1. Introduction to Modern Physics

1.1 Background

The Standard Model has evolved over the last 100 years as the theory of fundamental particles and their interactions. Prior to the 19th century, it was believed that all matter was made solely of atoms which were believed to be fundamental, indivisible, elementary particles, having no substructure. Then, in 1897, J.J. Thomson discovered the electron. Initially, very little was known about the electron except that it was *very* small (much smaller than an atom) and negatively charged. The concept of the atom quickly evolved to consist of a positively charged blob of essentially uniform density enveloping negatively charged electrons. The famous analogy was that it was similar to plum pudding with the electrons being the plums surrounded by the positively charged pudding.

Ernest Rutherford was the first to radically alter this initial concept in 1911 when he performed his now-famous gold foil experiment. The experiment consisted of firing positively charged α particles (now known to consist of a pair of protons and a pair of neutrons) at a piece of gold foil and observing the trajectory at which they scattered. According to the "plum pudding" atomic model, Rutherford expected all of the α particles to fly straight through the thin layer of gold foil, plowing through the "pudding" with ease. Most of the particles did indeed pass through the foil essentially unimpeded. However, some of them unexpectedly scattered straight back at nearly 180 degrees. Rutherford later described this result as though "you had fired a 15-inch shell at a piece of tissue paper and it came back and hit you".

This result shocked the physics community and Rutherford was forced to conclude that the contemporary model of the atom was incorrect. In order to explain how some of the α particles could be recoiled at such extreme angles, he proposed that atoms were actually made up mostly of empty space. Rutherford postulated that there was a very dense positively charged core or "nucleus" at the center of the atom with a diameter on the order of 1 fm and that the electrons then orbited the nucleus, similar to a mini-solar system. With this abrupt conceptual change in atomic thinking the seed of modern physics was planted.

Rutherford's atomic model envisioned the electrons as tiny particles held in orbit around the nucleus due to the electrostatic attraction. In this respect, it was very much like a mini-solar system; the electrons were analogous to the planets, the nucleus to the sun, and the electrostatic force representative of the force of gravity. This, however, posed a serious problem because accelerating electrons (those with a curved trajectory) were known to radiate energy by emitting photons. If electrons were orbiting around the nucleus in this fashion then they should quickly lose potential energy and spiral into the nucleus, thereby making the atom unstable. This model, however, obviously needed to

be modified since orbiting atomic electrons did not appear to radiate photons and, more importantly, the atom itself is stable.

Meanwhile, at the end 19^{th} century, Max Planck was working on a solution to a problem called the "ultraviolet catastrophe". Blackbody radiators, theoretical objects that completely absorb all wavelengths of thermal radiation and then reradiate the energy, were easily modeled at high wavelengths but not at low wavelengths. Derived from classical thermodynamics, the Rayleigh-Jeans law agreed with experimental data at high wavelengths but predicted an infinite amount of energy to be radiated at the lower end of the spectrum, an obvious impossibility.

Planck found in 1900 that he could fit the data with an empirical formula. However, the only way that he could explain the reasoning behind the function was if he made the assumption that the blackbody could only absorb and emit energy in discrete packets. In addition, he found that these quanta were proportional to the frequency of the radiation:

$$E = h \cdot f \qquad \text{Eq. [1.1]}$$

where $h = 6.626 \times 10^{-34}$ J·s and was given the name Planck's constant.

Building on Planck's postulate, in 1905 Einstein proposed that electromagnetic radiation was actually a stream of particles, not a wave. This idea stemmed from the photoelectric effect, in which incident light on a surface ejects electrons from the surface. The wave theory of light predicts that the kinetic energy of the scattered electrons will depend on the intensity of the radiation. However, experiments showed that the energy of the ejected electrons does not depend on the intensity of the light, only on the frequency. Einstein reasoned that each particle of light, called a photon, is responsible for ejecting one electron and that the kinetic energy of the electron directly depends on the energy, and hence the frequency according to equation 1, of the incident photon.

It then followed that in 1913, two years after Rutherford's discovery of the nucleus, Niels Bohr, a student of Rutherford's, decided to apply Max Planck's concept of quantized energy to electron orbits. He postulated that electrons could only occupy certain discrete energy levels corresponding to specific orbits. It was possible for an electron to get bumped to a higher orbit, by absorbing a photon, but eventually, the electron would drop back down to the lowest possible state, reemitting another photon whose energy is equivalent to the difference in potential energy of the two orbits in the process. In addition, he asserted that an electron in its lowest orbital energy state should be stable.

While at the time not even Bohr himself really understood *why* electrons behaved this way, the picture agreed quite well with experimental data. In particular, an electron

of an atom was observed to absorb and emit photons at very precise wavelengths described by the Rydberg-Ritz formula

$$\frac{1}{\lambda} = R \cdot Z^2 \cdot \left(\frac{1}{n_2^2} - \frac{1}{n_1^2} \right) \qquad n_1 > n_2 \qquad \text{Eq. [1.2]}$$

where λ is the wavelength of the photon, Z is the the atomic number of the atom, R is the Rydberg constant, and n_1 and n_2 are positive integers.

It wasn't until 1924 that Louis de Broglie first proposed an explanation for these quantized orbits. Following the never-ending debate of whether light was made of particles or waves, de Broglie applied the wave-particle duality concept to matter. He hypothesized that Einstein's equation for the energy and momentum of a photon might apply also to matter, thereby giving every particle a wavelength. The equation then becomes

$$\lambda = \frac{h}{p} \qquad \text{Eq. [1.3]}$$

where f is the frequency of the wave, λ is the wavelength, E is the energy, p is the momentum, and h is Planck's constant. Hence, de Broglie predicted that electrons (and all matter) should have wave-like properties. This, he claimed, could account for the stable, non-radiating orbits. An electron could only occupy an energy state in which a standing wave could constructively exist along the distance of the orbital circumference. In order for the electron's wave to not destructively interfere with itself, the electron was only allowed to have certain energies. In addition, this model dictated that there should be a lowest energy state of an electron and that it couldn't spiral into the nucleus because this would mean having a lower energy than the state that corresponded to one standing wavelength along the orbit. Experimentally, these energies agreed perfectly with the Rydberg-Ritz formula, lending credence to this "matter wave" theory.

In the following decades, physicists such as Erwin Schrödinger, Werner Heisenberg, Wolfgang Pauli, Paul Dirac and Enrico Fermi among others developed the theory of Quantum Mechanics, describing the world in terms of wave functions and probabilities. Quantum Theory is based on the premise that everything, matter and energy, is granular or quantized in nature. This concept of a smallest size, a minimum energy level, an elementary, fundamental chunk of stuff in nature is also the foundation for the Standard Model, the current theory of elementary particles and their interactions.

1.2 The Standard Model

1.2.1 Fundamental Forces

There are four basic ways that particles interact with each other:

1. The gravitational force
2. The electromagnetic force
3. The weak nuclear force
4. The strong nuclear force

Gravity is the familiar force that pulls things to the surface of the Earth and holds the planets in orbit around the sun. It is an exclusively attractive force between all particles that possess mass inversely proportional to the square of the distance. It is the weakest of the four forces, having a coupling constant, the dimensionless measure of the relative strength of the force, of ~ 10^{-39}, over 35 orders of magnitude less than any of the other forces. Therefore, the gravitational interaction does not significantly contribute to sub-atomic particle interactions. However, since the gravity has an infinite range and is only attractive, when a bodies mass becomes great enough this force is eventually seen on a macroscopic scale. Again, the sun and planets are excellent examples. In addition, it is the only force that is not incorporated as a part of the standard model.

The electromagnetic force accounts for the rest of the macroscopic forces encountered in everyday life such as friction, contact forces, and magnets. While these examples are complex manifestations of the electric force, at the atomic level it is simpler; like charges repel, opposite charges attract. The electric coupling constant is ~ 1/137 and its range is infinite. This force is not the dominant force on a macroscopic scale because most matter has approximately an equal number of positive and negative charges and is electrically neutral.

