry. (Image credit: Fermi National Accelerator Laboratory)

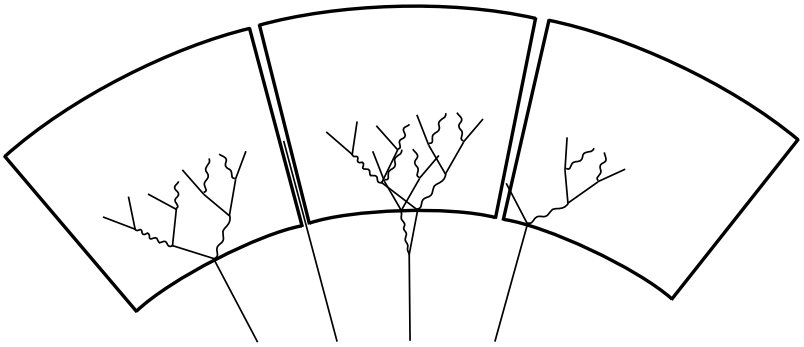


Figure 10: A diagram of various factors that can affect the accuracy of measurements. From left to right: an electron that enters the calorimeter in a usual fashion; an electron that has missed the CEM modules and will not be detected; an electron that has emitted a photon prior to entering the detector; an electron that enters a CEM module near the edge, leading to the possibility of some of the electromagnetic shower leaking out of the side.

4 Methodology

All of the data analysis in the project was carried out on a Linux platform with programs written in either C++ or FORTRAN. The Root data analysis package was used extensively throughout the investigation, with the custom C++ classes and histogram generation extremely useful.

4.1 Initial HERWIG analysis

Various properties of the decay of the Z^0 and $W^{\pm}$ were investigated at the start of the project. The initial data was generated using HERWIG (Hadron Emission Reactions With Interfering Gluons), a Monte Carlo simulation that models the collision of particles.

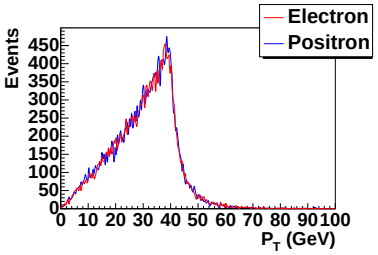
An analysis program was written that extracted the 4-momenta of all the final particles in each event generated by HERWIG. This program could be used to generate histograms using Root, an object-orientated data analysis package, which has been developed at CERN. The $W^{\pm}$ decay products were investigated first.

Each histogram is declared at the beginning of the program, to be filled with data later on. For each event in the input file, the program would loop over all the particles HERWIG had generated. Each particle would be identified, and only electrons and positrons would be considered. At this point the transverse momenta, pseudorapidity and energy of each would be calculated.

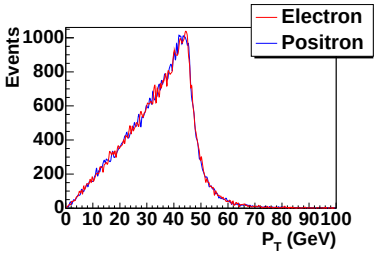
There is no reason to suppose that this particle would have been produced by the decay of the $W^{\pm}$, so only the particle with the highest transverse momentum per event was retained. This was achieved by simply storing the values of decay particle outside the loop, with the current particle transverse momentum being checked against this transverse momentum. If the current particle had a higher P_T, then the values were replaced. The properties of the particle were then binned into a histogram. The output of the program was a Root file containing histograms, which could then be formatted inside Root itself. The same was conducted for the Z^0.

Figure 11 shows a comparison of the electron and positrons that were produced from the decay of the $W^{\pm}$ and Z^0 bosons. An important aspect of the project is that the $W^{\pm}$ can be modelled from Z^0 data. As can be seen, the electronic decay products are very similar for both bosons. The P_T distributions, figures 11(a) and 11(b), are as would be expected. One would imagine that the electron and positron that result from the decay of the $W^{\pm}$ would contain around half the energy of the original boson. The HERWIG data indicates that there is a cut-off point in the transverse momenta at 40 GeV. One would assume that the remainder of the momentum is contained in the neutrino. The same is the case for the Z^0 with a cut-off being at 45 GeV.

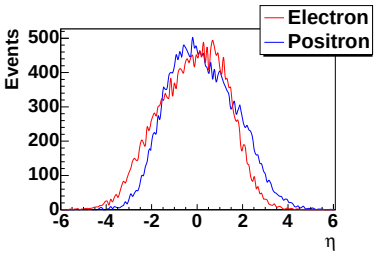
The pseudorapidity distributions, figures 11(c) and 11(d), are not the same for both bosons. The W^+ and W^- have asymmetric pseudorapidity distributions, there is a preferential direction of decay. Therefore, the electron distribution would



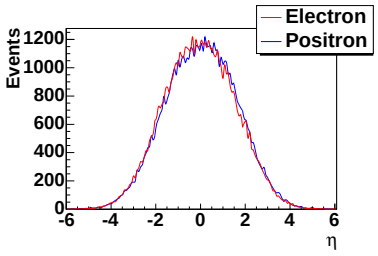
(a) P_T distribution of the products of $W^{\pm}$ decay.



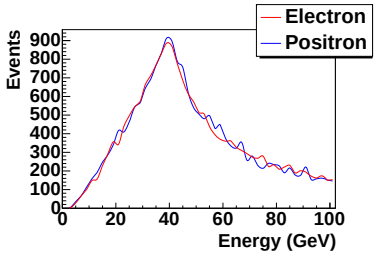
(b) P_T distribution of the products of Z^0 decay.



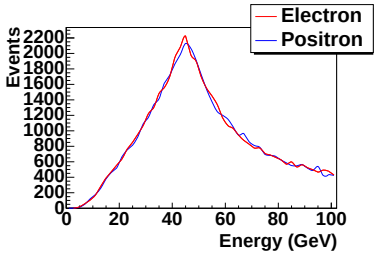
(c) Pseudorapidity distribution of the products of $W^{\pm}$ decay.



(d) Pseudorapidity distribution of the products of Z^0 decay.

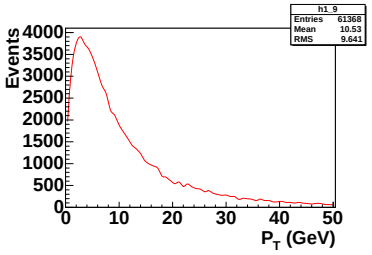


(e) Energy distribution of the products of $W^{\pm}$ decay.

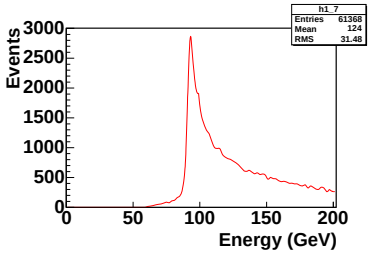


(f) Energy distribution of the products of Z^0 decay.

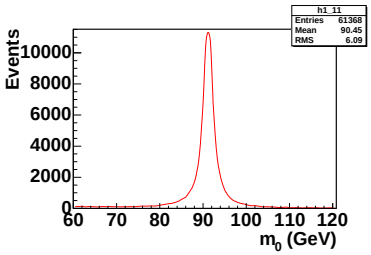
Figure 11: Data generated from a HERWIG simulation of a $p\bar{p}$ collision and $W^{\pm}$ and Z^0 decay.



(a) P_T distribution.



(b) Energy distribution.



(c) Invariant mass distribution

Figure 12: Various distributions of the decay of a Z^0 generated from a HERWIG simulation of a $p\bar{p}$ collision

reflect that of the W^-, and the same for the positron and W^+. Both the electrons and positrons are emitted, with no bias, in the decay of the Z^0, and the pseudorapidity distributions reflect this. The energy distributions, figures 11(e) and 11(f), both show a peak energy at half that of the mass of the original boson.

Observing the data, it can been seen that the $W^\pm$ and Z^0 decay to leptons are very similar, and that the Z^0 decay could be used to model the $W^\pm$ decay, albeit, after the Z^0 distributions are corrected for the subtle differences between the two.

The actual Z^0 was also investigated in a similar fashion. To determine the properties of the Z^0, the 4-momenta of the electron and positron were combined once they had been identified as the main particles in each event. The reconstructed Z^0 4-momentum was used to generate transverse momentum, energy, and invariant mass distributions.

The energy, figure 12(b), and the rest mass, figure 12(c), distributions are both peaked around 90 GeV, which is expected; mass of the Z^0 is 91.1875 GeV, so one would hope that the rest mass distribution generated by HERWIG would peak at this value. It should be noted that it is possible for the mass to be

measured with much lower values as a result of electroweak mixing. The $p\bar{p} \rightarrow e^- e^+$ process has two routes: the weak neutral current, ie. the Z^0 boson; the other is via the electromagnetic force. Hence, the photonic contributions to the measure mass become more significant the lower one gets in the transverse mass spectrum. However, in the 60–120 GeV range, the one investigated, the dominant contribution to the measured boson mass would be that of the Z^0.

The energy distribution exhibits a clear threshold at which boson production is allowed. It implies that it is highly unlikely for the boson to be measured with an energy that is lower than its rest mass, which is expected.

The transverse momentum distribution, figure 12(a), is peaked at 3–4 GeV with a slowly decaying tail. This value is not too large, but it is significant. Any estimates of the transverse mass of the boson would have an uncertainty of this magnitude. If it is assumed that there is zero transverse momentum, then the measured energy would correspond to the mass, for example, measuring the energy to be 91.187 GeV would correspond to a mass of 91.187 GeV. Yet, should the P_T of the boson be 4 GeV, rather than the assumed value of zero, then the mass measurement would be affected by 90 MeV. Work is being carried out trying to reduced the error on the $W^{\pm}$ mass to less than 30 MeV; being able to model the P_T would be very useful in this matter.

However, it should be noted that one major benefit of the determining the mass of particles via the transverse mass, is that the Jacobian peak is not greatly affected by smearing. Yet, with such stringent requirements on the knowledge of the $W^{\pm}$ mass, all work that can reduce the uncertainty in mass measurements is helpful.

This part of the project was not without problems. The first HERWIG simulation for the Z^0 was incorrect, and the project was delayed while a correct simulation was generated. In addition, there was a period of familiarisation with the programs such as Root, and using Linux as an development environment.

4.2 Detector effects

The next stage involved building the detector simulation. Detector simulations are crucial when testing theories, data detailing a particle collision will be tempered by the resolution of the apparatus used to record it. In order for theoretical simulations to be compared with real recorded data, the theoretical data has to have to effectively 'go through' a detector.

The detector simulation in this project will model the effect of the CEM and PEM. The energy resolution would be built into the data analysis program used to investigate the HERWIG model of the boson decay.

The processes that occur inside the calorimeters bring about fluctuations in the the signal recorded, however, these fluctuations are completely random, hence, Gaussian statistics can be employed to model them.

Using the CEM resolution, (11), a formula for adding this uncertainty into the detector measurements can be derived. If one assumes that the ratio between the

transverse energy, E_T, and the total energy is the same as the transverse momentum ratio, ie. $\frac{E_T}{E} = \frac{P_T}{P}$, then the CEM resolution may be written as an error upon the energy:

$$\Delta E = E\sqrt{\frac{(0.135)^2 P}{P_T E} + \kappa_{CEM}^2} \tag{13}$$

The error was applied using Gaussian statistics. Defining σ to represent a normally distributed random number, where the mean is zero and the standard deviation is unity, then the energy of a particle, as seen by the CEM is:

$$E_{CEM}(\kappa) = E\left(1 + \sigma\sqrt{\frac{0.135^2 P}{E P_T} + \kappa_{CEM}^2}\right) \tag{14}$$

The error on the momentum was determined using (5) and standard error propagation:

$$\begin{aligned}
(\Delta p)^2 &= \left(\frac{dp}{dE}\right)^2 (\Delta E)^2 && (15)\\
&= \left(\frac{E}{\sqrt{E^2 - m_0^2}}\right)^2 (\Delta E)^2 && (16)\\
\therefore \Delta p &= \frac{E\Delta E}{\sqrt{E^2 - m_0^2}} && (17)
\end{aligned}$$

The position of the particles in the detector is measured by the likes of the SVX and other tracking detectors. These have a very good resolution (the SVX has a spatial resolution of 30–50μm [1]), therefore, the angular uncertainty was neglected in the simulation.

Thus, using equations (2), (3), and (4):

$$\frac{dp_x}{dp} = \frac{dp_y}{dp} = \frac{dp_z}{dp} = 1 \tag{18}$$

$$\therefore \Delta p = \Delta p_x = \Delta p_y = \Delta p_z = \frac{E\Delta E}{\sqrt{E^2 - m_0^2}} \tag{19}$$

The error in the momentum was applied in the same fashion as the energy error, thus:

$$P_{i,CEM}(\kappa) = P_i\left(1 + \sigma\frac{E\Delta E}{\sqrt{E^2 - m_0^2}}\right) \tag{20}$$

The data analysis code was modified to simulate the resolution of the CEM. Once again the code would loop over all the events in a HERWIG simulation.

If the particle was an electron or a positron, then code would create a clone of the 4-vector momentum. The cloned 4-vector would have its components replaced with 'smeared' components, using the CEM resolution error equations (14) and (20). The smeared and unsmeared particles were recombined to form a smeared and unsmeared Z^0. The required properties were binned into histograms. The initial CEM smearing was conducted using a κ of 2.5% and no pseudorapidity cuts were made at this stage; the purpose was to observed the effect of the CEM resolution upon the simulated data.

The effect of the CEM resolution upon the energy distribution, figure 13, of the Z^0 is as expected. The sharp edge to the distribution is being smeared out by the uncertainty in the measurements. This allows more measurements where there should not have been enough energy to produce the boson, the uncertainty lessens the rigidity of the threshold.

The transverse momentum of the Z^0, figure 14, shows the peak shifting toward a higher value.

With uncertainty in the the momentum and energy measurements means that there will be a greater error in the mass measurements. This is reflected in the data, figure 15. The smeared distribution has a much larger spread of mass values because of the greater error in the measurements.

The effect of κ upon these distributions was then looked into; it is an unknown parameter, but if its effect was negligible, then it could have been safely neglected. Therefore, an investigation how it affects the data, and its significance was necessary.

The PEM calorimeter can also be modelled. Using (12) and a similar argument as before, the energy smearing for the Plug can be determined:

$$\Delta E = E\sqrt{\frac{(0.16)^2 P}{P_T E} + \kappa_{PEM}^2} \tag{21}$$

Thus,

$$E_{PEM}(\kappa) = E\left(1 + \sigma\sqrt{\frac{(0.16)^2 P}{E P_T} + \kappa_{PEM}^2}\right) \tag{22}$$

The momentum smearing is the same as before:

$$P_{i,PEM}(\kappa) = P_i\left(1 + \sigma\frac{E\Delta E}{\sqrt{E^2 - m_0^2}}\right) \tag{23}$$

The effect of κ_{CEM} and κ_{PEM} was investigated by holding either one of the κ at an expected value, and varying the other over an expected range. The same data analysis program was to be used, but with different code to run every event. Rather than create one smeared clone 4-vector for each particle, a smeared 4-vector was created for every value of κ being tested. Thus, for n values of κ there were n

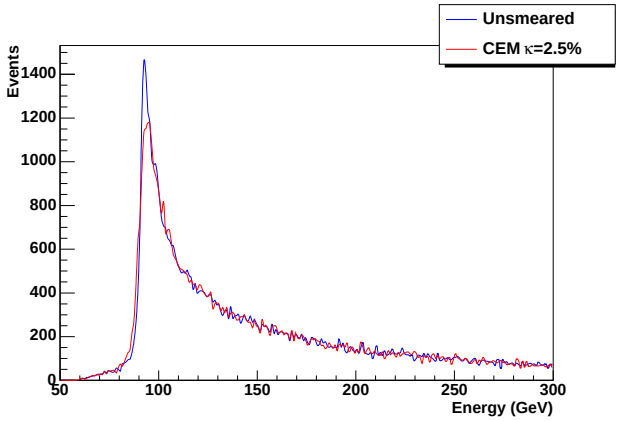


Figure 13: The CEM smearing effect upon the energy of the Z^0. No cuts were made.