The weak nuclear force is responsible for β-decay, when a neutron decays into a proton, electron, and antineutrino. Its coupling constant is ~ 10^{-6} but is mediated by very massive particles which have a lifetimes on the order of 10^{-27} seconds, giving the weak force a range of ~ 10^{-3} fm.[1] At energies significantly higher than 100 GeV, the weak force and the electromagnetic force unify and show themselves to be different manifestations of one interaction, the *electroweak* interaction.[2]

The strong nuclear force is what bind the nucleons together inside a nucleus. The strong interaction has the largest coupling constant at ~ 1 and is a unique force for two reasons. First, in stark contrast to the electromagnetic force, it has three different charges instead of one. Secondly, it is a non-Abelian, meaning the mediating particles also carry charge and can therefore interact with each other. As a result of these two properties, the

strong force actually nullifies itself beyond a range of about 1 fm, comparable to the diameter of a nucleon. [1]

1.2.2 Proliferation of Particles

Prior to the 1930's, protons and electrons were thought to be the elementary particles of which everything is made. Then, in 1932, both the neutron and positron were discovered. The neutron was very similar to the proton except that it had no electric charge. It was also found to accompany protons within the atomic nucleus. The positron is essentially identical to the electron except it has positive, instead of negative, charge. This confirmed the existence of anti-matter, which was comforting to the physics community for two reasons. The first is that Dirac predicted anti-matter in 1927 in an attempt to explain negative energy solutions to the Schrödinger equation, which is the basis for Quantum Mechanics. Secondly, physicists always hope for and strive towards finding symmetry in nature as a general rule. The 1940's led to the discovery of the muon (μ), a particle nearly identical to the electron except that it is nearly 200 times more massive, and the pi meson (π), or pion, a particle with an intermediate mass between the proton and electron. This trend of discovering new seemingly "elementary" particles continued for decades and, in fact, new particles are still being discovered today.[2]

Particle physics experiments over the past 50 years have found hundreds of sub-atomic particles to exist in addition to the proton, neutron and electron. So many were found, and with such apparent systematic order, that physicists were forced to the realization that they were not discovering increasingly more truly fundamental particles. A particle substructure seemed imminent. Murray Gell-Mann and George Zweig predicted in 1963 a model of quarks (named after a James Joyce quotation) providing underlying structure to two particular classifications of particles, baryons and mesons, which was confirmed in 1969. Current theory holds that all matter is made from only two families of fundamental particles: quarks and leptons.[1]

1.2.3 Classification and Characteristics of Particles

The Pauli exclusion principle states that no two particles of matter with anti-symmetric wave functions can exist in exactly the same energy quantum state. This essentially refers to the common sense notion that no two identical particles can be in exactly the same place at exactly the same time (although at the sub-atomic level, the argument is a bit more subtle). Particles that obey the Pauli exclusion principle are called *fermions*. All fermions are also characterized by odd half-integer spins (1/2, 3/2, 5/2, …). Particles that do not obey the Pauli exclusion principle are called *bosons*. Bosons have either zero or integer spin. All particles existing in nature are either fermions or bosons.[2]

Sub-atomic particles are also classified by the interactions in which they participate. One such distinction is the *hadron*, a particle that experiences the strong

nuclear force. All hadrons are composite particles made of quarks (more on quarks later). There are two types, *baryons* and *mesons*. Baryons are fermions that are bound three-quark states. The proton and neutron (collectively referred to as nucleons) are the "lowest lying" or least massive baryons in a spectrum of over 100 particles. The proton is the only stable baryon.[1]

Mesons are bosons that consist of quark/anti-quark pairs. They are bosons and therefore have either zero or integral spin. All mesons are quickly decaying particles, the longest living being the pion with a decay time of ~ 0.26 ns.[3]

1.2.4 Truly Fundamental Particles??

All fermions, and therefore all matter, are made out of either quarks or leptons. The current standard model theorizes that these are truly elementary, having no internal substructure. While the pattern throughout history has been for nature to consistently reveal ever smaller constituent parts to what scientists believe are fundamental particles, current theory still maintains that quarks and leptons are probably *the* elementary quanta of nature.[1] The hierarchy of fundamental particles is shown in Figure 1.1 and some of their properties are shown in Table 1.1.

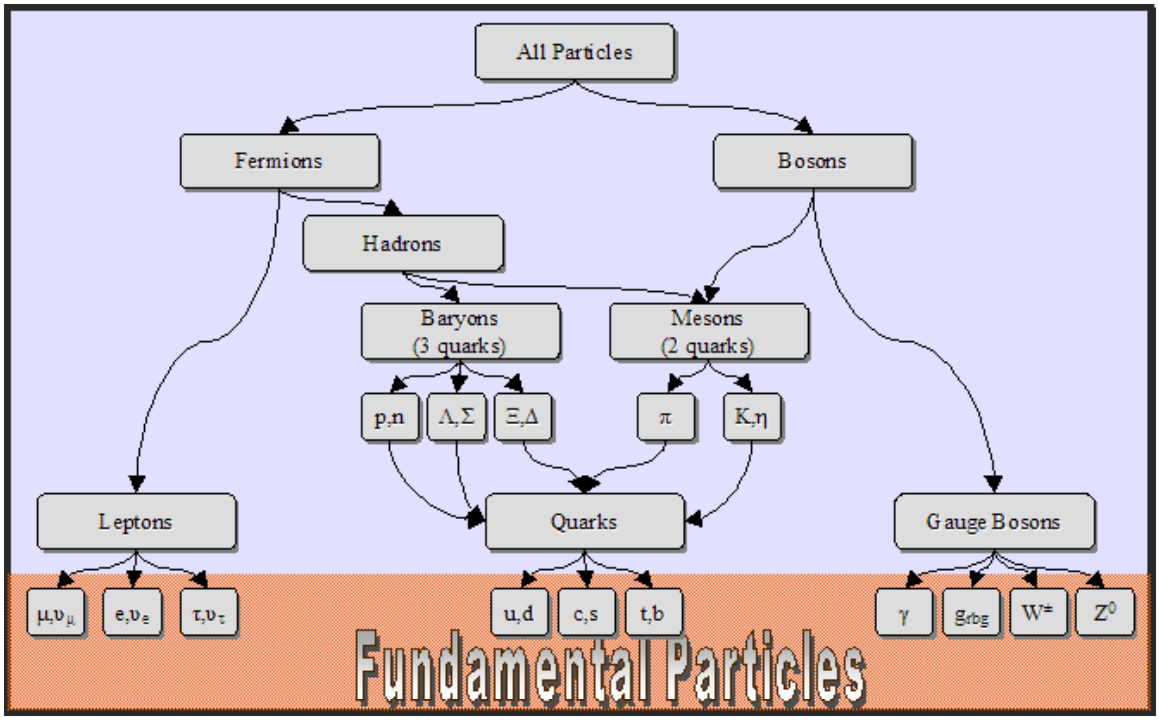

Figure 1.1

Not all of the baryons and mesons are listed.

There are a total of 12 known leptons, 6 particles paired with their anti-particles. They can be divided into 3 families or generations. The electron (e) and electron neutrino

(υ_e), the muon (μ) and the muon neutrino (υ_μ), and the tau (τ) and tau neutrino (υ_τ). The electron, muon and tau are essentially identical particles except that they are increasingly more massive. Therefore, the muon always decays into an electron and neutrino/anti-neutrino pair, while the tau has enough mass to have nearly a dozen different decay channels.[4]

Quarks

	mass (MeV/c^2)	charge (e)
up	~ 400	2/3
down	~ 700	- 1/3
charm	~ 1500	2/3
strange	~ 150	- 1/3
top	~ 1.74×10^5	2/3
bottom	~ 4700	- 1/3

Leptons

	mass (MeV/c^2)	charge (e)
e	0.511	-1
υ_e	< 1.6×10^{-5}	0
μ	106	-1
υ_μ	< 0.3	0
τ	1780	-1
υ_τ	< 40	0

Table 1.1

Leptons are not baryons and, consequently, do not interact via the strong nuclear force. The neutrinos are all electrically neutral, but the e, μ, and τ all carry charge -1*e* (or +1*e* if they are anti-particles). The primary interaction mechanism is the electromagnetic force. Since neutrinos are electrically neutral, they interact extremely infrequently with other matter (neutrinos regularly pass unaffected straight though the Earth; scientists can only measure them rarely and with great difficulty).[2]

Quarks are also organized in a way that parallels the leptons. Twelve total quarks, 6 particles and their anti-particles are arranged into 3 families. They are given arbitrary names; the 1st family consists of an *up* quark and *down* quark, the 2nd of a *charm* and a *strange* quark, and the 3rd of a *top* and a *bottom* quark. The up, charm, and top quarks have electric charge (2/3)*e* and the down, strange, and bottom all have electric charge (-1/3)*e*. The families appear to have identical particle properties except for their increasing masses.[2]

Where the families of both quarks and leptons differ is in their decay channels. While all decaying particles conserve things like mass/energy, electric charge, momentum and angular momentum, they also conserve quantities called lepton numbers (L) and baryon numbers (B).[2]

Every lepton in a given family has an associated lepton number. In the 1st family, e and υ_e both have L_e; in the 2nd family, μ and υ_μ both have L_μ; in the 3rd family, τ and υ_τ both have L_τ. The corresponding antiparticles are identical in all ways except for a reversal of sign in the Lepton numbers and electric charge. This is an example of an allowed decay (the line above the υ_e signifies an anti-particle):

$$\mu \Rightarrow e + \overline{\nu}_e + \nu_\mu$$

The quarks have a simpler system for determining baryon numbers. Each quark carries B of 1/3. However, to complicate the quark scheme, quarks also conserve 6 properties (in strong nuclear interactions): up-ness, down-ness strangeness, charm, top-ness and bottom-ness. The up quark has a up-ness of 1, the down quark has a down-ness of 1, the strange quark has a strangeness of –1, the charm quark has a charm of 1, the top quark has a top-ness of 1 and the bottom quark has a bottom-ness of 1. The anti-quarks all have a sign reversal of these and baryon number from their associated quark.[2]

The Pauli exclusion principle states that no two particles can be in the same quantum state. However, this is exactly what seems to happen with, for instance, the Δ^{++} particle which consists of three identical up quarks, all in the same state. This particle is observed in nature but cannot, within the framework presented thus far, be consistent with Pauli's exclusion principle.[2]

As mentioned earlier, quarks have three different kinds of charge associated with them. These charges are labeled red, green and blue. Anti-quarks are labeled anti-red, anti-green, and anti-blue. It is important to realize that these do *not* refer to actual colors, rather they are a merely labels, a convenient way to categorize something that is otherwise completely unintuitive. There are, however, analogies to color that apply to these properties of quarks.[1]

Isolated quarks has never been found in nature. Only baryons (3 quark states) and mesons (quark/anti-quark states) have been observed. The presence of quarks has been deduced from experiments where nuclei are probed with extremely high-momentum particles. As the incident particles gain more energy, their quantum mechanical wavelength decreases (equation 1.3). In electron-proton scattering with electron energies of ~ 4.9 GeV/c^2, excited nucleon states were observed, indicating nucleon sub-structure. In addition, *deep inelastic scattering* experiments were performed on nuclei that showed analogous results to Rutherford's gold-foil experiment on atoms. Deep inelastic scattering refers to colliding electrons or protons with nucleons using enough energy to penetrate deep into the nucleon (anywhere between 20 and 400 GeV, depending on the experiment). Nearly identical arguments to Rutherford's can be made from these data supporting the case for composite systems made built of three quarks.[1]

In order to describe why lone quarks are never observed (and also to maintain consistency with the Pauli exclusion principle) physicists have introduced this new property, color. Then, quarks are only able to exist in colorless (white) combinations. Therefore, there are a limited number of groupings that can occur: equal mixture of red, green, and blue (R-G-B), equal mixture of anti-red, anti-green, and anti-blue (aR-aG-aB) or equal mixture of color and anti-color (R-aR, G-aG, B-aB). This explanation is consistent with Pauli's exclusion principle since, in the above example, the three up quarks are not identical but are red, blue, and green instead. The color concept also has very profound implications for how quarks interact with each other.[1]

1.2.5 Other Fundamental Particles…

While all matter is made of quarks and leptons, there is another classification of fundamental particle called the *gauge boson*. As the name implies, they are bosons, but more specifically, they are particles that mediate interactions. The standard model holds that there must be carrier particles for all interactions that must be associated with quantum excitations of the field of some interaction. Photons are the particles that mediate the electromagnetic interaction, the $W^{\pm}$ and Z^0 bosons facilitate the weak nuclear interaction, and gluons mediate the strong nuclear force.[2]

Photons have exactly zero mass and therefore, according to special relativity, travel at the 'speed of light' or $c \sim 3 \times 10^8$ m/s. One consequence of this is that photons never decay and subsequently have an infinite range. The $W^{\pm}$ and Z^0 bosons are both very massive particles (79.8 and 91.2 GeV/c^2, respectively) and so their lifetimes are very short, on the order of $\sim 10^{-27}$ s, corresponding to a range of $\sim 10^{-3}$ fm for the weak interaction.[1]

As mentioned above, gluons differ from the other gauge bosons in two ways: they have eight charges and they conform to a non-Abelian gauge theory. According to SU3 theory, since there are three kinds of quarks (red, green, blue) there are eight types of bi-colored gluons. Each gluon must consist of a color and an anti-color because gluons actually exchange color between quarks. In addition, every gluon carries a color charge also, meaning the gluons interact with each other. These two special properties make the strong force nearly impossible to conceptualize, however, they result in interesting measurable results.[1]

This non-Abelian gauge theory of gluons explains why isolated quarks have never been found. If a quark is pulled from a hadron, the field lines of the strong force don't spread out and get weaker with distance like electromagnetic charges. Because the gluons interact with each other, the field lines actually condense and get squeezed into a tube-like region (see figure 1.2). The energy that it takes to separate them increases linearly as a function of distance, meaning that it would take an infinite amount of energy to completely separate two quarks.[1]

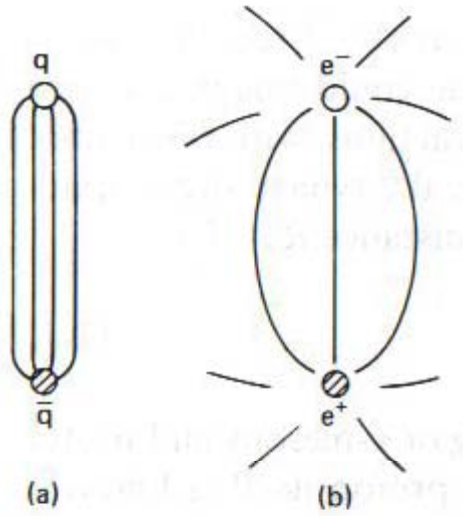

Figure 1.2
The quark/anti-quark color field with V(r) ~ r (a), and the e^+e^- coulomb field with V(r) ~ 1/r (b).[1]

The consequence of this is, in fact, observed. As a quark is pulled away from another quark, the potential energy increases until there is enough energy in the system to create a quark/anti-quark pair. When this happens one of the newly created quarks joins the original hadron, taking the place of the departed quark while the other created quark pairs with the departed quark to form a new meson. In this way, solitary quarks cannot exist in nature and anytime an attempt is made to isolate one, more hadrons are produced.[1]

The other interesting feature of the strong force is that even though it is theoretically infinite in range (gluons are massless), hadrons do not interact with each other via the strong force beyond a range of ~ 1 fm. This is because gluons carry charge and cannot travel very far outside the hadron until they get pulled back into the hadron by the strong force. In comparison, photons do not carry electric charge and, therefore, have nothing stopping them from flying away from an electron, thus giving the electric force an infinite range.[1]

Hadrons do, however, stick together via the strong force. The strong force can also be conducted by mesons (quark/anti-quark pairs). Mesons can, in a sense, house gluons within them and conduct the strong force for a short distance. It is interesting to note that the lowest-lying meson is the pion (~ 140 GeV/c^2). Governed by the Heisenberg uncertainty principle, a virtual particle can travel

$$\Delta x \approx \frac{h_{bar}}{m \cdot c} \qquad \text{Eq. [1.4]}$$

before it is reabsorbed back into the vacuum. Substituting in the mass of the pion, the distance that a pion could conduct the strong force between hadrons is ~ 1 fm, which is exactly what is observed. When mesons conduct the strong force it is generally referred to as the *residual* strong force.[1]

2. Experimental High Energy Physics

2.1 Introduction

2.1.1 Basics

Elementary particles are inherently undetectable to any of a person's senses. Alone, they can't be seen, felt, smelled, tasted, or heard. Obviously, scientists need ways to detect these particles in order to study them. Fortunately, as outlined in the previous sections, every particle possesses specific properties and participates in particular interactions that make them distinguishable.

In order to study these sub-microscopic particles, scientists build machines called *accelerators* that accelerate and then collide particles. When particles are smashed together their energies combine and a multitude of particles are produced, spraying out from the *collision point*, the location where the particles collide. The collision point is then surrounded by an array of detectors, typically in a cylindrical configuration that measure the momenta, positions, and electric charges of the produced particles.[4]

In high energy physics, scientists need to reach minimum particle beam energies on the order of ~ 1 GeV or ~ 1.603×10^{-10} Joules. One electron volt is the amount of energy given to an electron that has been accelerated across a potential difference of 1 volt. According to equation 2.5, this corresponds to a spatial resolution of ~ 1 fm, about the size of a nucleon.[1]

There are two main types of configurations for accelerators: linear (*linacs*) and circular (*synchrotrons*). Linear accelerators consist of a series of accelerating tubes laid out in a straight line. Each pair of accelerating tubes has oppositely charged potentials that accelerate charged particles located in between them. The tubes are controlled by a generator that pulses the potentials at an extremely high frequency. The frequency is carefully set so that the particles ride a "wave" of acceleration until the end of the linac where they are finally slammed into the target particles.[5]

In addition, the beam particles need to be focused along the length of the linac by magnetic fields to make sure that the particles don't stray from a linear course. Linacs cannot produce a continuous beam of particles; they can only accelerate them in bunches or packets. The final energy of the accelerated particles is essentially determined by:

$$E = n \cdot Z \cdot e \cdot U \qquad \text{Eq. [2.1]}$$

where n is the number of accelerating tubes, Ze is the charge of the particle, and U is the potential difference between the tubes.[5]

The largest linac in the world at this time is the Stanford Linear Accelerator Center (SLAC). It is ~ 3 km long and contains ~ 10^5 accelerating stages resulting in energies of ~ 50 GeV.[5]

Synchrotrons are circular accelerators that are similar in concept to the linacs but are more sophisticated. The beam particles repeatedly pass through the same accelerating tubes and are focused by a magnetic field to have a circular path. The generator frequency, ω, and the focusing magnetic field, B, must synchronously adjust to the momentum of the beam particles in order to keep the particles in a circular orbit in the following way:

$$\omega = n \cdot \frac{c}{r} \cdot \beta \qquad \text{Eq. [2.2]}$$

$$B = \frac{p}{Z \cdot e \cdot r} \qquad \text{Eq. [2.3]}$$

where n is a positive integer and r is the radius of the beam orbit.[5]