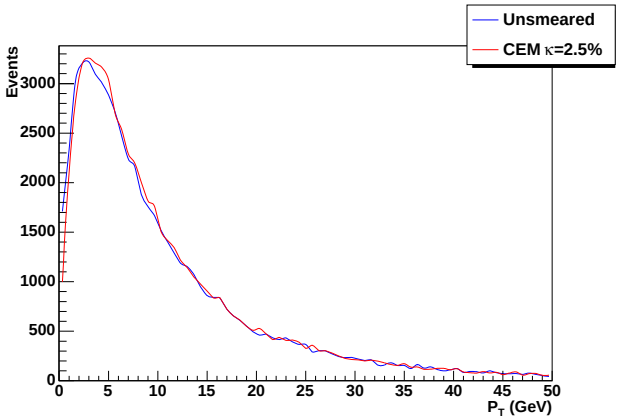


Figure 14: The CEM smearing effect upon the transverse momentum of the Z^0. No cuts were made.

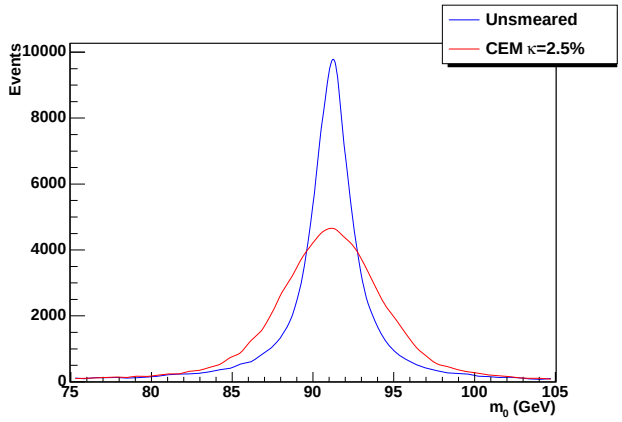


Figure 15: The CEM smearing effect upon the rest mass of the Z^0. No cuts were made.

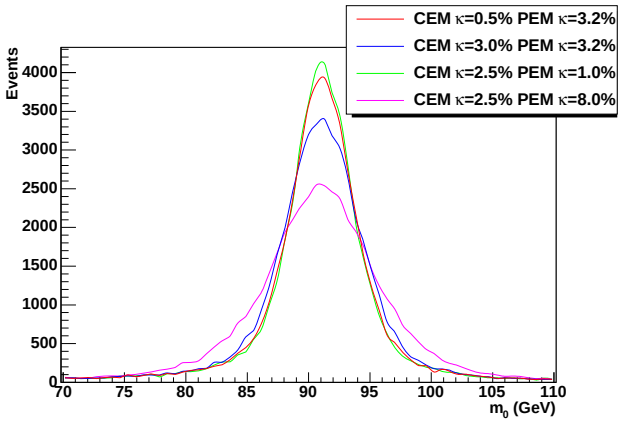


Figure 16: The invariant mass of the Z^0 boson in HERWIG simulated $p\bar{p}$ collisions with various CEM and PEM smearing effects applied. All particles with $|\eta| > 2.4$ were rejected.

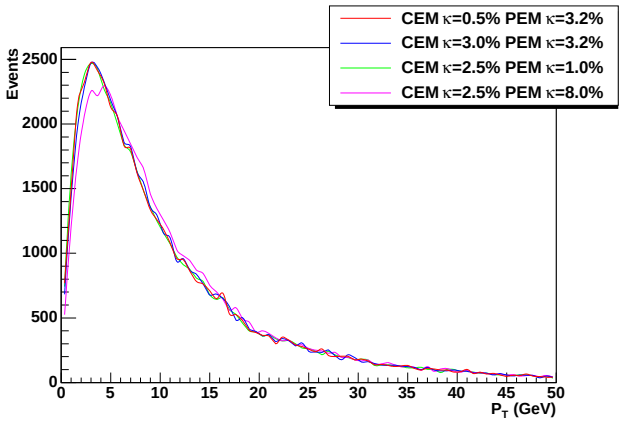


Figure 17: The transverse momentum of the Z^0 boson in HERWIG simulated $p\bar{p}$ collisions with various CEM and PEM smearing effects applied. All particles with $|\eta| > 2.4$ were rejected.

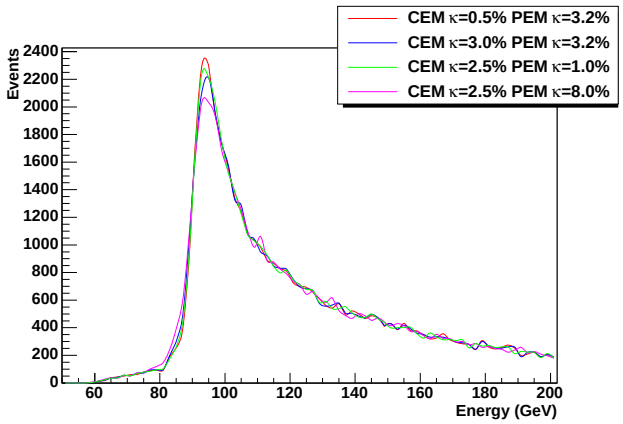


Figure 18: The energy distribution of the Z^0 boson in HERWIG simulated $p\bar{p}$ collisions with various CEM and PEM smearing effects applied. All particles with $|\eta| > 2.4$ were rejected.

smeared Z^0 4-vectors. However, the CEM and PEM effects are only valid within certain pseudorapidity ranges; they are physical objects, and can only measure in the range in which they exist.

If the electron or positron fell in a $|\eta| \leqslant 1$ range, then CEM formulae were used; if it were to land in the $1 < |\eta| \leqslant 2.4$ range, then the PEM formulae were applied instead. All particles with $|\eta| > 2.4$ were rejected by the detector simulation. Once the detector smearing has been applied, and the Z^0 bosons reconstructed, the histograms were binned.

The CEM effect was applied with κ_{CEM} varied between 0.5%–3.0% and the PEM effect was applied with κ_{PEM} in the range 1%–8%. While κ_{CEM} was being varied, κ_{PEM} was held constant at 3.2%; κ_{CEM} was kept constant at 2.5% when the κ_{PEM} was being varied.

The rest mass distribution of the Z^0 boson, figure 15, was affected the most by introducing the detector effect, hence, observing changes in this property would best indicate the significance of κ_{CEM} and κ_{PEM}. Figure 16 show that the range that κ_{PEM} can take affects the data more than the range of values that κ_{CEM} can take. This is not surprising, the constraint upon the κ_{CEM} is much stricter than that upon κ_{PEM}. In both cases, increasing the value of κ increases the uncertainty in the invariant mass of the boson, which is expected. Rough estimates on the error in the CEM and PEM are 2% and 4%; the size of extra error, κ, in both cases can exceed the magnitude of the original error.

This is not as apparent when looking at the transverse momentum distributions, figure 17. This serves to highlight how sensitive the detectors are to changes in the transverse momentum. The CEM curves are very close to one another, albeit with the $\kappa_{CEM} = 3.0\%$ curve slightly shifted to higher values. The PEM effect over the range of acceptable values is far more noticeable.

The energy distributions, figure 18, exhibit smearing of the peak, with larger values of κ increasing the amount of smearing observed.

Neither κ_{CEM} or κ_{PEM} were known and needed to be determined for the data that was being used. While the lack of knowledge about the PEM errors has more of an effect upon the distributions, the CEM is the more important detector, given its location in the central detection region. Yet, all the differences between the distributions are meaningless without a comparison of real data from the CDF, complete with statistical error. The CEM distributions in figure 16 were compared with CDF rest mass data, and χ^2 fits of the predictions to the real data were made.

The CDF data was read in from a file, and the binned into a histogram. The CEM smeared HERWIG distributions were normalised, then compared bin by bin, producing figure 19.

The χ^2 values for the number of bins do not indicate that the fits are particularly good, but the figures serve to indicate the difference in χ^2 over the range of the acceptable κ_{CEM}, hence it needed to be determined.

This section of the project was completed without many problems. Early on there were some problems with the application of the smearing effects. Other concerns were generally with the use of C++ and recognising what the compilation

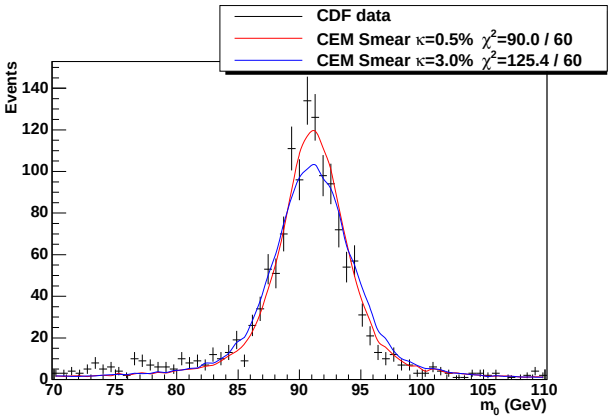


Figure 19: The invariant mass of the Z^0 boson in HERWIG simulated $p\bar{p}$ collisions with various a constant PEM smearing effect applied ($\kappa = 3.2\%$). The CEM smearing has been applied with κ of 0.5% and 3.0%. All particles with $|\eta| > 2.4$ were rejected and the χ^2 fit information has been shown for both predicted distributions when compared with actual mass information from the CDF.

errors were. As means of checking if the energy smearing had been applied correctly, the energy error added was plotted in histograms with each run of the program. If the smearing had been applied correctly, then the resulting histogram should have been a Gaussian distribution.

4.3 Finding κ_{CEM}

The previous part of the project had shown that in order to model the effects of the CEM, accurate knowledge of κ_{CEM}[1] was required. The rest mass distributions are very sensitive to changes in κ, and the rest mass of the Z^0 is very well known. The fore is was very reasonable to use the distributions to find κ, determining which $M_Z\kappa$-distribution best fit the CDF mass data.

The fit was made using a `TMinuit`[2] Setting up the fit involved writing a lengthy program that could take an array of $M_Z\kappa$ distributions and generate a new normalised distribution for any value of M_Z or κ. A χ^2 or log likelihood comparison of the distribution would be the function that `TMinuit` would minimise.

The main program would first read the CDF data from an ASCII file and store this in an array. The program was intended to be object-orientated, thus a class was written to store this data. The program would then open the Root file

[1] Now to be referred to as κ for brevity.

[2] More on `TMinuit` can be found in Appendix A.2.1.

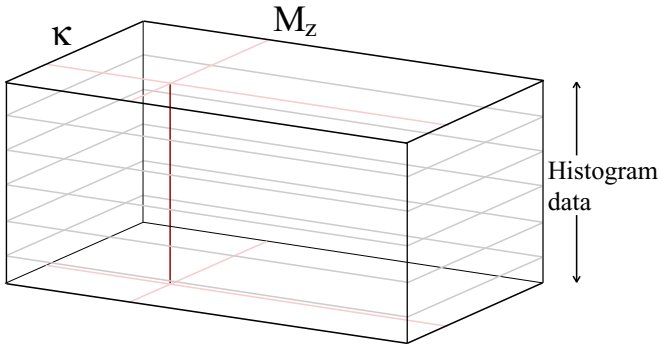


Figure 20: A graphical representation of the 3D array of data. Each plane depicted represents all the possible values that one bin in mass distribution may take. Interpolating down the line would generate the mass distribution for those given values of M_Z and κ.

that contained histograms. These histograms were rest mass distributions with M_Z ranging from 90–92 GeV in 0.125 GeV steps and κ ranging from 0.3–2.3 with 0.2 as the step size. A class was written to extract the information about the distributions, and to allow the extraction of the correct histogram for a particular M_Z and κ combination.

This index class was used to create a 3D array of data containing the information in each histogram in the input Root file. Explicitly, the array contained the bin contents of the histogram, with the particular point in the histogram the third dimension extending from the two dimensions of M_Z and κ, figure 20.

This array was interpolated to create new mass distributions. If the input Root file contained histograms with n bins, then there would be a 3D array with n $M_Z\kappa$-planes. A mass distribution for an arbitrary $M_{Z,i}$ and κ_i could then be created by interpolating the each of n planes at $(M_{Z,i},\kappa_i)$, forming a new array of data with n entries. This was then used to create a Root histogram. Bicubic interpolation [2] was used in preference to polynomial interpolation because of the lack of stability in the interpolating polynomials at higher orders. The use of polynomials is not recommended when interpolating data that contains more than 6–10 data points; there were 17 points in the M_Z dimension and 10 points in the κ dimension. A class was written to store the generated mass distributions, containing methods to allow the conversion into Root histograms.

The generated histograms also needed to be normalised, so that the integral was equal to unity. The preferable method was to use a Gaussian Quadrature algorithm, but the program would crash due to a memory error every time it was implemented. This did not provide to be too much of a problem, a summation over all the bin areas agreed with the former method to four significant figures.

The χ^2 fits were made using a bin by bin comparison in two regions: over the whole range (70–110 GeV) and in the peak (80–100 GeV). Fitting over the whole

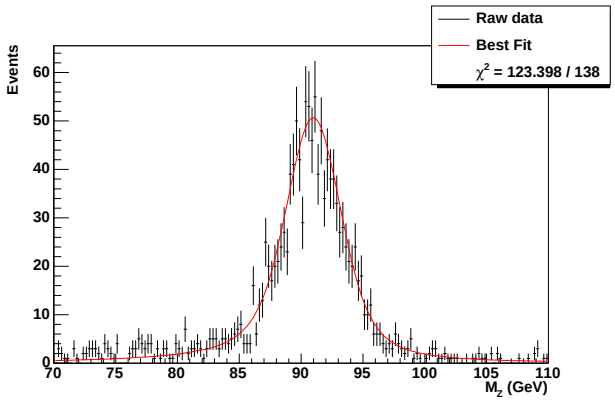


Figure 21: The best fit mass profile to the CDF data.

distribution determines M_Z to be 91.1241 ± 0.0130 and κ as 1.5065 ± 0.2368. The χ^2 for the fit was 123.383 with 138 degrees of freedom. Fitting in the range of the peak gives M_Z to be 91.1229 ± 0.0136 with κ as 1.5105 ± 0.2299. The χ^2 for this fit was 76.1488 with 79 degrees of freedom. Hypothesis testing gave 80% and 54% as the maximum significance level the fits would pass.