In addition, whenever a charged particle is radially accelerated it loses energy by photon emission called synchrotron radiation. This energy loss must be compensated for by the accelerating stages and is found to be

$$-\Delta E = \frac{4 \cdot \pi \cdot \alpha \cdot h_{bar} \cdot c}{3 \cdot r} \cdot \beta^3 \cdot \gamma^4 \qquad \text{where} \quad \gamma = \frac{E}{m \cdot c^2} \qquad \text{Eq. [2.4]}$$

where α is the electromagnetic coupling constant and β is the velocity of the particle divided by the speed of light. For highly relativistic particles ($\beta \approx 1$), the mass dependence of the energy loss is inversely proportional to the mass of the particle to the fourth power. The energy loss rate for electrons is ~ 10^{13} times higher than it is for protons, making synchrotrons more suitable for experiments involving accelerated nucleons than electrons.[6]

Synchrotrons are usually built to collide two beams head on. This is done by accelerating the beams in opposite directions giving them counter-rotating orbits. Then, when the desired beam energies are reached, the beams are deflected to collide with one another. At Brookhaven National Laboratories, the Relativistic Heavy Ion Collider (RHIC) is an example of a synchrotron. RHIC collides gold nuclei at energies of about 100 GeV per Au nucleon. While this amount of energy is enough to recreate conditions that existed less than 10^{-32} seconds after the Big Bang and reach temperatures nearly 10^5 times hotter than the center of the sun, in macroscopic terms it is about the amount of energy of a fly landing a wall (wow!).[7]

2.1.2 Resolution

According to Heisenberg's Uncertainty Principle, the spatial resolution of the probing particle is determined by [5]

$$\Delta x \approx \frac{h_{bar} \cdot c}{E} \qquad \text{Eq. [2.5]}$$

Therefore, higher collision energies lead to a deeper probe of the colliding particles. The center of mass energy determines the amount of available energy in the collision. In order to investigate matter at its most fundamental level, it is preferential to maximize E_{cm}. The center of mass energy of the collision in a linear accelerator is

$$E_{cm} = \sqrt{2 \cdot E_a \cdot m_b \cdot c^2 + \left(m_a^2 + m_b^2\right) \cdot c^4} \qquad \text{Eq. [2.6]}$$

where E_a and m_a are the energy and mass of the incident particle, respectively, and m_b is the mass of the target particle. In high energy experiments where the masses of the particles are negligible compared to the energies this can be approximated by [4]

$$E_{cm} = \sqrt{2 \cdot E_a \cdot m_b \cdot c^2} \qquad \text{Eq. [2.7]}$$

The available energy in the center of mass frame scales as the square root of the beam energy. For this reason, particularly for hadronic collisions, synchrotrons tend to be more preferable than linear accelerators. Since there are two beams colliding in synchrotrons, the lab frame is the center of mass frame, thereby making the center of mass energy a linear function of twice the beam energy.

2.1.3 Coordinate Systems

In order to describe a particle's mass and motion there are four pieces of information necessary:

In Cartesian – (p_x, p_y, p_z, E)
In Spherical – (p_{radial}, θ, φ, E)
In Cylindrical – (p_{radial}, θ, p_z, E)

From these four components, the three dimensional velocity and the mass of the particle can be extracted. The cylindrical nature of the detector suggests a different coordinate system than the conventional Cartesian system; the most convenient system is a sort of combination of cylindrical and spherical. The four components are (P_T, η, φ, E).

Defining the z-axis along the beam line, the y-axis along the vertical line, and the x-axis horizontal and perpendicular to the beam line, the *transverse momentum* (P_T), *pseudo-rapidity* (η), and azimuthal angle (φ) can determine the location of any particle. The transverse momentum is the momentum of the particle which is perpendicular (or transverse) to the beam line. It is the projection of the total momentum onto the x-y plane. The azimuthal angle is the angle around the beam line in the x-y plane. Finally, the pseudo-rapidity (η) is an estimate of the *rapidity* (Y) if the particle is traveling close to the speed of light. The rapidity relates to the polar angle, θ_{cm}, the angle off the beam line in the frame of reference of the center of mass, by

$$Y = \frac{1}{2}\cdot\ln\left(\frac{1+\beta\cdot\cos(\theta_{cm})}{1-\beta\cdot\cos(\theta_{cm})}\right) \qquad \text{Eq. [2.8]}$$

Where, again, β is the velocity of the particle divided by the speed of light. If $\beta \approx 1$ then equation 2.8 (after some algebra) can be approximated by

$$\eta = -\ln\left(\tan\left(\frac{\theta_{cm}}{2}\right)\right) \qquad \text{Eq. [2.9]}$$

Since particles produced in high-energy collisions are essentially moving at the speed of light, η is a good approximation for Y (most of the time). Figure 2.1 illustrates some of these relationships between the Cartesian system and the (P_T, η, φ, E) system.

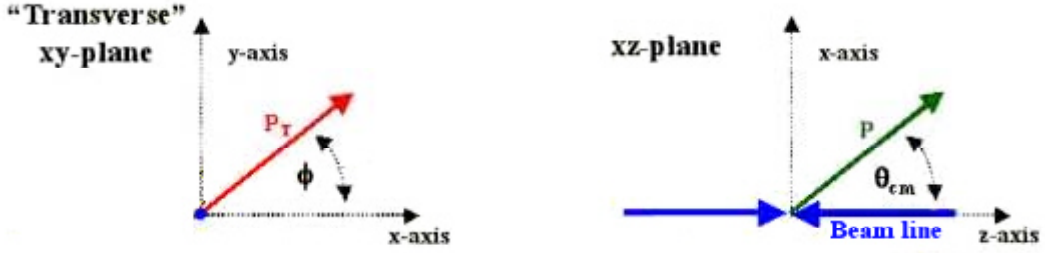

Figure 2.1

Relationships between (p_x, p_y, p_z, E) and (P_T, η, φ, E) coordinate systems.[8]

The advantage to using the coordinate Y and η instead of θ_{cm} is that the Y or η distribution (the differences between rapidities) is invariant under Lorentz transformations. When a distribution of η's is transformed from one frame to another, (for instance, the lab frame to the center of mass frame), each individual η is simply shifted by some constant. Particle detectors generally do not measure at $|\eta|$ larger than ~ 5, which corresponds to a θ_{cm} of less than 1 degree (~ 0.772 degrees). The dependence of η on θ_{cm} is shown in figure 2.2:

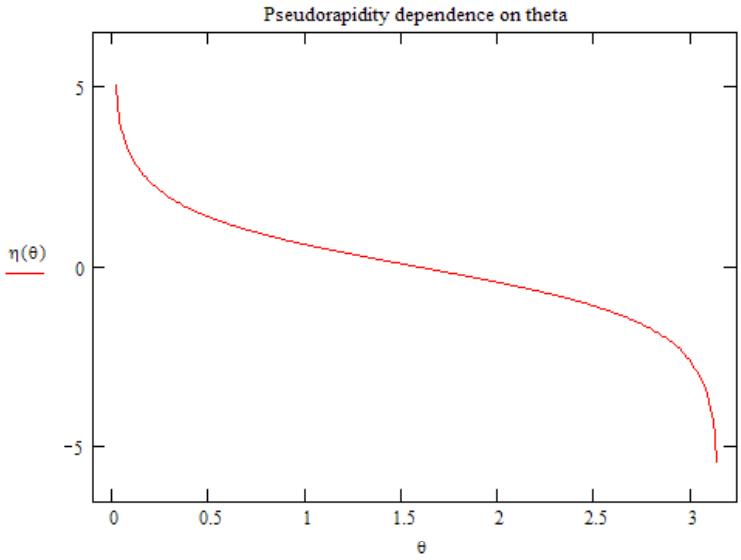

Figure 2.2
Pseudo-rapidity dependence on θ_{cm}. This relationship is nearly linear in the region $|\eta| < 2.5$.

These four components (P_T, η, φ, E), with E being the total energy of the particle, can describe a particles mass and motion through three dimensional space.

2.1.4 Measuring Particles

Most fundamental particles interact with matter in some measurable way. The primary processes for charged particles and photons is the electromagnetic interaction, in particular, inelastic collisions with atomic electrons. Neutrons, however, are usually observed in strong nuclear interactions.[6]

When a charged particle passes through matter, there are generally two things that occur. The particle loses energy and it is deflected from its original trajectory. Different types of particles are best detected my different methods. Therefore, by layering many different types of detectors around the collision point, information can be gathered and pieced together about the collision event. A multi-component detector is referred to as an *experiment*.[6]

2.2 ATLAS Detector

The ATLAS (A Toroidal LHC ApparatuS) experiment is an example of a multi-component detector that is presently being built for use at the Large Hadron Collider (LHC) at CERN (European Organization for Nuclear Research) in Geneva, Switzerland.

The LHC is a synchrotron that accelerates two beams of protons in opposite directions around a large (~ 17 mile circumference) circle. When completed, the LHC will collide the proton beams together at an energy of 14 TeV (14 x 10^{12} eV). Plans have also been set to later collide heavy nuclei (lead, Pb) at energies of up to 1150 TeV.[9]

There are four main sub-systems to the ATLAS experiment:

1. Magnet System
2. Inner Detector
3. Calorimeters
4. Muon Spectrometer

The ATLAS experiment is an amazing one for both its simplicity and its sophistication. It attempts to measure all types of particles that interact with matter but uses a minimal number of sub-detectors. Some, such as the PHENIX (Pioneering High Energy Nuclear Interaction eXperiment) at RHIC, use more than 10 main sub-systems.[7] Figure 2.3 shows a three-dimensional cutout view of the ATLAS experiment:

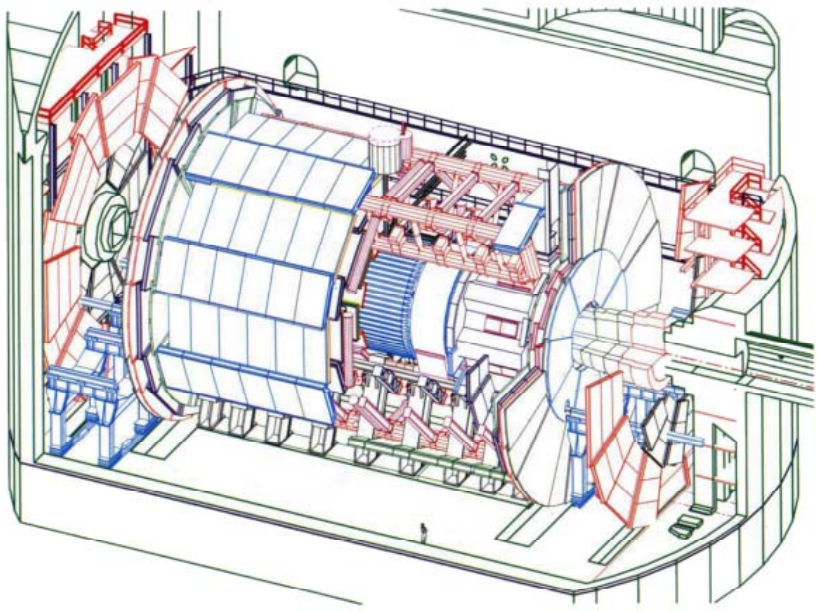

Figure 2.3
Three-dimensional view of the ATLAS experiment. Some muon chambers and parts of the barrel toroid are removed to show the inner structure of the detector. A person is drawn in the lower center of the image to give perspective.[10]

A high energy particle physics experiment must be capable of measuring particle energies, momenta, directions, and charges. With these properties in hand, a particle's point of origin and mass can be inferred. Contrary to naïve expectations, not all particles that are measured in the detectors originate from the collision point. Some of the measured particles are the decay products (*daughter particles*) of parent particles with

very short lifetimes that decay before they hit any of the detectors. Usually this happens within a few millimeters of the collision point. In addition, any other undetectable particles (e.g. neutrinos) can be inferred by conservation laws.[4]

In order to accomplish all of these measurements, the sub-systems of the experiment are layered cylindrically around the beam line. Different particles leave tracks in different parts of the detectors. A very generalized scheme of which detectors measure which particles is shown in figure 2.4:

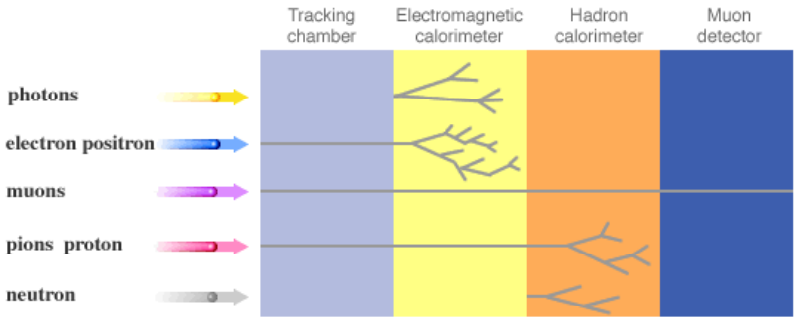

Figure 2.4

Different particles leave tracks in different detectors. By studying the tracks, scientists can determine what type of particles have passed through the experiment.[9]

By essentially using similar logic to figure 2.4, all of the particles that are created in the collision can be recorded and their trajectories can be mapped. The individual detector components are explained in greater detail in the following sections. Unless otherwise stated, all material in sections 2.2.1 – 2.2.3 is referenced from CERN/LHCC 94-43, 1994.[12]

2.2.1 Magnet System

Strong magnetic fields are needed throughout the experiment for two main reasons. First, they make charge determination very easy. Every charged particle will have a curved path in the field. Positively charged particles bend in one direction, negatively charged particles in the other. The second chief motivation for having magnetic fields is for determining the momenta of particles. Particles with high momenta will have less curved trajectories within the field whereas those with low momenta will have trajectories that are more heavily influenced by the field and therefore more curved.[5]

The layout of the magnet system with respect to the other detector sub-systems is shown in figure 2.5:

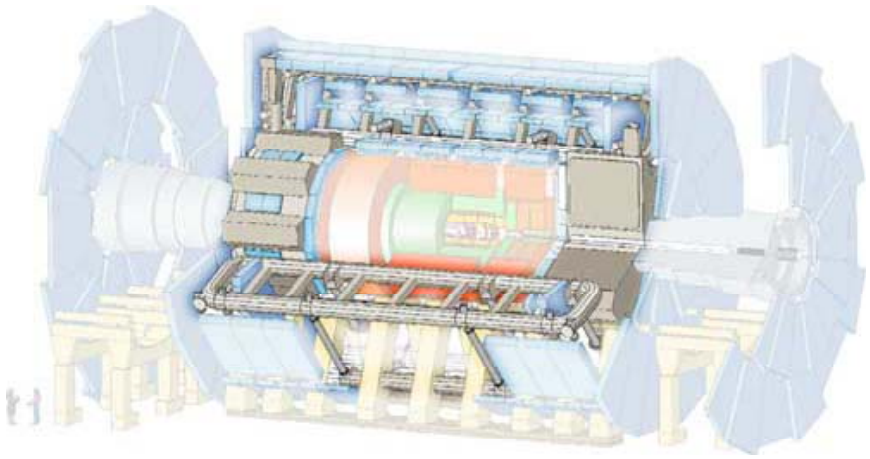

Figure 2.5

ATLAS experiment: the highlighted grey portions are the magnets.[9]

There are three types of superconducting magnets in the ATLAS experiment. As shown in figure 2.6, the *barrel toroids* (red), produce a cylindrical field which curves around the beam line, through the openings in the toroids. There are 8 barrel toroid loops assembled radially and symmetrically to the beam axis. The *end-cap toroids* (green) produce a field around the ends of the detector. Each end-cap toroid consists of 8 flat coils arranged again radially and symmetrically around the beam axis; however, they are offset from the barrel toroids by 22.5°. A interior view of an end-cap toroid is shown in figure 2.7.

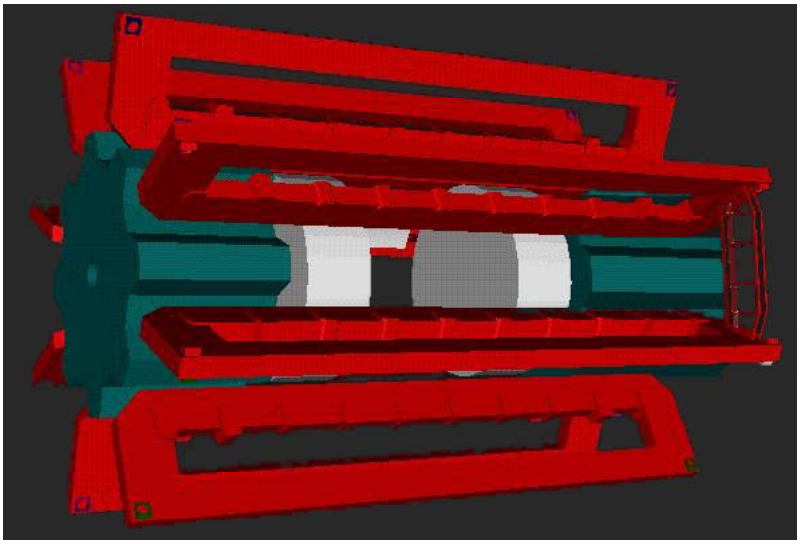

Figure 2.6

Illustration of the magnet system. Barrel toroids (red), end-cap toroids (green), and solenoids (white) are shown.[11]

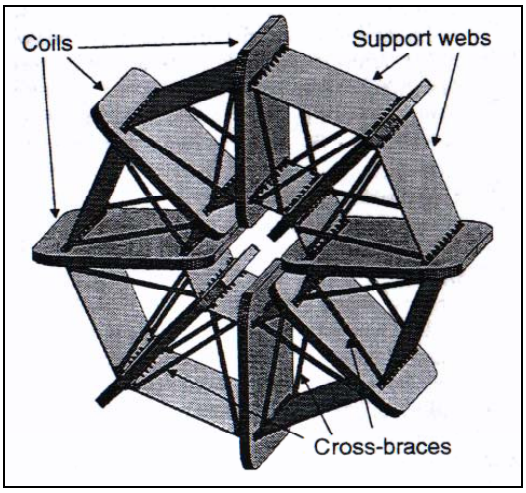

Figure 2.7
Interior view of an end-cap toroid.