There were some problems with this section of the project. Firstly, the values of κ for the data were not as close to a previously determined estimate of 1.742% as hoped. However, the figure that was calculated, 1.5105 ± 0.2299, has this lying on at the very edge of the error limits. Unfortunately, the measured mass of the Z^0 also falls outside the range of the mass predicted from the fitting program, but only by 80 MeV. While the derived κ value is close to the estimate, the estimate was used in the later fitting.

Other problems were centred around the actual programming; having little experience of C++ before undertaking the task made progress slow, for until that point in the project, all analysis was conducted inside a framework that has already been written and tested. Isolating and removing bugs from the code was very time-consuming, and days could be lost attempting to correct bugs.

Furthermore, when the program was initially completed, it would not return a reasonable fit, nor an acceptable answer. However, the added experience in using `TMinuit` allowed the program to be corrected at a later date. The decision had already been made to continue the project with a proper κ value before this time, however.

At first, the interpolation routines were suspected: at one point the routines were creating mass distributions with negative regions. This proved not to be the cause of the problem when the interpolated mass histograms were compared

with an external check. The check was conducing using Root, and two adjacent distributions. The distributions were added to one another with a weight of 50% each, producing a linear interpolation between the added mass distributions. These linearly interpolated histograms were exactly the same as those generated by the fitting program.

The problem was found to have been in the actual χ^2 test. Later experience showed that the best method involved creating Root histograms with identical bins, and then conducting the χ^2 on a bin by bin basis.

Some of the problems with the program would have been avoided had it been created with proper exception handling from the start; debugging would have been quicker, and modification of the program would have been simpler. This would have been done had the program been written in a more familiar language, Java, for example, but it was not possible; the only language that would work with Root was C++.

4.4 Transverse momentum of Z^0

Having ascertained the κ_{CEM} for the data, an theoretical model of the transverse momentum of the Z^0 could be run through a detector simulation and then compared with CDF data.

A functional form [3] was chosen to represent the form of the Z^0 transverse momentum:

$$\frac{\mathrm{d}\sigma}{\mathrm{d}p_T} = \frac{\left(\frac{P_T}{50}\right)^{P_4}}{\Gamma\left(P_4+1\right)}\left[\left(1-P_1\right)P_2^{P_4+1}e^{\frac{-P_2P_T}{50}} + P_1P_3^{P_4+1}e^{\frac{-P_3P_T}{50}}\right] \tag{24}$$

The form contains no physical basis whatsoever, it is no more then an *ad hoc* function, containing a superposition of two exponentials, that may be used to represent the shape of the transverse momentum distribution. This form was to have the detector resolution applied and then fitted to the CDF data. The smeared functional form is a convolution of the unsmeared form and the smearing function:

$$P_T^{smear} = P_T^{true} \otimes f_{smear} \tag{25}$$

Attempting the convolution of (24) with the smearing function analytically proved to be too time consuming to compute, it took about 5 minutes to process the Fourier transform of (24) with '*Mathematica*', and the result was consisting of nested Fourier transforms. This would slow down the fit far too much so a less elegant, but far more effective method was chosen.

HERWIG events were used to generate two $P_T(Z)$ distributions, one that described the original Z^0, and one that described the Z^0 as seen by the CEM. Normalising these distributions, and dividing one by the other would give a weighting array. A histogram was created from the functional form and each bin was reweighted according to the weighting array. Then functional form could then be fitted to CDF data.

A program was written, which read in HERWIG generated 4-vector momenta for the electron and positron produced from the Z^0 decay. These would be converted into a smeared and unsmeared Z^0 using the detector simulation developed in §4.2. It then created the weighting from these distributions. The program then read in $P_T(Z)$ data from the CDF and binned it into a distribution. `TMinuit` was used to find the four parameters that would give the best fit functional form. The function it would minimise would generate a histogram using an arbitrary selection of parameters. Each bin of the histogram was that the form described the smeared $P_T(Z)$. The χ^2 fit of this histogram to the data was the figure used to minimise. During the program's testing phase, approximately 60000 HERWIG events were used to create the weighting array, but 4.6 million events were required to generate smooth histograms during program execution.

The parameters that generated the best fit, figure 22, are discussed in §5.

There were not many problems with this part of the project. Having gained significant experience in C++, programming the fit did not take long to set up. It was not completely without problems, some time was spent trying to get `TMinuit` to generate a fit that has reasonable errors. The first fits were also incorrect, and by pure chance, a bug in the detector simulation partially corrected for the mistake. Initially, the value of κ entered into the program was 1.742 but this corresponded to an extra 170% error in the measured data as a result of the CEM. This coupled with the incorrect implementation of the smearing formulae actually managed to produce fits that looked reasonable, albeit with extremely anomalous errors. Correcting both the κ value used, and the detector simulation resulted in more appropriate errors.

4.5 $W^{\pm}$ mass uncertainty

A method of producing new $P_T(Z)$ distributions relatively quickly[3] had now been developed. This was to be converted into an estimate on the mass of the $W^{\pm}$.

As has already been mentioned in §1.4, while the Z^0 and $W^{\pm}$ are similar enough to model $P_T(W)$ from $P_T(Z)$, the differences between them affect the results beyond the sensitivity of the data. Therefore, the $P_T(Z)$ needs to be re-weighted to account for the differences in the theory. The $P_T(W)$ could then be used to estimate the transverse mass of the $W^{\pm}$. There was not enough time in the project to complete this final section, however the method to produce an mass uncertainty from the $P_T(Z)$ will be discussed.

`TMinuit` produces an error matrix, this can be diagonalised with the errors limited to within one standard deviation, giving a new set of parameters to produce a new $P_T(Z)$. The program would read in a file containing the results of the re-weighting calculations[4]: transverse mass results and the $P_T(W)$ used to generate

[3]The program took 3–4 minutes to read and process the 4.6 million HERWIG events, but this is faster than generating n sets of HERWIG events.

[4]The file contained a next to leading order calculation of $\frac{\mathrm{d}^2\sigma}{\mathrm{d}y\mathrm{d}P_T}|_W / \frac{\mathrm{d}^2\sigma}{\mathrm{d}y\mathrm{d}P_T}|_Z$ averaged over all rapidity.

them. Firstly, the program would create a transverse mass plot from the data in the file, it would also create a theoretical $P_T(W)$ distribution.

The program would then create a new set of parameters by diagonalising the covariant matrix from `TMinuit`. A new $P_T(Z)$ would be created from these using (24). Normalising and dividing these P_T distributions would set up a re-weighting array. Then each $P_T(W)$ in the file would have its weight be multiplied by the corresponding 'smearing' weight, to produced a smeared $P_T(W)$. This was then to be used to fill a new transverse mass histogram. The deviation of this histogram from the original histogram would be retained, and this process repeated at least 10,000 times. This would generate an histogram with a width representing the uncertainty in the transverse mass of the $W^{\pm}$.

This part of the project was not completed because it proved too difficult to diagonalise the covariant matrix in the time remaining. There was a FORTRAN program that would diagonalise the matrix, but integrating it into the C++ code was tricky, and the FORTRAN program would crash when the code executed; having very little experience of FORTRAN, this eventually was too difficult a task to complete with the time remaining. Had there been more time available, then the program would more than likely have been finished.

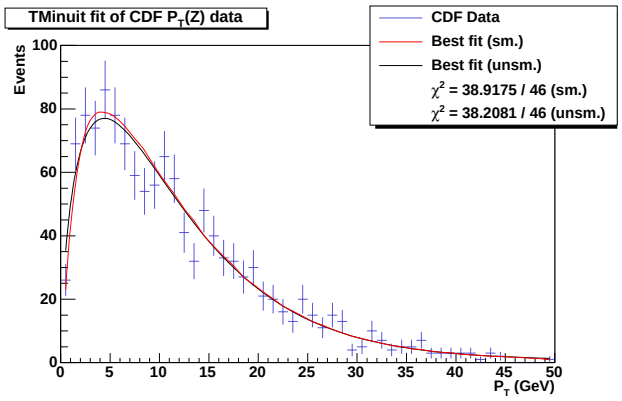


Figure 22: The best fit of the smeared functional form to CDF $P_T(Z)$ data. The pre-simulation functional form has been plotted for comparison. The χ^2 for both has been shown for reference, along with the number of degrees of freedom for the fit.

Parameter	Value
P_1	0.1311 ± 0.0103
P_2	0.0302 ± 0.0262
P_3	6.8299 ± 0.4895
P_4	0.6070 ± 0.0607

Table 1: The parameters for the best fitting smeared functional form to the CDF $P_T(Z)$ data.

5 Results

The parameters that gave the best smeared functional form fit to the data are listed in table 5. The χ^2 of the this fit to the CDF was 38.9175 with 46 degrees of freedom. This corresponds to passing a 76% significance test. However, the original function form had a χ^2 value of 38.2081 with the same 46 degrees of freedom, passing a 78% significance test. How much can be read into this is unclear, a visual inspection of the fit, figure 22, seems to indicate that the smeared form fits the peak better than the original form. However, the CDF data in the 7–15 GeV range is erratic. More data would be required before anything more definite could be said of the detector simulation.

6 Conclusion

The project has shown that the transverse momentum of the Z^0 can be used to represent the $W^{\pm}$. This was done by analysing Monte Carlo events generated by HERWIG.

The effect of the resolution of the CDF's Central Electromagnetic calorimeter upon various properties of the Z^0 was also investigated using HERWIG data.

How the detector simulation depended upon κ_{CEM} was investigated, and found to be significant. In order to find this κ_{CEM}, a bicubic interpolation fit was made to find the rest mass distribution of the Z^0 that contained the κ_{CEM} that best described the data. This task was not as successful as hoped, but by no means a failure. A value of κ_{CEM} was found to be $1.5105\% \pm 0.2299\%$, and when the uncertainty is considered, it is close to a previous derived estimate of 1.742%. However, latter stages of the project were already being worked upon before the bugs in the fitting program were corrected. Nevertheless, the program did eventually return answers with reasonable errors. This was more than likely the result of a lack of experience in C++ programming; a person more experienced in the language would have built exception handling into the program from start, which may have reduced the time spent chasing bugs and errors.

The value of κ_{CEM} was used to complete the modelling the CEM, and this was used with an *ad hoc* functional form to build a theoretical prediction for the observed transverse momentum of the Z^0. This was fitted against CDF data to find the four parameters that generated the best form; these parameters were reasonable with believable errors. The χ^2 for the fit was respectable, and it passed a 76% significance test.

A estimate for the corresponding uncertainty on the measured transverse mass of the $W^{\pm}$ was not completed; there was not enough time to complete the program, and integrating the FORTRAN subroutines into C++ was tricky. With more experience of FORTRAN, the analysis should have been completed, it was the implementation that caused the delay in working.

Is believed that whether the CEM simulation is valid, or not, could only be determined with more transverse momenta data from the CDF. The χ^2 fits of the smeared and original functional form both fit the data, but the error bars on the data are large enough to allow this.

Throughout the project work was hampered by bug and other programming related issues. Given that C++ was an unfamiliar language at the start of the project, progress was slower than liked when faced with difficult programming tasks. However, the language was picked up, with analysis programs being written from scratch, which could generate histograms that could be viewed in Root immediately.

It is hoped that the final section of the project could have been completed within a fortnight if it were to be continued.

Modelling some of the other detector, such as the Plug EM calorimeter, would improve the simulation; the effect of κ_{PEM} was significant when the PEM was

modelled along with the CEM. Taking other detector effects into account would also be beneficial.

A Software used

A.1 HERWIG

HERWIG (Hadron Emission Radiation With Interfering Gluons) is a Monte Carlo event generator. It simulates events using the probabilities that a particle will decay via a particular decay mode.

A.1.1 Monte Carlo numbering scheme

In an effort to maintain a standard of particle identification for event generators, a numbering system was devised [4]. A negative number represents an antiparticle.

Particle	ID	Particle	ID	Particle	ID
d	1	e^-	11	γ	22
u	2	ν_e	12	Z^0	23
s	3	μ^-	13	W^+	24
c	4	ν_μ	14		
b	5	τ^-	15		
t	6	ν_τ	16		

Table 2: Some particle IDs in the Monte Carlo numbering system.

A.2 Root

One of the major packages used was Root. Root was developed by Rene Brun and Fons Rademakers to provide an object-orientated programming framework from which they could link together software together.

Root is freely available for download at the CERN's Root website http://root.cern.ch

A.2.1 Fitting with Root - `TMinuit`

Root is equipped with a minimisation package, `TMiniut`, which takes operates upon a static external function written by the user. The class is a C++ wrapper around an older FORTRAN program `Minuit`.

The user created the function to be minimised. For example, to find the minimum of $f(x) = x^2$:

```
static void myFCN( Int_t &npar , Double_t* gin , Double_t &f ,
Double_t* par , Int_t flag)
{
  f=0;
  double x = par[0];
  f = x*x;
}
```

A `TMinuit` object would then be created and the minimisation algorithm invoked.

```
int main()
{
   // Create object with number of paramerter to minimise
   TMinuit minuit = new TMinuit(1)

   // Set a pointer to the user function
   minuit->SetFCN(myFCN);
   int ierflg  = 0;

   // Define the parameter to be minmised
   string name = "x";
   start       = 10;
   upper_lim   = 20;
   lower_lim   = -20;
   step        = 0.1;

   // Set the level of feedback (1=normal,0=no warnings,-1=nothing)
   minuit->SetPrintLevel(1);

   // Input the parameter into TMinuit
   minuit->mnparm(0,start.c_str(),start,step,lower_limn,upper_lim,ierflg)

   // Error checking to use (1=Chi-squared)
   minuit->SetErrorDef(1);

   // Maximum number of iterations
   minuit->SetMaxIterations(500);

   // Invokes the Migrad algorithm and minimises myFCN
   minuit->Migrad();
}
```

The Migrad algorithm would then find the value of x where $f(x)$ is a minimum, which should be 0 in this case. More information about `TMinuit` can be found in the Root '*Users Guide*' [5]

B Source Code

B.1 Initial HERWIG analysis

The following code was used to investigate the energies, transverse momenta, pseudorapidities of the electrons and positrons that are produced from the decay of the $W^{\pm}$ and Z^0. It was part of a larger framework that was written to read and write to Root files.

The section of code was run for every event in the file.

```
  // Loop over all the particles and select the highest Et electron in the
  // event
  int np =  _HepEvtBlock.hepevt.nhep;
  double ptmax_e = -10.0;
  double ptmax_p = -10.0;
  double rap_e = -100.0;
  double rap_p = -100.0;
  double ptZ = 0.0;
  double emax_e = -1.0;
  double emax_p = -1.0;
  for (int i = 0; i < np; i++){
    if (_HepEvtBlock.hepevt.id_pdg[i] == 11 ||
_HepEvtBlock.hepevt.id_pdg[i] == -11) {
      double px2 = _HepEvtBlock.hepevt.px[i]*_HepEvtBlock.hepevt.px[i];
      double py2 = _HepEvtBlock.hepevt.py[i]*_HepEvtBlock.hepevt.py[i];
      double pz2 = _HepEvtBlock.hepevt.pz[i]*_HepEvtBlock.hepevt.pz[i];
      double energy = _HepEvtBlock.hepevt.e[i];
```

The `_HepEvtBlock` is a custom class that defines a struct which contains all the various information a Root file.

```
      double pt = TMath::Sqrt(px2 + py2);
      double p  = TMath::Sqrt(px2 + py2 + pz2);
      double cos_theta = (_HepEvtBlock.hepevt.pz[i])/p;
      double angle = TMath::ACos(cos_theta);
      double debugger = TMath::Tan(angle/2);
      double prap = (-1)*TMath::Log(debugger);
      if (pt > ptmax_e && _HepEvtBlock.hepevt.id_pdg[i] == 11) {
      // electron
        ptmax_e = pt;
        rap_e = prap;
        emax_e = energy;
      } else if (pt > ptmax_p && _HepEvtBlock.hepevt.id_pdg[i] == -11){
      // positron
        ptmax_p = pt;
        rap_p = prap;
        emax_p = energy;
      }
      ptZ =TMath::Abs(ptmax_e - ptmax_p); //PT of the z-boson
    }
  }
  rf_out = AnalysisSteer::Instance()->getOutputRootFile();
```

```
  gH->Hf1(1,ptmax_e);
  gH->Hf1(2,ptmax_p);
  gH->Hf1(3,rap_e);
  gH->Hf1(4,rap_p);
  gH->Hf1(5,emax_e);
  gH->Hf1(6,emax_p);
  gH->Hf1(7,emax_e+emax_p);
  gH->Hf1(8,emax_e+emax_p);
```

The properties in the event are binned into histograms.

B.2 Detector simulation

This code is a basic simulation of the CEM and PEM. It runs inside a larger program (as with the code segment in §B.1) which reads a Root file containing approximately 60000 HERWIG $Z^0 \rightarrow e^- e^+$ events.

```
  // set the CEM and PEM terms

  double termCEM  = 0.135;
//  double kappaCEM = 0.025;
  int num_kappaCEM= 6;
  double kappaCEM[6] = {0.005,0.01,0.015,0.02,0.025,0.03};

  double termPEM  = 0.16;
//  double kappaPEM[8] = {0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08};
  double kappaPEM = 0.032;
  int num_kappaPEM = 8;

  bool is_CEM = false;          // CEM truth check
  bool is_PEM = false;          // PEM truth check
  bool is_insiderange = true;   // Is particle inside the detection range?

  // Loop over all the particles and select the highest Et electron in the
  // event
  int np =  _HepEvtBlock.hepevt.nhep;

  //Empty objects for the final particles
  TLorentzVector electron;
  TLorentzVector positron;
  TLorentzVector zboson;
  TLorentzVector electron_sm[num_kappaCEM];
  TLorentzVector positron_sm[num_kappaCEM];
  TLorentzVector zboson_sm[num_kappaCEM];
```

The `TLorentzVector` object is a custom Root class that can contain a position or momentum 4-vector.

```
  for (int i = 0; i < np; i++){
    if (_HepEvtBlock.hepevt.id_pdg[i] == 11 ||
                            _HepEvtBlock.hepevt.id_pdg[i] == -11) {
      TLorentzVector particle;
```

```
      TLorentzVector particle_sm[num_kappaCEM];
      particle.SetPxPyPzE(_HepEvtBlock.hepevt.px[i],
                          _HepEvtBlock.hepevt.py[i],
                          _HepEvtBlock.hepevt.pz[i],
                          _HepEvtBlock.hepevt.e[i]);

      // Time to mess up the energy

      double E_in = particle.E();
      double r_gaussCEM = gRandom->Gaus(0,1);
      double r_gaussPEM = gRandom->Gaus(0,1);
      double pxyz[3] = {particle.Px(),particle.Py(),particle.Pz()};
      double pxyz_smearCEM[3][num_kappaCEM];
      double pxyz_smearPEM[3][num_kappaCEM];
      double m0 = 0.511e-3; // GeV
      double E_T = TMath::Sqrt(pxyz[0]*pxyz[0] + pxyz[1]*pxyz[1]);
      double E_z = pxyz[2];

      double deltaE_CEM[num_kappaCEM];
      double E_smearCEM[num_kappaCEM];
      double deltaE_PEM[num_kappaCEM];
      double E_smearPEM[num_kappaCEM];

      // Check whether the CEM or PEM picks it up
      double eta = particle.PseudoRapidity();
      is_CEM = TMath::Abs(eta)<=1 ;
      is_PEM = TMath::Abs(eta)<=2.4 && TMath::Abs(eta)>1;

      for (int k=0;k<num_kappaCEM;k++) { // loop through my kappa array

         // CEM energy calculations
         deltaE_CEM[k] = TMath::Sqrt(((termCEM*termCEM)/E_T) +
         (kappaCEM[k]*kappaCEM[k]));
         E_smearCEM[k] = E_in*(1+(r_gaussCEM*deltaE_CEM[k]));

         // PEM energy calculations
         deltaE_PEM[k] =
                TMath::Sqrt(((termPEM*termPEM)/E_in)+(kappaPEM*kappaPEM));
         E_smearPEM[k] = E_in*(1+(r_gaussPEM*deltaE_PEM[k]));

         // Momentum (CEM and PEM)
 for (int j = 0; j < 3; j++) {
    pxyz_smearCEM[j][k] = pxyz[j] *
(1+(r_gaussCEM*((E_in*deltaE_CEM[k])/(TMath::Sqrt(E_in*E_in-m0*m0)))));
    pxyz_smearPEM[j][k] = pxyz[j] *
(1+(r_gaussPEM*((E_in*deltaE_PEM[k])/(TMath::Sqrt(E_in*E_in-m0*m0)))));
 }

         //...and put into the smeared particle
         if (is_CEM) particle_sm[k].SetPxPyPzE(pxyz_smearCEM[0][k],
                                               pxyz_smearCEM[1][k],
                                               pxyz_smearCEM[2][k],
                                               E_smearCEM[k]);
         if (is_PEM) particle_sm[k].SetPxPyPzE(pxyz_smearPEM[0][k],
```

```
                                                pxyz_smearPEM[1][k],
                                                pxyz_smearPEM[2][k],
                                                E_smearPEM[k]);
    }

    // Identify and dump....

    if (particle.Pt() > electron.Pt() &&
        _HepEvtBlock.hepevt.id_pdg[i] == 11 ) {
       electron = particle;
       for (int k=0;k<num_kappaCEM;k++) {
          electron_sm[k] = particle_sm[k];
       }
    } else if (particle.Pt()>positron.Pt() &&
               _HepEvtBlock.hepevt.id_pdg[i] == -11 ) {
       positron = particle;
       for (int k=0;k<num_kappaCEM;k++) {
          positron_sm[k] = particle_sm[k];
       }
     }
   }
}

// The electon and positron have been identified, now to combine to
// form the z bosons

zboson     = electron + positron;
for (int k=0;k<num_kappaCEM;k++) {
   zboson_sm[k] = electron_sm[k] + positron_sm[k];
}

// Final pseudorapidity cut, are they in the range |eta| <= 2.4

double eta_e = electron.PseudoRapidity();
double eta_p = positron.PseudoRapidity();
is_insiderange = TMath::Abs(eta_e)<=2.4 && TMath::Abs(eta_p) <= 2.4;

// Invariant mass calc.

double ze_sm[num_kappaCEM];
double zp_sm[num_kappaCEM];
double ze = zboson.E();
double zp = zboson.P();
double zmass_sm[num_kappaCEM];
double zmass = TMath::Sqrt(ze*ze - zp*zp);
for (int k=0;k<num_kappaCEM;k++) {
   ze_sm[k] = zboson_sm[k].E();
   zp_sm[k] = zboson_sm[k].P();
   zmass_sm[k] = TMath::Sqrt(ze_sm[k]*ze_sm[k] - zp_sm[k]*zp_sm[k]);
}

rf_out = AnalysisSteer::Instance()->getOutputRootFile();

if (is_insiderange) {
```

```
    gH->Hf1(10,zboson.Pt());
    gH->Hf1(11,zboson_sm[0].Pt());
    gH->Hf1(12,zboson_sm[1].Pt());
    gH->Hf1(13,zboson_sm[2].Pt());
    gH->Hf1(14,zboson_sm[3].Pt());
    gH->Hf1(15,zboson_sm[4].Pt());
    gH->Hf1(16,zboson_sm[5].Pt());

    gH->Hf1(20,zboson.E());
    gH->Hf1(21,zboson_sm[0].E());
    gH->Hf1(22,zboson_sm[1].E());
    gH->Hf1(23,zboson_sm[2].E());
    gH->Hf1(24,zboson_sm[3].E());
    gH->Hf1(25,zboson_sm[4].E());
    gH->Hf1(26,zboson_sm[5].E());

    gH->Hf1(30,zmass);
    gH->Hf1(31,zmass_sm[0]);
    gH->Hf1(32,zmass_sm[1]);
    gH->Hf1(33,zmass_sm[2]);
    gH->Hf1(34,zmass_sm[3]);
    gH->Hf1(35,zmass_sm[4]);
    gH->Hf1(36,zmass_sm[5]);

    // Sanity tests
    double E = zboson.E();
    double E_sm[num_kappaCEM];
    double sanity[num_kappaCEM];
    for (int k=0;k<num_kappaCEM;k++){
       E_sm[k] = zboson_sm[k].E();
       sanity[k] = (E - E_sm[k])/E;
    }
```

This checks the amount of energy applied or removed from the particle because of the detector smearing, which should result in a Gaussian distribution.

```
    gH->Hf1(90,sanity[0]);
    gH->Hf1(91,sanity[1]);
    gH->Hf1(92,sanity[2]);
    gH->Hf1(93,sanity[3]);
    gH->Hf1(94,sanity[4]);
    gH->Hf1(95,sanity[5]);
  }
  rf_out.cd();
  incrementEventCount();
}
```

B.3 Rest mass-κ fit

B.3.1 main.cc

This is the main program for the fitting program. It sets up the required objects needed to compute the interpolation fitting, then invokes `TMinuit` to complete the

fit.

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <vector>
#include <stdlib.h>
#include <sstream>

#include "TROOT.h"
#include "realData.hh"
#include "masterIndex.hh"
#include "iArray.hh"
#include "numRep.hh"
#include "fit.hh"
#include "TAxis.h"
#include "profile.hh"
#include "TMinuit.h"
#include "ControlCards.hh"
#include "TGraph.h"
#include "TCanvas.h"
#include "TLegend.h"

static realData* CDF_MZ;
static iArray* bigArray;
static profile* current;
static bool isDefaultRange;
static double fitMin,fitMax,norm;
static string sssss;
static string fitType;
static TH1D* actual;

using namespace std;

static void myFCN( Int_t &npar , Double_t* gin , Double_t &f ,
Double_t* par ,  Int_t flag )
  // Function minimised by Minuit.
{
  f=0;
  double xIn = par[0];
  double yIn = par[1];
  current = bigArray->GetNormZProfileForXY(xIn,yIn,sssss);
  double* chi = current->Fit(actual,norm,fitMin,fitMax,fitType);
  f = chi[0];
}

int main () {

  // Define the variables to be read from the cards
  ControlCards cCards;
  cCards.define("EXP_DATA_FILE","");
  cCards.define("HISTO_FILE","");
  cCards.define("MASTER_ID","");
```

```
cCards.define("X_PARAM_NAME","");
cCards.define("X_PARAM_START",(float) 0);
cCards.define("X_PARAM_MIN",(float) 0);
cCards.define("X_PARAM_MAX",(float) 0);
cCards.define("X_PARAM_STEP",(float) 0);
cCards.define("Y_PARAM_NAME","");
cCards.define("Y_PARAM_START",(float) 0);
cCards.define("Y_PARAM_MIN",(float) 0);
cCards.define("Y_PARAM_MAX",(float) 0);
cCards.define("Y_PARAM_STEP",(float) 0);
cCards.define("MAX_ITER",(int) 500);
cCards.define("M_ERROR",(float) 0.5);
cCards.define("MIN_TYPE",(float) 1.0);
cCards.define("M_FEED",(int) 1);
cCards.define("DEFAULT_RANGE",(int) 1);
cCards.define("FIT_MAX_X",(float) 0);
cCards.define("FIT_MIN_X",(float) 0);
cCards.define("OUTPUT_FILE","");
cCards.define("OUTPUT_ROOT",(int) 0);
cCards.define("HISTO_NAME","");
cCards.define("BIN_TYPE","");
cCards.define("Z_AXIS","");
cCards.define("Y_AXIS","");
cCards.define("X_AXIS","");
cCards.Read();
cCards.Print();

bool isSavingOutput = cCards.IntValue("OUTPUT_ROOT")==1;
sssss = cCards.StringValue("BIN_TYPE");
if (cCards.FloatValue("M_ERROR")==0.5) fitType="LOG";
if (cCards.FloatValue("M_ERROR")==1) fitType="CHI";

// File path

string fileInput = cCards.StringValue("EXP_DATA_FILE");
cout << "Fitter> Creating realData object" << endl; ;
CDF_MZ = new realData(fileInput,false);
cout << "Fitter> Initate with `" << fileInput << "'" << endl;

// Set up MasterID array

string rootFileName = cCards.StringValue("HISTO_FILE");
string mode = "READ";
string title = "Master ID";
cout << "Fitter> Opening " << rootFileName;
cout << " to find the master ID" << endl;
TFile *rootFile =
new TFile(rootFileName.c_str(),mode.c_str(),title.c_str(),0);

if (!rootFile->IsOpen()) {
  cout << "Fitter> Error file not found" << endl;
}

cout << "Fitter> Extracting the index histogram" << endl;
```

```
TH2F *masterHis =
(TH2F*) rootFile->Get((cCards.StringValue("MASTER_ID")).c_str());
masterIndex *MasterID = new masterIndex(masterHis);
masterHis->~TH2F();

// Set up interpolation array
// It should be noted that the program will assume that all the histograms
// in the index have identical x-axes

cout << "Fitter> Creating the interpolation array" << endl;
TH1D *dummy = (TH1D*) rootFile->Get("h1");
int zBins = dummy->GetNbinsX();
TAxis *Xaxis = dummy->GetXaxis();
double zMin = Xaxis->GetXmin();
double zMax = Xaxis->GetXmax();
double zDelta = (zMax-zMin)/zBins;
dummy->~TH1D();

// Turn data into a histogram and get the normalisation factor
actual = CDF_MZ->getHistogram("act","Actual data",zBins,zMin,zMax);
for (int i=0;i<=zBins;i++) {
  norm+= (actual->GetBinContent(i))*zDelta;
}

// now we have both ranges, choose which to use
isDefaultRange =  cCards.IntValue("DEFAULT_RANGE")==1;
if (!isDefaultRange) {
  fitMin = (double) cCards.FloatValue("FIT_MIN_X");
  fitMax = (double) cCards.FloatValue("FIT_MAX_X");
} else {
  fitMin = zMin;
  fitMax = zMax;
}

cout << "Fitter> zBins = " << zBins << "\tzMin = " << zMin;
cout << "\tzMax = " << zMax << "\tBin Width = " << zDelta << endl;
bigArray = new iArray(rootFile,MasterID,zBins,zMin,zMax,zDelta);
cout << "Fitter> Root file closed" << endl;

if(!isDefaultRange) {
  cout << "Fitter> Minuit minimisation with custom range: fitMin=";
  cout <<fitMin << " fitMax=" << fitMax << endl;
} else {
  cout << "Fitter> Minuit minimisation with default range: fitMin=";
  cout << zMin << " fitMax=" << zMax << endl;
}