The *solenoid* magnets are the white disks in figure 2.6 and produce a field that is parallel to the beam line. This axial magnetic field has a strength of 2 T along the beam line. The force from the solenoids effectively tries to make the collision particles rotate around the beam line. This is different from the toroidal magnets which effectively "push" the particles in a direction parallel to the beam line.

In high energy experiments, most of the particles that are studied are moving with velocities close to the speed of light. With such high velocities, extremely strong magnetic fields are necessary to appreciably deflect the particles. In addition, these fields must be held at very precise levels in order to keep track of what a particular particle will do.

2.2.2 The Inner Detector

The inner detector is used to measure the paths or *tracks* of all charged particles within a radius of 115 cm and |z| of 245 cm symmetrically around the beam line (z-axis). The configuration of the inner detector with respect to the rest of the experiment is shown in figure 2.8.

Figure 2.8
ATLAS experiment: the highlighted central section is the inner detector.[9]

The inner detector consists of three main types of mini-detectors: the *silicon pixel detectors*, the *silicon strip detectors*, and the *transition radiation tracker* (TRT). Figure 2.9 shows a close-up of the inner detector and identifies its components.

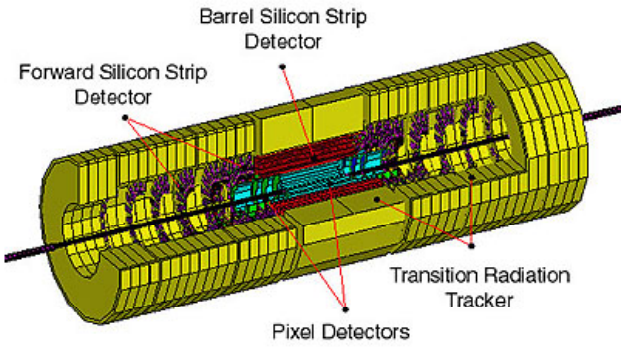

Figure 2.9
The inner detector.[9]

The sensors closest to the collision point are the pixel detectors and are divided into two sub-types. The *barrel pixel detectors* consist of two thin layers of silicon that wrap around the beam line. They are positioned at radii of 11.5 and 16.5 cm and have lengths along the z axis of 70 and 90 cm, respectively, and are centered on the collision point.

The *forward pixel detectors* consist of 8 (4 on either end of the barrel pixel detectors) flat "donut" shaped layers, perpendicular to the beam line. The inner radius of the forward pixel detectors is 11.5 cm while the outer radius is 21.3 cm and are positioned at lengths $|z| = 50, 55, 80$ and 85 cm from the collision point.

In both sets of pixel detectors, the silicon layers are segmented into very small rectangular regions, called pixels, with dimensions of 50 x 300 μm. When a charged particle strikes a pixel, electrons are released and a signal is measured. By tracking which pixels get "hit", the tracks of the charged particles can be reconstructed using sophisticated computer software. In total, the inner detector will contain over 140 million pixel sensors.

Pixel detectors provide extremely sensitive track measurements and can determine from where a particle originated. The are sensitive enough to precisely determine if a particle was a direct result of the initial collision or if it was a decay product of a particle that decayed within (as little as) a few millimeters of the collision point. Unfortunately, they are also extremely expensive, averaging a staggering $4.5 million per square meter.

This high level of precision is not as necessary for detectors that are farther away from the collision point. The silicon strip detectors are significantly less expensive at about $0.8 million per square meter. Arranged similarly to the pixel detectors, the strip detectors have both barrel layers wrapped around the beam line and forward layers that are perpendicular to the beam line at the ends of the barrel layers. Each layer consists of strips of silicon, instead of rectangular pixels, that are either 75 μm or 112.5 μm in width and 12 cm in length. The barrel strips are aligned parallel to the beam line while the forward strips are aligned radially.

Strip detectors can therefore measure the azimuthal angle, φ, of a particle very well. Unfortunately, the sacrifice that strip detectors make for being less expensive is that they do not yield much useful information about the pseudo-rapidity, η. This lack of precision is compensated for by combining the information from the pixel detectors with the information from the TRT.[6]

The transition radiation tracker is comparatively less expensive than even the silicon strip detectors but again sacrifices some overall precision. The TRT consists of collections of gas-wire drift tubes (370,000 tubes in total). These are small (4 mm in diameter; similar to a drinking straw) gas filled tubes of various lengths with thin wires running down the middle of them. The wires are set with an electric potential with respect to the tube walls. A mix of polypropylene and polyethylene fibers are packed around the tubes and is called the *radiator* material.

As charged particles fly though the tubes, they ionize the gas atoms along their path through the tube. While the leftover ions are attracted to the outer tube walls, the ionization electrons are attracted to the wire. The electrons drift toward the wire (hence the name drift tubes) with velocities on the order of 50 μm/ns (~ 10^5 miles/hr) toward the wire, resulting in a current on the wire. In this way, the tubes that the particles traverse can be recorded and the particle tracks can be mapped.

In addition to mapping the tracks, the TRT is also used for electron identification. Whenever an electron passes through the radiator material in between the tubes it produces *transition radiation* in the form of X-rays. It is called transition radiation because it is specifically produced when the electron passes from one medium (in this case, air) into another with a higher dielectric constant (the radiator material). In the moment that the particle crosses the boundary between the two materials, the charged particle's electric and magnetic fields are changing, thereby resulting in an electromagnetic wave.[4]

The amount of energy released in the electromagnetic wave (photon) is inversely proportional to the mass of the traveling charged particle. Since electrons are much less massive than any other particle, they emit the highest energy photons (in the form of X-rays). The resulting X-rays from the electrons are emitted in the general direction of the electrons velocity and then also interact with the gas in the drift tubes creating a much larger pulse on the wire whenever an electron passes than other heavier particles. At the energies present in ATLAS, electrons (and positrons) are generally the only particles that exhibit high enough energy transition radiation to interact with the gas in the straw tubes.[4]

A typical particle track consists of 3 pixel, 8 strip, and 36 TRT hits.[13]

2.2.3 Calorimeters

There are two major calorimeter systems in ATLAS: the *hadronic calorimeter* and the *electromagnetic calorimeter*. The placement of the calorimeters with respect to the rest of the experiment is shown in figure 2.10:

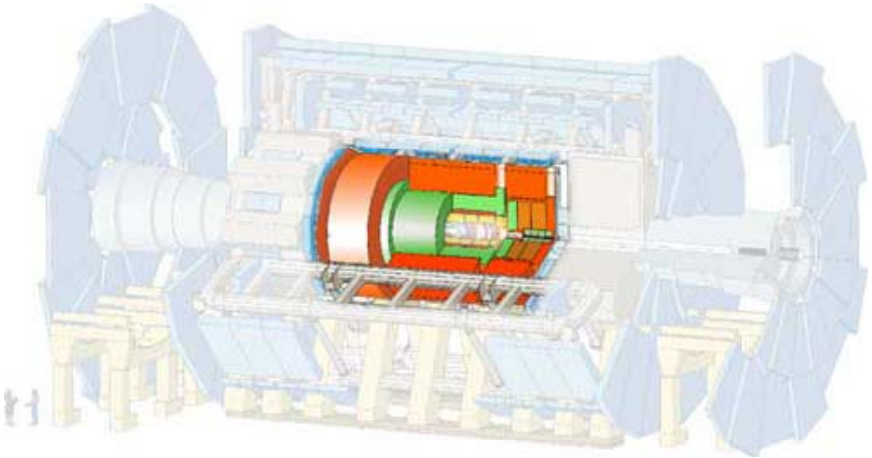

Figure 2.10

ATLAS experiment: the highlighted green section is the electromagnetic calorimeter and the orange section is the hadronic calorimeter.[9]

The electromagnetic calorimeter consists of a series of thin, stainless steel coated lead plates called *absorbers*. They are accordion shaped (see figure 2.11), with thicknesses ranging from 1.2 mm to 1.8 mm depending on their position within the detector. Plates positioned at larger η can be made thinner since particles passing through these plates will be less perpendicular to the thickness and, therefore, will pass through more lead. The plates are immersed in liquid argon and are held at a constant positive high voltage. The are spaced about 4 mm apart.

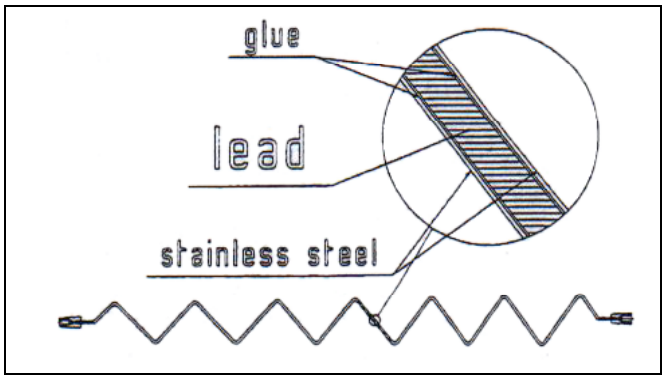

Figure 2.11
Accordion shape of a lead absorber plate.