// Define Minuit
Int_t pm = 2;
TMinuit *minuit = new TMinuit(pm);
minuit->SetFCN(myFCN);
int ierflg = 0;
```

```
// Minimisation Parameters
string* name = new string[2];
name[0] = cCards.StringValue("X_PARAM_NAME");
name[1] = cCards.StringValue("Y_PARAM_NAME");
double* start_values = new double[2];
start_values[0] = (double) cCards.FloatValue("X_PARAM_START");
start_values[1] = (double) cCards.FloatValue("Y_PARAM_START");
double* lim_min = new double[2];
lim_min[0] = (double) cCards.FloatValue("X_PARAM_MIN");
lim_min[1] = (double) cCards.FloatValue("Y_PARAM_MIN");
double* lim_max = new double[2];
lim_max[0] = (double) cCards.FloatValue("X_PARAM_MAX");
lim_max[1] = (double) cCards.FloatValue("Y_PARAM_MAX");
double* step = new double[2];
step[0] = (double) cCards.FloatValue("X_PARAM_STEP");
step[1] = (double) cCards.FloatValue("Y_PARAM_STEP");
Double_t* val = new double[1];

// Set parameters
minuit->SetPrintLevel(cCards.IntValue("M_FEED"));

minuit->mnparm(0,name[0].c_str(),start_values[0],step[0],lim_min[0],
lim_max[0],ierflg);
minuit->mnparm(1,name[1].c_str(),start_values[1],step[1],lim_min[1],
lim_max[1],ierflg);

minuit->SetErrorDef(cCards.FloatValue("M_ERROR"));
minuit->SetMaxIterations(cCards.IntValue("MAX_ITER"));
val[0] = (Double_t) cCards.FloatValue("MIN_TYPE");
minuit->mnexcm("SET STR",val,1,ierflg);
minuit->Migrad();

double xFit,xErr,yFit,yErr;

minuit->GetParameter(0,xFit,xErr);
minuit->GetParameter(1,yFit,yErr);

cout << "Fitter> Finished with X=" << xFit << " error=" << xErr;
cout <<  " and Y=" << yFit << " error=" << yErr << endl;

  string bit = cCards.StringValue("HISTO_NAME");
  string sig,sigf;
  string s1 = "bfH_";
  string s1f = s1.append(bit);
  profile* bestFit = bigArray->GetNormZProfileForXY(xFit,yFit,sssss);
  TH1D* bfH = bestFit->getHistogram(s1f.c_str(),"Best Fit");
  double* chi = bestFit->Fit(actual,norm,fitMin,fitMax,fitType);

cout << "Fitter> Goodness of fit: chi^2 = " << chi[0] << endl;
cout << "Fitter> Degrees of freedom = " << chi[1] << endl;
cout << "Fitter> Probability test = " << chi[2] << endl;

if(isSavingOutput) {
```

```
double d;

stringstream ss;
ss << "#chi^{2} = " << chi[0] << " / " << chi[1] << endl;
string chistr = ss.str();
TH1D* chichi = new TH1D("chi",chistr.c_str(),1,1,1);
chichi->SetLineColor(0);

bfH->Scale(norm);

s1 = "act2_";
s1f = s1.append(bit);
TH1D* act2 =
new TH1D(s1f.c_str(),"Actual Data in fit range",zBins,zMin,zMax);

for (int i=0;i<CDF_MZ->numItems();i++){
  d = CDF_MZ->getDataElement(i);
  act2->Fill(d);
}
for (int z=0;z<zBins;z++) {
  norm+=(act2->GetBinContent(z+1))*zDelta;
}
s1 = "act1_";
s1f = s1.append(bit);
TH1D* act1 =
new TH1D(s1f.c_str(),"Actual Data in fit range",zBins,zMin,zMax);
for (int ii=0;ii<zBins;ii++){
  d = act2->GetBinContent(ii+1);
  act1->SetBinContent(ii,d/norm);
}

s1 = "ovl_";
s1f = s1.append(bit);
TCanvas* ovl1 = new TCanvas(s1f.c_str(),"Fitting output");
ovl1->SetFillColor(0);
bfH->SetLineColor(2);
act2->GetXaxis()->SetTitle((cCards.StringValue("Z_AXIS")).c_str());
act2->Draw("e0");
bfH->Draw("c same");
TLegend* leg = new TLegend(0.78,0.80,1,1);
leg->AddEntry(act2,"Raw data","L");
leg->AddEntry(bfH,"Best Fit","L");
leg->AddEntry(chichi,chistr.c_str(),"L");
leg->SetFillColor(0);
leg->SetTextFont(62);
leg->SetTextSize(0.04);
leg->Draw();
ovl1->SetBorderMode(0);
ovl1->Draw();

string fName = cCards.StringValue("OUTPUT_FILE");

TFile* f2 = new TFile(fName.c_str(),"RECREATE");
act1->Write();
```

```
    act2->Write();
    bfH->Write();
    ovl1->Write();
    f2->Close();

    cout << "Fitter> Output written to " << fName << endl;
  }
}
```

B.3.2 realData.hh

The class written to store the data from the CDF.

```
#ifndef realData_HH
#define realData_HH

#include "TH1D.h"

using namespace std;

class realData {
 private:
  vector<double> Mz_CDF;
  int items;
 public:
  realData();
  realData( string , bool );
  ~realData();
  double getDataElement( int );
  void setDataElement( int, double );
  void create( string, bool );
  int numItems();
  TH1D* getHistogram(string id,string name,int bins,double min,double max);
};
#endif
```

B.3.3 realData.cc

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <vector>
#include <stdlib.h>

#include "realData.hh"

realData::realData() {
  items = 0;
}
realData::realData( string fileInput, bool showNums ) {

  string buffer;
  string buffer2;
```

```
  cout << "realData> Attempting to open " << fileInput << "...\n";

  ifstream file (fileInput.c_str());
  cout << "realData> File opened.\n";
  cout << "realData> Attempt to read file...\n";

  if (!file){
    cout << "realData> File not found.\n";
    cout << "realData> Aborting...\n\n";
  }

  items = 0;
  char buffArray[100];
  char *stopchar = "";
  int i=0;

  while (file >> buffer) {
    strcpy(buffArray, buffer.c_str());
    Mz_CDF.insert(Mz_CDF.end(),strtod(buffArray,&stopchar));
    items++;
    if (showNums) cout << buffer << "\n";
    if (showNums) cout << Mz_CDF[i] << "\n";
    i++;
  }

  cout << "realData> " << items << " entries, storing data to array...\n";

  file.close();

  cout << "realData> File closed.\n";
}

realData::~realData() {
}

double realData::getDataElement( int id ) {
  if (id < Mz_CDF.size()) return Mz_CDF[id];
  else {
    cout << "realData>Error index out of bounds." << endl;
    cout << "realData> id=" << id << " size of array=" << items << endl;
    return 0;
  }
}

void realData::setDataElement( int id, double value ) {
  if (id < Mz_CDF.size()) {
    Mz_CDF[id] = value;
  }
  else {
    cout << "realData>Error index out of bounds." << endl;
    cout << "realData> id=" << id << " size of array=" << items << endl;
  }
}
```

```
int realData::numItems() {
  return items;
}

TH1D* realData::getHistogram( string id,string name,int bins,double min,
 double max ) {
  TH1D* histo = new TH1D(id.c_str(),name.c_str(),bins,min,max);
  for (int i=0;i<this->numItems();i++) {
    histo->Fill(this->getDataElement(i));
  }
  return histo;
}
```

B.3.4 masterIndex.hh

The class that created the index of histograms. It reads the Root file and creates an array, which relates the particular value of M_Z and κ to a named histogram in the file.

```
#ifndef masterIndex_HH
#define masterIndex_HH

#include "TH2F.h"

using namespace std;

class masterIndex {
 private:
  int *ID;
  int nXbins;
  int nYbins;
  double deltaX;
  double deltaY;
  double Xmin;
  double Xmax;
  double Ymin;
  double Ymax;
 public:
  masterIndex();
  masterIndex( TH2F* );
  ~masterIndex();
  inline int GetNbinsX() { return nXbins; }
  inline int GetNbinsY() { return nYbins; }
  inline double GetXMin() { return Xmin; }
  inline double GetXMax() { return Xmax; }
  inline double GetYMin() { return Ymin; }
  inline double GetYMax() { return Ymax; }
  inline double GetXDelta() { return deltaX; }
  inline double GetYDelta() { return deltaY; }
  int GetHistID ( double , double );
  int GetHistID ( float , float );
  int GetHistID ( int , int );
  int GetHistIDwBin ( int , int );
```

```
  string myCast ( int );
  string myCastH ( int );
};

#endif
```

B.3.5 masterIndex.cc

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <vector>
#include <stdlib.h>

#include "TROOT.h"
#include "TFile.h"
#include "TKey.h"
#include "masterIndex.hh"
#include "TMath.h"

masterIndex::masterIndex() {
}

masterIndex::masterIndex( TH2F *masterHis ) {

  cout << "masterIndex> Creating MasterID matrix\n";

  nXbins = masterHis->GetNbinsX();
  nYbins = masterHis->GetNbinsY();

  TAxis *Xaxis = masterHis->GetXaxis();
  TAxis *Yaxis = masterHis->GetYaxis();

  Xmin = Xaxis->GetXmin();
  Xmax = Xaxis->GetXmax();
  Ymin = Yaxis->GetXmin();
  Ymax = Yaxis->GetXmax();

  Xaxis->~TAxis();
  Yaxis->~TAxis();

  deltaX = ( Xmax - Xmin ) / nXbins ;
  deltaY = ( Ymax - Ymin ) / nYbins ;

  cout << "masterIndex> " << this->GetNbinsX() << " bins for X and ";
  cout << this->GetNbinsY()  << " for Y\n";
  cout << "masterIndex> X min = " << this->GetXMin() <<" \t X max = ";
  cout << this->GetXMax() << "\n";
  cout << "masterIndex> X bin width = " << this->GetXDelta() << "\n";
  cout << "masterIndex> Y min = " << this->GetYMin() <<" \t Y max = "
  cout << this->GetYMax() << "\n";
  cout << "masterIndex> Y bin width = " << this->GetYDelta() << "\n";
```

```
  ID = new int[nXbins * nYbins];

  for (int i=0;i<nXbins;i++) {
    for (int j=0;j<nYbins;j++) {
      ID[(i*nYbins) + j] = (int) masterHis->GetBinContent(i+1,j+1);
    }
  }
}

masterIndex::~masterIndex() {
}

int masterIndex::GetHistID( int xIn, int yIn ){
  return GetHistID((double)xIn,(double)yIn);
}

int masterIndex::GetHistID( float xIn, float yIn ){
  return GetHistID((double)xIn,(double)yIn);
}

int masterIndex::GetHistID( double xIn, double yIn ){
  if (xIn<Xmin || xIn>Xmax || yIn<Ymin || yIn>Ymax) {
    cout << "masterIndex> getHisID(" << xIn << "," << yIn;
    cout << ") outside range\n";
    return 0;
  }
  int xBin = (int)TMath::Floor((xIn-Xmin) / deltaX);
  int yBin = (int)TMath::Floor((yIn-Ymin) / deltaY);
  return ID[xBin*nYbins + yBin];
}

int masterIndex::GetHistIDwBin( int xBinNum, int yBinNum ){
  return ID[xBinNum*nYbins + yBinNum];
}

// Not sure how to cast from int to char*, so I wrote my own crude methods.

string masterIndex::myCastH ( int input ) {
  string s;
  s = "h";
  if(TMath::Floor(input/100)) {
    int num = (int)TMath::Floor(input/100);
    s += myCast(num);
    input-=(num*100);
  }
  if(TMath::Floor(input/10)) {
    int num = (int)TMath::Floor(input/10);
    s += myCast(num);
    input-=(num*10);
  }
  if(input) {
    s += myCast(input);
  } else {
```

```
    s += "0";
  }
  return s;
}

string masterIndex::myCast( int digit ){
  string s;
  switch (digit) {
  case 0:
    s = "0";
    break;
  case 1:
    s = "1";
    break;
  case 2:
    s = "2";
    break;
  case 3:
    s = "3";
    break;
  case 4:
    s = "4";
    break;
  case 5:
    s = "5";
    break;
  case 6:
    s = "6";
    break;
  case 7:
    s = "7";
    break;
  case 8:
    s = "8";
    break;
  case 9:
    s = "9";
    break;
  default:
    s = "NaN";
    break;
  }
  return s;
}
```

B.3.6 iArray.hh

This class uses the `masterIndex` class to create the 3D array of data in the Root file. It extracts all the information from the Root file to hold in memory. It contains a method that will return a normalised mass distribution for any pairing of M_Z and κ (while it can extrapolate profiles, they are extremely unreliable, thus it should only be used to interpolate inside the data range)

```
#ifndef iArray_HH
#define iArray_HH

#include "profile.hh"
#include "TROOT.h"
#include "TFile.h"

using namespace std;

class iArray {
 private:
 double* array;
     int zBins;
  double zDelta;
  double zMin;
  double zMax;
     int xBins;
  double xDelta;
  double xMin;
  double xMax;
     int yBins;
  double yDelta;
  double yMin;
  double yMax;
 public:
  iArray();
  iArray( TFile* , masterIndex* , int , double , double , double );
  ~iArray();
  inline int GetNbinsX() { return xBins; }
  inline int GetNbinsY() { return yBins; }
  inline int GetNbinsZ() { return zBins; }
  inline double GetXMin() { return xMin; }
  inline double GetYMin() { return yMin; }
  inline double GetZMin() { return zMin; }
  inline double GetXMax() { return xMax; }
  inline double GetYMax() { return yMax; }
  inline double GetZMax() { return zMax; }
  inline double GetXDelta() { return xDelta; }
  inline double GetYDelta() { return yDelta; }
  inline double GetZDelta() { return zDelta; }
  inline double GetBinContent( int i ) { return array[i]; }
  inline int GetTotalNumEntries() { return xBins*yBins*zBins; }
  inline double* GetXAxisValues() { return this->GetXAxisValues(""); }
  inline double* GetYAxisValues() { return this->GetYAxisValues(""); }
  inline double* GetZAxisValues() { return this->GetZAxisValues(""); }
  double* GetXAxisValues( string );
  double* GetYAxisValues( string );
  double* GetZAxisValues( string );
  double* GetXYPlaneWBin( int );
  profile* GetNormZProfileForXY ( double , double , string  );
  double GetXYZBin( int , int , int );
  double GetXYZBin( float, float , float );
  double GetXYZBin( double , double , double );
  void CreateSplines();
```

```
  void Dump();
};