When an electron enters the calorimeter and the first lead plate, it begins to lose energy to a process called *bremsstrahlung*. Electrons are slowed down by the field produced by atomic nuclei and radiate photons. Bremsstrahlung radiation is inversely proportional to the square of the mass of the particle. Therefore, electrons are the only particles that are light enough to exhibit bremsstrahlung in high energy experiments. These photons are then converted into electron – positron pairs in the presence of the nuclear field. This process is called *pair production*.[6]

The result of this sequence of bremsstrahlung leading to pair production leading to more bremsstrahlung (and so on) is called an *electromagnetic shower*. Eventually, the original electron's energy has been totally dispersed into the electromagnetic shower. The electrons and positrons from the shower ionize the liquid argon atoms and the resulting ionization electrons are attracted to the voltage on the plates, in a process similar to the drift tubes. This current can then be amplified and measured and is proportional to the energy of the incident particle. In addition, the distance that the incident particle travels before its energy is fully drained also lends information about its original energy. Photons originating from the collision can also be identified by the electromagnetic calorimeter; they are simply the showers that do not correspond to any tracks from the inner detector (photons pass through the inner detector unobserved). Their energies can also be calculated.[4]

In addition to electrons, all charged particles leave tracks in the electromagnetic calorimeter. Charged particles, such as pions and protons, also leave tracks in the electromagnetic calorimeter. However, due to their much larger masses (~200 times and ~2000 times the mass of the electron, respectively), these particles have enough energy to completely travel through the electromagnetic calorimeter. A particles energy cannot be determined unless it is totally absorbed by the calorimeter. Therefore, a hadronic calorimeter is placed around the electromagnetic calorimeter to measure the energies of the hadrons.

The hadronic calorimeter consists of alternating layers of steel plate absorbers and plastic scintillator tiles. When a hadron enters the scintillator it excites atoms in the scintillator. As the atoms drop back down to their ground state they emit photons. The scintillators are connected to photomultipliers that turn the photons into a measurable electric current.[5]

The scintillator – photomultiplier combination is a tool that is often used in particle physics experiments. A semi-transparent photocathode is coated with a material with a low work function and is placed at the end of the photomultiplier. When the photons strike the surface, electrons are emitted through the photoelectric effect, producing a weak current of electrons. An applied voltage inside the photomultiplier creates a potential, which accelerates the photoelectrons and slams them into an dynode producing many additional electrons. This process is repeated many times in the photomultiplier creating a cascade-like effect of electrons. When they reach the end of the photomultiplier there are enough electrons to produce a measurable pulse of current. Photomultipliers have the ability to amplify the photoelectric current by factors of millions. This process is illustrated in figure 2.12:[4]

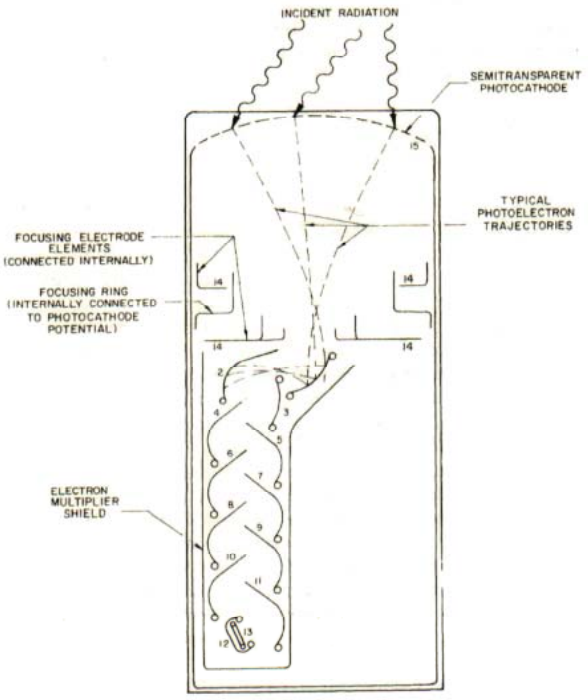

Figure 2.12

Schematic diagram of a photomultiplier tube. 1-15 are the dynodes producing the electron cascade, 13 is the anode for the final current, 14 are electrodes that help to focus the weak photoelectrons and 15 is the photocathode.[4]

The hadronic calorimeter is similar to the electromagnetic calorimeter in concept, however, the processes involved are somewhat different. Hadrons plow through many steel absorber plates and lose some of their energy in elastic collisions with the absorber's nuclei. These collisions produce more lower energy hadrons (that produce more hadrons and so on) and a hadronic shower is created. This shower is then measured by the scintillator and photomultiplier.[5]

Hadrons usually emerge from the collision in clusters or *jets*. The total energy of the jet is usually of more interest than the energy of any individual hadron. Therefore, the spatial resolution does not need to be as precise in the hadronic calorimeter as it does in the electromagnetic calorimeter.

Once the energy of the incident particle is known (from the calorimeters) and the momentum is known (from the inner detector), the mass can be determined by

$$m \cdot c^2 = \sqrt{E^2 - p^2 \cdot c^2} \qquad \text{Eq. [2.10]}$$

2.2.4 Muon Spectrometer

The muon mass is ~ 200 times larger than the electron mass and so does not lose an appreciable amount of energy in the electromagnetic calorimeter; particles with less mass will be more affected by the electromagnetic forces of the nuclei in the absorbers. Also, being leptons, muons do not interact via the strong nuclear force in the hadronic calorimeter. Therefore, they are the only particles (besides neutrinos, which go completely undetected) that can survive past the calorimeters and reach the muon spectrometer. The muon has an average lifetime of about 2 μs;[16] however, due to its high velocity and relativistic effects, it can survive well past the ATLAS detectors. The muon spectrometer is shown in figure 2.13 with respect to the rest of the ATLAS experiment.

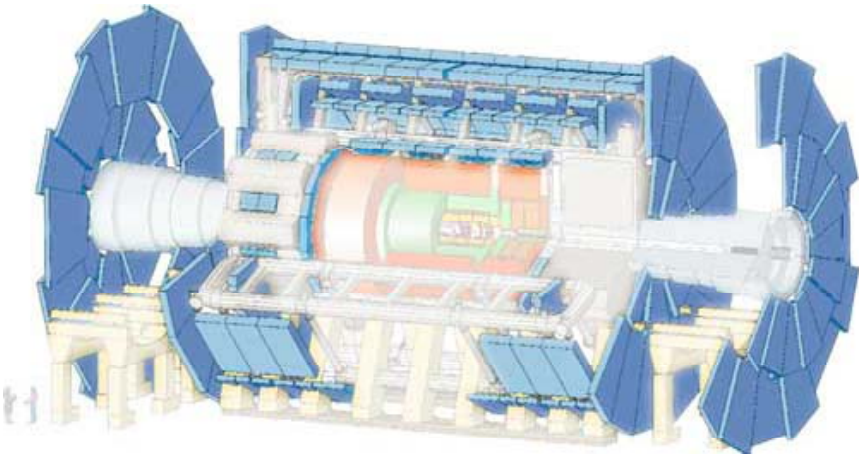

Figure 2.13
ATLAS experiment: the highlighted outer blue sections make up the muon spectrometer.[9]

The muon spectrometer is comprised of four variations of *drift chambers*. The two main types both fall under the category of *proportional chambers*: *monitored drift tubes* (MDTs) and *cathode strip chambers*. Drift chambers refer to detectors that consist of gas filled containers subjected to an electric field. When a charged particle passes through the gas, it ionizes gas atoms along its path. The ionization electrons are then attracted to the positive voltage and, as they collect on the anodes, they create current that can be measured. Proportional chambers operate at high enough voltages that the ionization electrons have enough momentum on their way to the anode that they cause secondary ionization. The operating voltages, however, are still small enough that the measured signal is still proportional to the primary ion pairs. Both the MDTs and cathode strip chambers are *precision chambers*, meaning that their main purpose is to measure the positions of particles.[10, 4]

The remaining two sub-systems of the muon spectrometer are the *resistive plate chambers* and the *thin gap chambers*. Both of these systems are referred to as *triggering*

chambers. This means that they are not designed to give precise information about the location of a particle. Their primary purpose is to affirm that a particle has passed through the detector system, thus 'triggering' the other detectors to record measurements. This lets the precision chambers know that something of interest has passed and helps to reduce background 'noise'. The arrangement of the muon spectrometer is shown in figures 2.14 and 2.15.[10]

The cathode strip chambers are proportional chambers (about 1 m x 0.5 m x 5 mm) filled with an $Ar/CO_2/CF_4$ gas mixture. The wires inside the chamber run perpendicular to the beam line. When a muon passes through the chamber it ionizes the gas atoms and the resulting liberated electrons are measured on the wires. This yields excellent spatial resolution (< 60 μm) on the axis perpendicular to the wires (i.e. which wire the muon passes is well known). In order to know *where* on the wire the muon passes, however, metallic cathode strips are lined on the inside of the chamber walls orthogonal to the wires. While the ionization electrons are attracted to the wire, the left-over ions are attracted to the strips. By measuring the current from the strips, a precise measurement of the particles position in the plane that the wires and strips create can be obtained. [10]

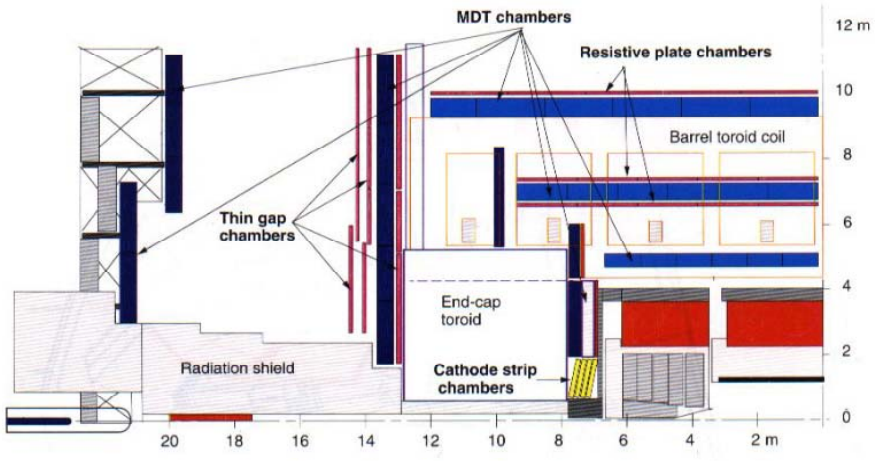

Figure 2.14

Side view of one quadrant of the muon spectrometer. The collision point is in the lower right corner. [10]

The configuration of the MDT chambers is shown in Figure 2.15. The details of the MDT system are discussed in the following section.