#endif
```

B.3.7 iArray.cc

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <vector>
#include <stdlib.h>

#include "TROOT.h"
#include "TH1.h"
#include "TH1D.h"
#include "masterIndex.hh"
#include "numRep.hh"
#include "profile.hh"

#include "iArray.hh"

iArray::iArray() {
}

iArray::iArray(TFile *rootFile , masterIndex *mID , int zBinsI ,
double zMinI , double zMaxI , double zDeltaI ) {
  zBins = zBinsI;
  zMin = zMinI;
  zMax = zMaxI;
  zDelta = zDeltaI;

  xBins  = mID->GetNbinsX();
  xDelta = mID->GetXDelta();
  xMin   = mID->GetXMin();
  xMax   = mID->GetXMax();
  yBins  = mID->GetNbinsY();
  yDelta = mID->GetYDelta();
  yMin   = mID->GetYMin();
  yMax   = mID->GetYMax();

  cout << "iArray> Creating interpolation array\n";

  array = new double[xBins*yBins*zBins];
  for (int x=0;x<xBins;x++) {
    for (int y=0;y<yBins;y++){
      string s = mID->myCastH(mID->GetHistIDwBin(x,y));
      TH1D *f = (TH1D*) rootFile->Get(s.c_str());
      for (int z=0;z<zBins;z++) {
int where = (zBins*yBins*x) + (y*zBins) + z;
array[where] = (double) f->GetBinContent(z+1);
      }
```

```
      f->~TH1D();
    }
  }

  cout << "iArray> Interpolation array created\n";
  cout << "iArray> xBins = " << this->GetNbinsX() << "\tyBins = ";
  cout << this->GetNbinsY() << "\tzBins = " << this->GetNbinsZ();
  cout << "\tNum entries = " << this->GetTotalNumEntries()  << "\n";
}

iArray::~iArray() {
}

double* iArray::GetXAxisValues( string type ) {
  //Returns bottom (MIN), top (MAX) or middle (default) of bin values
  double* XAxis = new double[xBins];
  if (type=="MIN")      XAxis[0] = xMin;
  else if (type=="MAX") XAxis[0] = xMin+xDelta;
  else                  XAxis[0] = xMin+(xDelta)/2;
  for (int x=1;x<xBins;x++) {
    XAxis[x]=XAxis[x-1]+xDelta;
  }
  return XAxis;
}

double* iArray::GetYAxisValues( string type ) {
  //Returns bottom (MIN), top (MAX) or middle (default) of bin values
  double* YAxis = new double[yBins];
  if (type=="MIN")      YAxis[0] = yMin;
  else if (type=="MAX") YAxis[0] = yMin+yDelta;
  else                  YAxis[0] = yMin+(yDelta)/2;
  for (int y=1;y<yBins;y++) {
    YAxis[y]=YAxis[y-1]+yDelta;
  }
  return YAxis;
}

double* iArray::GetZAxisValues( string type ) {
  //Returns bottom (MIN), top (MAX) or middle (default) of bin values
  double* ZAxis = new double[zBins];
  if (type=="MIN")      ZAxis[0] = zMin;
  else if (type=="MAX") ZAxis[0] = zMin+zDelta;
  else                  ZAxis[0] = zMin+(zDelta)/2;
  for (int z=1;z<zBins;z++) {
    ZAxis[z]=ZAxis[z-1]+zDelta;
  }
  return ZAxis;
}

double* iArray::GetXYPlaneWBin( int zIn ) {
  // Returns the XY plane array for a Z.
  double* xyPlane = new double[xBins*yBins];
  for (int x=0;x<xBins;x++) {
    for (int y=0;y<yBins;y++) {
```

```
      xyPlane[(x*yBins) + y] = this->GetXYZBin(x,y,zIn);
    }
  }
  return xyPlane;
}

profile* iArray::GetNormZProfileForXY( double xIn, double yIn, string type )
{
  // Method returns an normalised array of Z-values which make up a new
  // (X,Y) profile. Involes bi-cubic spline interpolation at each Z vertex
  // down the line of (X,Y) in the 3D array.

  vector<double> zNew;
  double* xAxis = this->GetXAxisValues(type);
  double* yAxis = this->GetYAxisValues(type);

  for (int z=0;z<zBins;z++) {
    double* xyPlane = this->GetXYPlaneWBin(z);
    double* splines = numRep::splie2(xAxis,yAxis,xyPlane,xBins,yBins);
    zNew.insert(zNew.end(),
    numRep::splin2(xAxis,yAxis,xyPlane,splines,xBins,yBins,xIn,yIn));
    delete [] xyPlane;
  }

  profile *prof = new profile(xIn,yIn,zNew,this->GetZAxisValues(),zBins);
  prof->Normalise();
  return prof;
}

double iArray::GetXYZBin(int x , int y , int z) {
  return array[(zBins*yBins*x) + (y*zBins) + z];
}
double iArray::GetXYZBin(float x , float y , float z) {
  return GetXYZBin((int) x , (int) y , (int) z);
}
double iArray::GetXYZBin(double x , double y , double z) {
  return GetXYZBin((int) x , (int) y , (int) z);
}

void iArray::Dump() {
  for (int i=0;i<xBins*yBins*zBins;i++) {
    cout << "iArray> array[" << i << "]:\t" << array[i] << endl;
  }
}
```

B.3.8 profile.hh

This class was written to contain the data that describes a mass distribution: the data in each bin, and the label of the bin. It also contains the second derivatives that the cubic spline interpolation needs to interpolate (this is to stop the program having to recompute them after the object is created).

```
#ifndef profile_HH
```

```
#define profile_HH

#include "realData.hh"
#include "TH1D.h"

using namespace std;

class profile {
 private:
  double* data;
  double* zAxis;
  double* splines;
  int zBins;
  double zMin;
  double zMax;
  double zDelta;
  double xParam;
  double yParam;
  double mean;
 public:
  profile();
  profile( double , double , double* , double* , int );
  profile( double , double , vector<double> , double* , int );
  ~profile();
  inline double getBinValue ( int bin ) { return zAxis[bin]; }
  inline double getBinElement ( int bin ) { return data[bin]; }
  inline double X() { return xParam; }
  inline double Y() { return yParam; }
  inline double Mean() { return mean; }
  inline double Min() { return zMin; }
  inline double Max() { return zMax; }
  inline double Size() { return zBins; }
  int getBin( double );
  double getDataElement( double );
  void Normalise();
  void Dump();
  inline double LogLikelihoodFit( realData* data , string fitType ) {
     return this->LogLikelihoodFit(data,this->Min(),this->Max(),fitType);
  }
  double LogLikelihoodFit( realData* , double fitMin , double fitMax ,
  string fitType );
  double* Fit( TH1D* dataIn , double norm , double fitMin , double fitMax ,
  string fitType );
  TH1D* profile::getHistogram( string name , string title );
};
#endif
```

B.3.9 profile.cc

```
#include <iostream>
#include <string>
#include <vector>
#include <stdlib.h>
```

```
#include <cstdlib>

#include "numRep.hh"
#include "TMath.h"
#include "fit.hh"
#include "realData.hh"

#include "profile.hh"

profile::profile() {
}

profile::profile( double xIn , double yIn , double* profileIn ,
double* profileAxis , int bins ) {
  xParam = xIn;
  yParam = yIn;
  zDelta = profileAxis[1]-profileAxis[0];
  zBins = bins;
  data = new double[bins];
  zAxis = new double[bins];
  for (int z=0;z<bins;z++) {
    data[z]=profileIn[z];
    zAxis[z]=profileAxis[z];
  }
  zMin = zAxis[0]-(zDelta/2);
  zMax = zAxis[zBins-1]+(zDelta/2);
  splines = numRep::spline(zAxis,data,zBins,1.0e30,1.0e30);
  mean = numRep::splint(zAxis,data,splines,zBins,xParam);
}

profile::profile( double xIn , double yIn , vector<double> profileIn ,
double* profileAxis , int bins ) {
  xParam = xIn;
  yParam = yIn;
  zDelta = profileAxis[1]-profileAxis[0];
  zBins = bins;
  data = new double[bins];
  zAxis = new double[bins];
  for (int z=0;z<bins;z++) {
    data[z]=profileIn[z];
    zAxis[z]=profileAxis[z];
  }
  zMin = zAxis[0]-(zDelta/2);
  zMax = zAxis[zBins-1]+(zDelta/2);
  splines = numRep::spline(zAxis,data,zBins,1.0e30,1.0e30);
  mean = numRep::splint(zAxis,data,splines,zBins,xParam);
}

profile::~profile() {
}

int profile::getBin( double zIn ) {
  if (zIn<zMin || zIn>zMax){
    cout << "profile [" << xParam << "," << yParam << endl;
```

```
    cout << "]> Error, input out of range" << endl;
    return 0;
  } else {
    return (int) TMath::Floor((zIn-zMin)/zDelta);
  }
}

double profile::getDataElement( double zIn ) {
  if (zIn<zMin || zIn>zMax){
    cout << "profile [" << xParam << "," << yParam << endl;
    cout << "]> Error, input out of range" << endl;
    return 0;
  } else {
    return numRep::splint(zAxis,data,splines,zBins,zIn);
  }
}

void profile::Normalise() {
  double N=0;
  for (int i=0;i<zBins;i++) {
    N += numRep::splint(zAxis,data,splines,zBins,zAxis[i]) * zDelta;
  }
  for (int j=0;j<zBins;j++) {
    data[j] = data[j]/N;
  }
  delete [] splines;
  double* splines = numRep::spline(zAxis,data,zBins,1.0e30,1.0e30);
  mean = numRep::splint(zAxis,data,splines,zBins,xParam);
}

void profile::Dump() {
  for (int i=0;i<zBins;i++){
    cout << "profile [" << xParam << "," << yParam << "]>\t";
    cout << "zAxis[" << i  << "]\t" << zAxis[i];
    cout << "\tdata[" << i  << "]\t" << data[i];
    cout << "\tspline[" << i  << "]\t" << splines[i] << endl;
  }
}

double profile::LogLikelihoodFit( realData* dataIn , double fitMin ,
double fitMax , string fitType ) {
  return fit::getFitParameter(dataIn,this,fitType,fitMin,fitMax);
}

double* profile::Fit( TH1D* dataIn , double norm , double fitMin ,
double fitMax , string fitType ) {
  return fit::getChiSq(dataIn,norm,this,fitType,fitMin,fitMax);
}

TH1D* profile::getHistogram( string name , string title ) {
  TH1D* h = new TH1D(name.c_str(),title.c_str(),zBins,zMin,zMax);
  for (int i=1;i<=zBins;i++) {
    h->SetBinContent(i,data[i-1]);
  }
```

```
  return h;
}
```

B.3.10 fit.hh

This class contains the fitting methods used to compare an input `profile` class with a input `TH1D` Root histogram of real data.

```
#ifndef fit_HH
#define fit_HH

#include "realData.hh"
#include "profile.hh"

using namespace std;

class fit {

 public:
  fit();
  ~fit();
  static double  getFitParameter(realData* , profile* , string ,  double ,
  double );
  static double* getChiSq(TH1D* data , double norm , profile*  profileIn ,
  string fitType , double fitMin , double fitMax );
  static double  gammln( double );
};
#endif
```

B.3.11 fit.cc

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <vector>
#include <stdlib.h>

#include "numRep.hh"
#include "TMath.h"

#include "fit.hh"

fit::fit() {
}

fit::~fit() {
}

double fit::getFitParameter(realData* data , profile *profileIn ,
string fitType , double fitMin , double fitMax ) {
  double fitSum;
  double sum;
```

```
  double mean = profileIn->Mean();

  for (int i=0;i<data->numItems();i++) {
    double realMZ = data->getDataElement(i);
    if( (realMZ>fitMin) && (realMZ<fitMax) ) {
      const double databit = profileIn->getDataElement(realMZ);
      if (fitType=="LOG") {
sum = (double) TMath::Log(databit);
      }
      if (fitType=="CHI") {
sum = (double) (databit-mean)/mean ;
      }
      fitSum = fitSum + sum;
    }
  }
  return fitSum;
}

double* fit::getChiSq(TH1D* data , double norm , profile *profileIn ,
string fitType , double fitMin , double fitMax ) {
  TH1D* theory = profileIn->getHistogram("bfit","Estimate");
  theory->Scale(norm);
  double* chi = new double[3];
  double sum,th,ex;
  int min,max;
  // calculate which bins to check over.
  min = profileIn->getBin(fitMin);
  max = profileIn->getBin(fitMax);
  chi[0] = chi[1] = chi[2] = 0;
  // Chi^2 fit over the bins
  for (int i=min;i<=max;i++) {
    th = theory->GetBinContent(i+1);
    ex = data->GetBinContent(i+1);
    if (ex!=0.0) {
      chi[0]+= (TMath::Power(ex-th,2))/ex;
      chi[1]++;
    }
  }
  theory->~TH1D();
  chi[1]--;
  chi[2] = TMath::Prob(chi[0],chi[1]);
  return chi;
}
```

B.3.12 numRep.hh

This class contains algorithms taken from '*Numerical Recipes in C*' and '*Numerical Recipes in C++*' [6].

```
#ifndef numRep_HH
#define numRep_HH

using namespace std;
```

```
class numRep {
 public:
  static double* spline(double*,double*,int,double,double);
  static double  splint(double*,double*,double*,int,double);
  static double* splie2(double*,double*,double*,int,int);
  static double  splin2(double*,double*,double*,double*,int,int,
  double,double);
  static double* extractYProfile(double*,int,int);
  static double* numRep::polint(double* xa,double * ya,int n,double x);
};

#endif
```

B.3.13 numRep.cc

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <vector>
#include <stdlib.h>
#include <math.h>

#include "TMath.h"
#include "numRep.hh"

double* numRep::extractYProfile( double* xyPlane , int yBins , int xIn ) {
  double *profile = new double[yBins];
  for (int y=0;y<yBins;y++) {
    int ik = (xIn*yBins)+y;
    profile[y] = xyPlane[ik];
  }
  return profile;
}

double* numRep::spline( double* x , double* y , int n , double yp1 ,
double ypn ) {

  // Given y[] where y[i] = y( x[i] ), the size of the array, n and the
  // first and last derivatives of y, will return the array needed for
  // cubic spline interpolation.

  // Calling with yp1 or ypn > = 0.99e30, will make it use a natural spline
  // there.

  int i,k;
  double p,qn,sig,un;
  double *u  = new double[n-1];
  double *y2 = new double[n];
  if (yp1 > 0.99e30 ) {
    y2[0] = u[0] = 0.0;
  } else {
```

```
    y2[0] = -0.5;
    u[0]  = (3/(x[1]-x[0]))*((y[1]*y[0])/(x[1]-x[0])-yp1);
  }
  for (i=1;i<n-1;i++) {
      sig = (x[i]-x[i-1])/(x[i+1]-x[i-1]);
        p = sig * y2[i-1] + 2;
    y2[i] = (sig-1)/p;
     u[i] = (y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i-1]);
     u[i] = (6 * u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
  }
  if (ypn > 0.99e30 ) {
    qn=un=0;
  } else {
    qn = 0.5;
    un = (3/(x[n-1]-x[n-2]))*(ypn-(y[n-1]-y[n-1])/(x[n-1]-x[n-1]));
  }
  y2[n-1] = (un-qn*u[n-2])/(qn*y2[n-1]+1);
  for (k=n-2;k>=0;k--) {
    y2[k]=y2[k]*y2[k+1]+u[k];
  }
  return y2;
}

double numRep::splint( double* xa , double* ya , double* y2a , int n ,
double x ) {
  // With spline run already, to calculate y2*, this will return a  y(x) for
  // given x

  int klo,khi,k;
  double h,b,a;

  klo = 0;
  khi = n-1;

  while (khi-klo > 1) {
    k = (khi+klo) >> 1;
    if (xa[k] > x) khi=k;
    else klo=k;
  }
  h=xa[khi]-xa[klo];
  if (h==0) cout << "numRep> Splint says something is wrong, bad input.\n";
  a=(xa[khi]-x)/h;
  b=(x-xa[klo])/h;
  double thing =
      a*ya[klo]+b*ya[khi]+((a*a*a-a)*y2a[klo]+(b*b*b-b)*y2a[khi])*(h*h)/6.0;
  return thing;
}

double* numRep::splie2( double* xAxis , double* yAxis , double*
xyPlane , int xBins , int yBins ) {
  // This is called before bicubic spline routine.  Like spline, it
  // generates the second derivatives needed.

  double *splines = new double[xBins*yBins];
```

```
  for (int x=0;x<xBins;x++) {

    double *temp = numRep::extractYProfile(xyPlane,yBins,x);

    // create temp array with new splines

    double *temp2;
    temp2 = spline(yAxis,temp,yBins,1.0e30,1.0e30);

    for (int y=0;y<yBins;y++) {
      splines[(x*yBins)+y] = temp2[y];
    }

    delete [] temp;
    delete [] temp2;
  }
  return splines;
}


double numRep::splin2( double* xAxis , double* yAxis , double* xyPlane,
double* splines , int xBins , int yBins , double xIn , double yIn ) {
  // With splie2 called, this generates a bicubic spline interpolated z
  // value for (x,y)
  double *yTemp = new double[xBins];
  double *yyTemp = new double[xBins];

  for (int x=0;x<xBins;x++) {
    double *temp_prof = numRep::extractYProfile(xyPlane,yBins,x);
    double *temp_spli = numRep::extractYProfile(splines,yBins,x);
    yyTemp[x] = splint(yAxis,temp_prof,temp_spli,yBins,yIn);
    delete [] temp_prof;
    delete [] temp_spli;
  }

  yTemp = spline(xAxis,yyTemp,xBins,1.0e30,1.0e30);

  return splint(xAxis,yyTemp,yTemp,xBins,xIn);

}

double* numRep::polint( double* xa , double * ya , int n , double x )
  // For this routine y[0] is the answer, y[1] is the error (y is returned).
{
  int i,m,ns=1;
  double den,dif,dift,ho,hp,w;
  double *c,*d;
  double *y=new double[2];

  dif=fabs(x-xa[0]);
  c = new double[n-1];
  d = new double[n-1];
  for (i=0;i<n;i++) {
```

```
    if ( (dift=fabs(x-xa[i])) < dif) {
      ns=i;
      dif=dift;
    }
    c[i]=ya[i];
    d[i]=ya[i];
  }
  y[0]=ya[ns--];
  for (m=1;m<n;m++) {
    for (i=0;i<n-m;i++) {
      ho=xa[i]-x;
      hp=xa[i+m]-x;
      w=c[i+1]-d[i];
      if ( (den=ho-hp) == 0.0) {
         cout << "numRep> Error in polint: ho=" << endl;
         cout << ho << ", hp=" << hp << endl;
      }
      den=w/den;
      d[i]=hp*den;
      c[i]=ho*den;
    }
    y[0] += (y[1]=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
  }
  return y;
}
```

B.3.14 fitter.cards

The program is controlled using this file. To avoid having to recompile the program every time a parameter is changed, this file stores the constants and parameters that a user would wish to change. An environment variable must be set telling the fitting program where to find this file. This class uses the `ControlCards` class, which was written by Dr. Mark Lancaster.

```
# ---------------------- FITTER -------------------------------
#
# Define using   'export CONTROL_CARDS=fitter.cards'
#
# The Makefile is edited so that it uses main_minuit.cc as the defult
# program main. To change this, just edit the Makefile in the correct
# place so that the link points to the program main you wish to uses.
#
# ============== Output ================================
#
# Set to '1' to save the best fit histogram as TH1D inside OUTPUT_FILE.
#
#
OUTPUT_ROOT 1
OUTPUT_FILE 040321_Fitter80.root
HISTO_NAME Mid
X_AXIS M_{Z}(GeV)
Y_AXIS #kappa
```

```
Z_AXIS M_{Z}(GeV)
#
# ============== Files ==================================
#
# --- Location of experimental data ---
#
#EXP_DATA_FILE /home/dan/4C00/4C00_fitter/zee_central_only_cdf_real.data
#EXP_DATA_FILE /home/beecher/4C00_fitter/zee_central_only_cdf_real.data
#EXP_DATA_FILE /home/markl/Z_DATA.dat
#EXP_DATA_FILE /home/beecher/4C00_fitterN/Z_DATA.dat
EXP_DATA_FILE /home/dan/4C00/4C00_fitterN/Z_DATA.dat
#
# --- Location of the stored histograms
#
#HISTO_FILE /home/dan/4C00/4C00_fitter/ze_mz_kappa.root
#HISTO_FILE /home/beecher/4C00_fitter/ze_mz_kappa.root
#HISTO_FILE /home/markl/Z_TEMPLATE.root
#HISTO_FILE /home/beecher/4C00_fitterN/Z_TEMPLATE.root
HISTO_FILE /home/dan/4C00/4C00_fitterN/Z_TEMPLATE.root
MASTER_ID h99999
#
# ============== Parameters to fit ======================
#
# Minuit seems to be very fussy about the starting values; there may only
# be a small minimun local to the best fit.  If Minuit tries to minimise
# outside of this region, then it will return the limits as the best fit.
# Thus, it is useful to have an idea of what the best fit parameters
# should be near.
#
# --- General params
#
# Bin min (MIN), max bin (MAX) or centre (default).
BIN_TYPE m
#
# --- First minimisation parameter
#
X_PARAM_NAME Mz
X_PARAM_START 91.178
X_PARAM_MIN 90
X_PARAM_MAX 92
X_PARAM_STEP 0.005
#
# --- Second minimisation parameter
#
Y_PARAM_NAME Kappa
Y_PARAM_START 1.7
Y_PARAM_MIN 0.31
Y_PARAM_MAX 2.29
Y_PARAM_STEP 0.005
#
# --- Range of fitting while calulating best fit
#
# Set DEFAULT_RANGE to 1 to use the whole range of the input histograms
# while calculating the best fit.  Set it to 0 to use a range specified
```

```
# by FIT_MIN_X and FIT_MAX_X
#
DEFAULT_RANGE 0
FIT_MIN_X 80
FIT_MAX_X 100
#
# ============= Minuit fitting parameters ===============
#
# --- Error calculation to use
#
# 0.5 Log likelihood
# 1   Chi squared
M_ERROR 1
#
# --- Maximum iterations
#
MAX_ITER 500
#
# --- Minimisation strategy
#
# 0 = Minimise the number of calls to function
# 1 = Try to balance speed against reliablity (standard)
# 2 = Make sure minimum true, errors correct
MIN_TYPE 1
#
# --- Feedback
#
#      -1 = No output
# 0 = No warnings, result only
# 1 = Normal
M_FEED 0
#
#
```

B.4 Transverse momentum fit

This program consisted of one main executable class, and a external file containing fitting parameters.

B.4.1 ptfit.cc

This program invokes `TMinuit` to minimise the χ^2 fit of the smeared functional form to the actual data. HERWIG data is read into the file, used to create the re-weighting array. This is then used to 'smear' a `TH1D` histogram containing the functional form.

The commands at the end of the code create a nice histogram and save it in a Root file of the user's designation.

```
#include <iostream>
#include <fstream>
#include <string>
```

```
#include <cstdlib>
#include <vector>
#include <stdlib.h>
#include <sstream>

#include "TMath.h"
#include "TFile.h"
#include "TH1D.h"
#include "realData.hh"
#include "TMinuit.h"
#include "TCanvas.h"
#include "TLegend.h"
#include "TRandom.h"
#include "TLorentzVector.h"
#include "TPaveText.h"
#include "ControlCards.hh"

using namespace std;

static TH1D* exptdata;
static vector<double> reweight;
static int   xBins;
static double xMax,xMin,xDelta;

double gammaln(double xx)
{
  double x,y,tmp,ser;
  static double cof[6]={
    76.18009172947146,
    -86.50532032941677,
    24.01409824083091,
    -1.231739572450155,
    0.1208650973866179e-2,
    -0.5395239384953e-5};
  int j;
  y=x=xx;
  tmp=x+5.5;
  tmp -= (x+0.5)*TMath::Log(tmp);
  ser=1.000000000190015;
  for (j=0;j<5;j++){
    ser += cof[j]/++y;
  }
  return -tmp+(TMath::Log(2.5066282746310005*ser/x));
}

double PTtrue(double P1 , double P2 , double P3 , double P4 , double PT )
  // The unsmeared 4-param PT functional form
{
  double X = PT/50.0;
  double gammaP4 = P4 * TMath::Exp(gammaln(P4));
  double part1 = TMath::Power(X,P4)/gammaP4;
  double part2 = (1-P1) * (TMath::Power(P2,P4+1)) * (TMath::Exp(-P2*X));
  double part3 = P1 * (TMath::Power(P3,P4+1)) * (TMath::Exp(-P3*X));
  return part1*(part2+part3);
```

```
}

double* chisq(TH1D* data, vector<double> reweight, int xBins, double P1,
double P2, double P3, double P4)
  // Loops over PT data in data object and returns a chi^2 for the current
  // PTtrue
{
  double *chi= new double[5];
  chi[0] = chi[1]  = 0;
  chi[2] = xBins;
  double* chisum = new double[2];
  TH1D* tested =
  new TH1D("buffer","Histogram being used to test",xBins,xMin,xMax);
  double norm=0;
  double nono=0;
  for(int i=1;i<=xBins;i++) {
    double pt = data->GetXaxis()->GetBinCenter(i);
    double w = PTtrue(P1,P2,P3,P4,pt);
    tested->Fill(pt,w);
    norm+=(w*xDelta);
    nono+=(data->GetBinContent(i))*xDelta;
  }
  tested->Scale(nono/norm);
  for(int i=1;i<=xBins;i++) {
    double expect = data->GetBinContent(i);
    double theory = tested->GetBinContent(i);
    double theory_sm = theory*reweight[i];
    if (expect!=0) {
      chisum[0] = TMath::Power((theory_sm-expect),2)/expect;
      chisum[1] = TMath::Power((theory-expect),2)/expect;
    } else {
      chisum[0] = chisum[1] = 0;
      chi[2]--;
    }
    chi[0]+=chisum[0];
    chi[1]+=chisum[1];
  }
  tested->~TH1D();
  chi[3] = TMath::Prob(chi[0],chi[2]);
  chi[4] = TMath::Prob(chi[1],chi[2]);
  return chi;
}

static void myFCN( Int_t &npar, Double_t* gin, Double_t &f, Double_t* par,
Int_t flag )
  // Function minimised by Minuit.
{
  f=0;
  double P1 = par[0];
  double P2 = par[1];
  double P3 = par[2];
  double P4 = par[3];
  double *chi = chisq(exptdata,reweight,xBins,P1,P2,P3,P4);
  f = chi[0];
```

```
}

int main()
{
  // Define the variables to be read from the cards
  ControlCards cCards;
  cCards.define("REAL_DATA","");
  cCards.define("HERWIG_DATA","");
  cCards.define("OUTPUT_FILE","");
  cCards.define("X_MIN",(float) 0);
  cCards.define("X_MAX",(float) 50);
  cCards.define("X_BINS",(int) 50);
  cCards.define("E_MASS",(float) 0.511e-3);
  cCards.define("KAPPA",(float) 1.8);
  cCards.define("P1_START",(float) 1);
  cCards.define("P2_START",(float) 1);
  cCards.define("P3_START",(float) 1);
  cCards.define("P4_START",(float) 1);
  cCards.define("P1_MIN",(float) 0);
  cCards.define("P2_MIN",(float) 0);
  cCards.define("P3_MIN",(float) 0);
  cCards.define("P4_MIN",(float) 0);
  cCards.define("P1_MAX",(float) 0);
  cCards.define("P2_MAX",(float) 0);
  cCards.define("P3_MAX",(float) 0);
  cCards.define("P4_MAX",(float) 0);
  cCards.define("P1_STEP",(float) 0);
  cCards.define("P2_STEP",(float) 0);
  cCards.define("P3_STEP",(float) 0);
  cCards.define("P4_STEP",(float) 0);
  cCards.define("MAX_ITR",(int) 500);
  cCards.define("STRENGTH",(int) 1);
  cCards.define("FEEDBACK",(int) 0);
  cCards.Read();
  cCards.Print();

  // Constants to be determined by CC.

  xMin   = (double) cCards.FloatValue("X_MIN");
  xMax   = (double) cCards.FloatValue("X_MAX");
  xBins  = cCards.IntValue("X_BINS");
  xDelta = (xMax-xMin)/xBins;

  const double E_MASS = (double) cCards.FloatValue("E_MASS");
  const double KAPPA  = (double) cCards.FloatValue("KAPPA");

  // Create the reweighting vector

  vector<double> weight;
  vector<double> px1;
  vector<double> py1;
  vector<double> pz1;
  vector<double> px2;
  vector<double> py2;
```

```
  vector<double> pz2;

  ifstream ff((cCards.StringValue("HERWIG_DATA")).c_str());
  if (!ff.is_open()) {
    cout << cCards.StringValue("HERWIG_DATA") << " not found...." << endl;
    return -1;
  }
  cout << "=== File opened ===" << endl;
  string buffer;
  char *stopchar;
  while (!ff.eof()) {
    ff >> buffer; // Event weight
    weight.insert(weight.end(),strtod(buffer.c_str(),&stopchar));
    ff >> buffer; // PX1
    px1.insert(px1.end(),strtod(buffer.c_str(),&stopchar));
    ff >> buffer; // PY1
    py1.insert(py1.end(),strtod(buffer.c_str(),&stopchar));
    ff >> buffer; // PZ1
    pz1.insert(pz1.end(),strtod(buffer.c_str(),&stopchar));
    ff >> buffer; // PX2
    px2.insert(px2.end(),strtod(buffer.c_str(),&stopchar));
    ff >> buffer; // PY2
    py2.insert(py2.end(),strtod(buffer.c_str(),&stopchar));
    ff >> buffer; // PZ2
    pz2.insert(pz2.end(),strtod(buffer.c_str(),&stopchar));
  }
  ff.close();
  cout << "=== File read ===" << endl;
  TH1D* DT1 = new TH1D("data_us","HERWIG Data (unsmeared)",xBins,xMin,xMax);
  TH1D* DT2 = new TH1D("data_sm","HERWIG Data (smeared)",xBins,xMin,xMax);

  TLorentzVector e1,e2,z0,z0s;
  for (int i=0;i<px1.size();i++) {
    e1.SetPxPyPzE(px1[i],
  py1[i],
  pz1[i],
TMath::Sqrt(px1[i]*px1[i] + py1[i]*py1[i] + pz1[i]*pz1[i])+E_MASS);
    e2.SetPxPyPzE(px2[i],
  py2[i],
  pz2[i],
TMath::Sqrt(px2[i]*px2[i] + py2[i]*py2[i] + pz2[i]*pz2[i])+E_MASS);

    // Determine the eta for e+/e-

    double eta1 = e1.PseudoRapidity();
    double eta2 = e2.PseudoRapidity();

    // If both are detected by the CEM -> |eta| <= 1

    if( TMath::Abs(eta1)<=1.0 && TMath::Abs(eta2)<=1.0 ) {
      // Smear the energy
      double *r = new double[8];
      for (int i=0;i<8;i++) {
r[i] = gRandom->Gaus();
```

```
    }

    double p1  = e1.P();
    double p2  = e2.P();
    double pt1 = e1.Pt();
    double pt2 = e2.Pt();
    double en1 = e1.E();
    double en2 = e2.E();

    double deltaE1 = TMath::Sqrt((0.135*0.135*p1/(en1*pt1))+KAPPA*KAPPA);
    double deltaE2 = TMath::Sqrt((0.135*0.135*p2/(en2*pt2))+KAPPA*KAPPA);

    // Smear the energy;
    double en1s = en1 * (1 + r[0]*deltaE1);
    double en2s = en2 * (1 + r[1]*deltaE2);

    // Smear the momenta
    double px1s = px1[i] * (1 + r[2]*(en1*deltaE1/p1));
    double py1s = py1[i] * (1 + r[3]*(en1*deltaE1/p1));
    double pz1s = pz1[i] * (1 + r[4]*(en1*deltaE1/p1));
    double px2s = px2[i] * (1 + r[5]*(en2*deltaE2/p2));
    double py2s = py2[i] * (1 + r[6]*(en2*deltaE2/p2));
    double pz2s = pz2[i] * (1 + r[7]*(en2*deltaE2/p2));

    // Smeared particles
    TLorentzVector e1s;
    e1s.SetPxPyPzE(px1s,py1s,pz1s,en1s);
    TLorentzVector e2s;
    e2s.SetPxPyPzE(px2s,py2s,pz2s,en2s);

    // Reconstruct the Z0's
    z0  = e1 + e2;
    z0s = e1s + e2s;

    // Fill smeared and unsmeared histograms

    DT1->Fill(z0.Pt(),weight[i]);
    DT2->Fill(z0s.Pt(),weight[i]);
  }
}

double norma=0;
double normb=0;
for (int i=1;i<=xBins;i++) {
  norma+= (DT1->GetBinContent(i))*xDelta;
  normb+= (DT2->GetBinContent(i))*xDelta;
}
DT1->Scale(1/norma);
DT2->Scale(1/normb);

// Create the reweighting array/vector
for (int i=1;i<=xBins;i++) {
  const double sme = DT2->GetBinContent(i);
  const double usm = DT1->GetBinContent(i);
```

```
  const double wei = sme/usm;
  reweight.insert(reweight.end(),wei);
}

// Create histogram of real data

vector<double> ac_px1;
vector<double> ac_py1;
vector<double> ac_pz1;
vector<double> ac_px2;
vector<double> ac_py2;
vector<double> ac_pz2;

ifstream f2((cCards.StringValue("REAL_DATA")).c_str());
while (!f2.eof()) {
  f2 >> buffer; // PX1
  ac_px1.insert(ac_px1.end(),strtod(buffer.c_str(),&stopchar));
  f2 >> buffer; // PY1
  ac_py1.insert(ac_py1.end(),strtod(buffer.c_str(),&stopchar));
  f2 >> buffer; // PZ1
  ac_pz1.insert(ac_pz1.end(),strtod(buffer.c_str(),&stopchar));
  f2 >> buffer; // PX2
  ac_px2.insert(ac_px2.end(),strtod(buffer.c_str(),&stopchar));
  f2 >> buffer; // PY2
  ac_py2.insert(ac_py2.end(),strtod(buffer.c_str(),&stopchar));
  f2 >> buffer; // PZ2
  ac_pz2.insert(ac_pz2.end(),strtod(buffer.c_str(),&stopchar));
}
f2.close();

exptdata =
new TH1D("expt","TMinuit fit of CDF P_{T}(Z) data",xBins,xMin,xMax);

for (int i=0;i<ac_px1.size();i++) {
  e1.SetPxPyPzE(ac_px1[i],
ac_py1[i],
ac_pz1[i],
TMath::Sqrt(ac_px1[i]*ac_px1[i] + ac_py1[i]*ac_py1[i] +
                ac_pz1[i]*ac_pz1[i])+E_MASS);
  e2.SetPxPyPzE(ac_px2[i],
ac_py2[i],
ac_pz2[i],
TMath::Sqrt(ac_px2[i]*ac_px2[i] + ac_py2[i]*ac_py2[i] +
                ac_pz2[i]*ac_pz2[i])+E_MASS);
  z0 = e1 + e2;
  exptdata->Fill(z0.Pt());
}

double norm1=0;
for (int i=1;i<=xBins;i++) {
  norm1+=(exptdata->GetBinContent(i))*xDelta;
}

// Define Minuit
```

```
Int_t pm = 4;
TMinuit *minuit = new TMinuit(pm);
minuit->SetFCN(myFCN);
int ierflg = 0;

// Minimisation Parameters
string* name = new string[4];
name[0] = "P1";
name[1] = "P2";
name[2] = "P3";
name[3] = "P4";
double* start_values = new double[4];
start_values[0] = (double) cCards.FloatValue("P1_START");
start_values[1] = (double) cCards.FloatValue("P2_START");
start_values[2] = (double) cCards.FloatValue("P3_START");
start_values[3] = (double) cCards.FloatValue("P4_START");
double* lim_min = new double[4];
lim_min[0] = (double) cCards.FloatValue("P1_MIN");
lim_min[1] = (double) cCards.FloatValue("P2_MIN");
lim_min[2] = (double) cCards.FloatValue("P3_MIN");
lim_min[3] = (double) cCards.FloatValue("P4_MIN");
double* lim_max = new double[4];
// set to the same as lim_min to run with no limits
lim_max[0] = (double) cCards.FloatValue("P1_MAX");
lim_max[1] = (double) cCards.FloatValue("P2_MAX");
lim_max[2] = (double) cCards.FloatValue("P3_MAX");
lim_max[3] = (double) cCards.FloatValue("P4_MAX");
double* step = new double[4];
step[0] = (double) cCards.FloatValue("P1_STEP");
step[1] = (double) cCards.FloatValue("P2_STEP");
step[2] = (double) cCards.FloatValue("P3_STEP");
step[3] = (double) cCards.FloatValue("P4_STEP");
Double_t* val = new double[1];

// Set parameters
minuit->SetPrintLevel(cCards.IntValue("FEEDBACK"));

for (int pms=0;pms<pm;pms++) {
  minuit->mnparm(pms,name[pms].c_str(),start_values[pms],step[pms],
  lim_min[pms],lim_max[pms],ierflg);
}

minuit->SetErrorDef(1);
minuit->SetMaxIterations(cCards.IntValue("MAX_ITR"));
val[0] = cCards.IntValue("STRENGTH"); // Strength
minuit->mnexcm("SET STR",val,1,ierflg);
minuit->Migrad();

double* result = new double[4];
double* error  = new double[4];

for (int pms=0;pms<pm;pms++) {
   minuit->GetParameter(pms,result[pms],error[pms]);
   cout << "PTFit> " << name[pms] << "\t" << result[pms] << endl;
```

```
   cout << "\tError=" << error[pms] << endl;
}

// Error checking

TH1D *PT1u   = new TH1D("PT1u","PT1u",xBins,xMin,xMax);
TH1D *PT2u   = new TH1D("PT2u","PT2u",xBins,xMin,xMax);
TH1D *PT3u   = new TH1D("PT3u","PT3u",xBins,xMin,xMax);
TH1D *PT4u   = new TH1D("PT4u","PT4u",xBins,xMin,xMax);
TH1D *PT1d   = new TH1D("PT1d","PT1d",xBins,xMin,xMax);
TH1D *PT2d   = new TH1D("PT2d","PT2d",xBins,xMin,xMax);
TH1D *PT3d   = new TH1D("PT3d","PT3d",xBins,xMin,xMax);
TH1D *PT4d   = new TH1D("PT4d","PT4d",xBins,xMin,xMax);


// Derive the chi^2 for the final histogram.
double *chi =
chisq(exptdata,reweight,xBins,result[0],result[1],result[2],result[3]);

stringstream ss1;
ss1 << "#chi^{2} = " << chi[0] << " / " << chi[2] << " (sm.)";
string chi_string1 = ss1.str();

stringstream ss2;
ss2 << "#chi^{2} = " << chi[1] << " / " << chi[2] << " (unsm.)";
string chi_string2 = ss2.str();

cout << " ==== SMEARED ====" << endl;
cout << "chi^2 = " << chi[0] << "\tDoF = " << chi[2] << "\tp = " << endl;
cout << chi[3] << endl;
cout << " === UNSMEARED ===" << endl;
cout << "chi^2 = " << chi[1] << "\tDoF = " << chi[2] << "\tp = " << endl;
cout <<  chi[4] << endl;


TH1D *PT1   = new TH1D("PTf_us","Unsmeared P_{T} form",xBins,xMin,xMax);
TH1D *PT2   = new TH1D("PTf_sm","Smeared P_{T} form",xBins,xMin,xMax);
TH1D *nu    = new TH1D("","",xBins,xMin,xMax);
nu->SetLineColor(10);

reweight.insert(reweight.end(),1);

for (double i=(xMin+xDelta/2);i<=xMax;i+=xDelta) {
  PT1->Fill(i,PTtrue(result[0],result[1],result[2],result[3],i));
  PT2->Fill(
  i,PTtrue(result[0],result[1],result[2],result[3],i)*reweight[i]);
  PT1u->Fill(
  i,PTtrue(result[0]+error[0],result[1],result[2],result[3],i));
  PT1d->Fill(
  i,PTtrue(result[0]-error[0],result[1],result[2],result[3],i));
  PT2u->Fill(
  i,PTtrue(result[0],result[1]+error[1],result[2],result[3],i));
  PT2d->Fill(
  i,PTtrue(result[0],result[1]-error[1],result[2],result[3],i));
```

```
  PT3u->Fill(
  i,PTtrue(result[0],result[1],result[2]+error[2],result[3],i));
  PT3d->Fill(
  i,PTtrue(result[0],result[1],result[2]-error[2],result[3],i));
  PT4u->Fill(
  i,PTtrue(result[0],result[1],result[2],result[3]+error[3],i));
  PT4d->Fill(
  i,PTtrue(result[0],result[1],result[2],result[3]-error[3],i));
}

double norm2,norm3,norm4;
norm2=norm3=norm4=0;
for (int i=0;i<xBins;i++) {
  norm2+= (PT1->GetBinContent(i))*xDelta;
  norm3+= (PT2->GetBinContent(i))*xDelta;
  norm4+= (exptdata->GetBinContent(i))*xDelta;
}

PT1->Scale(norm4/norm2);
PT2->Scale(norm4/norm3);

PT2->SetFillColor(0);
PT2->GetXaxis()->SetTitle("P_{T} (GeV)");
exptdata->SetFillColor(0);
exptdata->SetLineColor(9);
exptdata->GetXaxis()->SetTitle("P_{T} (GeV)");
PT2->SetLineColor(kRed);
PT2->SetStats(kFALSE);
exptdata->SetStats(kFALSE);

TCanvas* c1 = new TCanvas("can","Check");
c1->SetFillColor(0);
c1->SetBorderMode(0);
exptdata->Draw("E0");
PT1->Draw("same c");
PT2->Draw("same c");
TLegend *leg = new TLegend(0.6,0.7,1,1);
leg->AddEntry(exptdata,"CDF Data","L");
leg->AddEntry(PT2,"Best fit (sm.)","L");
leg->AddEntry(PT1,"Best fit (unsm.)","L");
leg->AddEntry(nu,chi_string1.c_str(),"L");
leg->AddEntry(nu,chi_string2.c_str(),"L");
leg->SetFillColor(0);
leg->SetTextFont(62);
leg->SetTextSize(0.04);
leg->Draw();

TFile *f1 =
new TFile((cCards.StringValue("OUTPUT_FILE")).c_str(),"RECREATE");
PT1->Write();
PT2->Write();
c1->Write();
exptdata->SetTitle("Experimental Data");
exptdata->Write();
```

```
  PT1u->Write();
  PT1d->Write();
  PT2u->Write();
  PT2d->Write();
  PT3u->Write();
  PT3d->Write();
  PT4u->Write();
  PT4d->Write();
  f1->Close();
}
```

B.4.2 ptcards.cards

The external file that contains the parameters of the fit.

```
# ----------------------- PTFIT --------------------------------
#
# Define using   'export CONTROL_CARDS=ptcard.cards'
#
# ============== Output ================================
#
OUTPUT_FILE 040322_dummy.root
#
# ============== Files ================================
#
# --- Location of experimental data ---
#
REAL_DATA /home/beecher/4C00_pt_form/zee_pxpypz.dat
#
# --- Location of HERWIG data ---
#
#HERWIG_DATA  /home/beecher/4C00_pt_form/ptz_4vec.dat
HERWIG_DATA /home/markl/ptz_4vec.dat
#
#
# ============== Variables ================================
#
# --- Histogram sizes ---
X_BINS 50
X_MIN 0
X_MAX 50
#
# The mass of the electron in GeV
E_MASS 0.511e-3
#
# The kappa factor in the CEM resolution.
KAPPA 0.01742
#
# ============== Parameters to fit ======================
#
# Minuit seems to be very fussy about the starting values; there may only
# be a small minimun local to the best fit.  If Minuit tries to minimise
# outside of this region, then it will return the limits as the best fit.
```

```
# Thus, it is useful to have an idea of what the best fit parameters should
# be near.
#
# To run with no limits, set the min and max to be the same number.
#
# --- P1
#
P1_START 0.1
P1_MIN 0
P1_MAX 0
P1_STEP 0.1
#
# --- P2
#
P2_START 0.1
P2_MIN 0
P2_MAX 0
P2_STEP 0.1
#
# --- P3
#
P3_START 10
P3_MIN 0
P3_MAX 0
P3_STEP 0.1
#
# --- P4
#
P4_START 0.1
P4_MIN 0
P4_MAX 0
P4_STEP 0.1
#
# ============= Minuit fitting parameters ===============
#
# --- Maximum iterations
#
MAX_ITR 500
#
# --- Minimisation strategy
#
# 0 = Minimise the number of calls to function
# 1 = Try to balance speed against reliablity (standard)
# 2 = Make sure minimum true, errors correct
STRENGTH 1
#
# --- Feedback
#
#      -1 = No output
# 0 = No warnings, result only
# 1 = Normal
FEEDBACK 1
```

References

[1] F. Abe *et al.*, *Nucl. Instrum. Methods A*, **276** (1988)

[2] W. Press *et al.*, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge (1988)

[3] M. Lancaster and D. Waters, *J. Phys. G: Nucl. Part. Phys.*, **26**, (2000)

[4] K. Hagiwara *et al. Physical Review D* **66**, 10001-1 (2002)

[5] R. Brun *et al. Root Users Guide 3.05* (2003) http://root.cern.ch

[6] W. Press *et al.*, *Numerical Recipes in C++: The Art of Scientific Computing*, Cambridge (2002)

[7] D. Griffiths, *Introduction to Elementary Particles*, Wiley (1987)

[8] DØ Collaboration, B Abbot *et al.*, *Phys Rev. D*, **61**, 032004 (2000)

[9] CDF Collaboration, T. Affolder *et al.*, *Phys Rev. Lett.* (1999)

[10] E. Gerchtein and M. Paulini, "CDF Detector Simulation Framework and Performance" *Computing in High Energy and Nuclear Physics* (2003)

[11] L. Balka *et al.*, *Nucl.Inst.Phys.A*, **267**, 272–279 (1988)