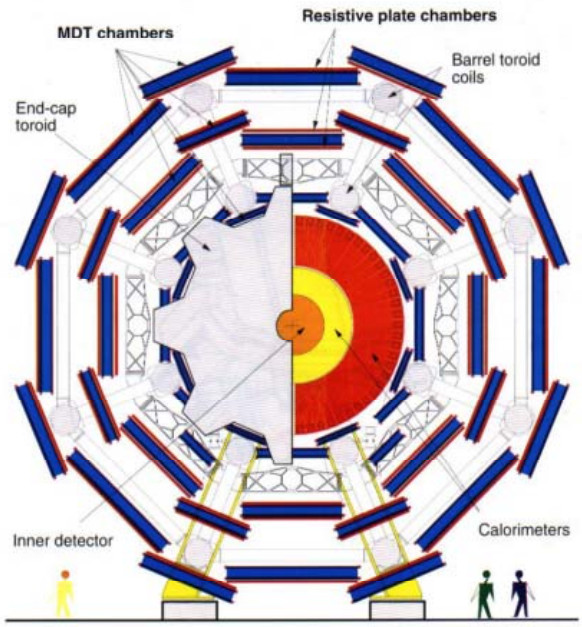

Figure 2.15

Transverse view of the muon spectrometer. [10]

3. Frascati Work

The scope of the ATLAS Muon Spectrometer is enormous. The estimated material and industrial costs for the MDT chamber systems alone was estimated in 1995 to be in excess of $20 million (US). As of 2004, there are 44 institutions worldwide officially participating in the ATLAS Muon Collaboration (excluding Hampshire College; they seem to have forgotten about us, *officially*). When the MDT chambers are installed in the ATLAS detector, they will cover a total of 5485 square meters around the collision point, nearly completely enclosing the other detector sub-systems. [10]

3.1 System Layout

Each monitored drift tube is made of aluminum, is 4.0000 ± 0.0005 m in length, and has a outer diameter of 2.9988 ± 0.0003 cm. The thickness of the aluminum tube wall is 405 ± 6 μm. Standard 50.0 ± 0.5 μm tungsten wire is fixed at a tension of 3.432 ± 0.069 N along the center of the tube. In order to increase the wire strength it is coated with a thin layer of rhenium (3% by weight) which increases the wire's rupture limit to 6.080 N, well above the tension of the wire. This coating also facilitates the handling of the wire. The potential of the wire in relation to the tube wall is set to 3270 V which creates the necessary electric field throughout the tube to maximize the linearity of the electrons drift path through the tube. [10]

The negatively charged electrons are attracted to the positive potential on the wire and the ions that are left behind are attracted to the tube wall. When the electrons begin to drift to the wire they themselves ionize more atoms along their path to the wire. This secondary ionization creates a cascade of additional electrons called the avalanche amplification. The avalanche electrons are also attracted to the wire and, therefore, result in a larger measured signal than would be expected.[4]

The incoming signal on the wire is amplified and shaped before being sent to the discriminator. The discriminator is set to a threshold that corresponds to the 22nd electron after subtracting for the avalanche amplification effect (Figure 3.1). [10]

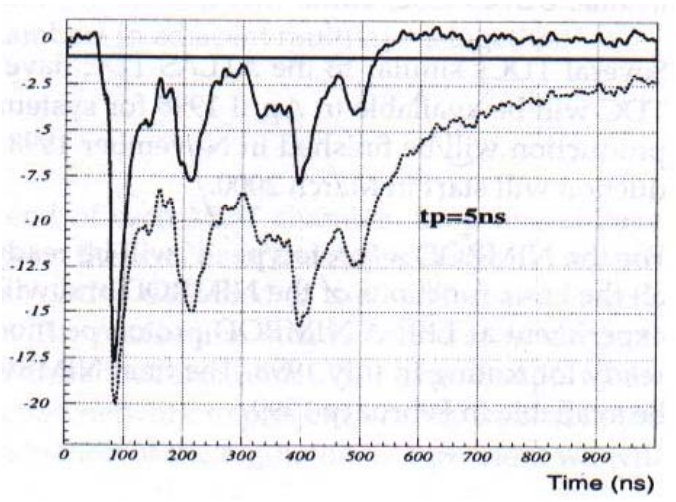

Figure 3.1

The simulated response for an energetic muon. The dashed line is the raw signal without avalanche compensation and the solid line is avalanche subtracted. The solid horizontal line indicates the discriminator threshold. The y-axis is in arbitrary units. [10]

The MDT's are constructed on flat granite tables to limit their deformities and thereafter monitored by built-in optical systems to keep track of any deformations. This is where the name *monitored* drift tube originates. [10]

The layout of the MDT chambers with respect to the test beam is shown in Figure 3.2:

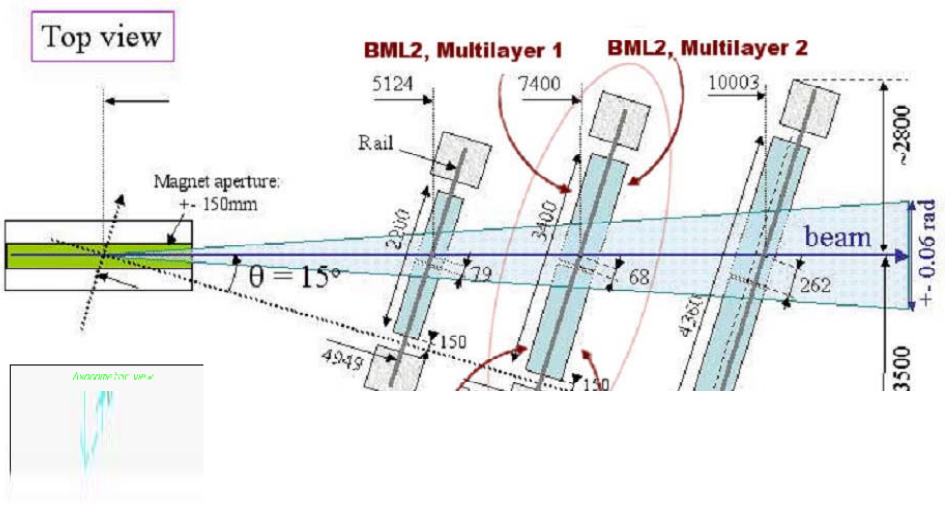

Figure 3.2

Test Beam layout at CERN: the positions for the bottom rails are given in cm. [10]

The Barrel Middle Large (BML) is the middle bank of MDTs in figure 3.2 and is divided into two stations (creatively labeled 1 and 2). Each chamber consists of 2 multi-layers separated by a support structure. Each multi-layer is made of three rows of 56 drift tubes as illustrated in Figure 3.3.

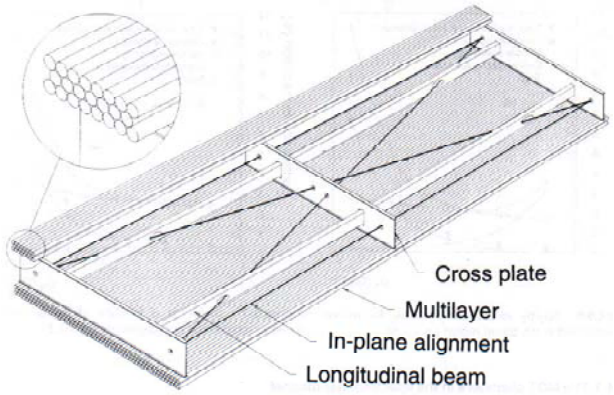

Figure 3.3
Schematic drawing of an MDT chamber. Structural components are indicated.

Within each multi-layer the tubes are closely spaced so that the height of the multi-layer has a thickness of 8.2 cm.

The MDTs of multi-layer 2 in the BML1 station use a parallel gas distribution system. Every tube of the parallel system has its own input of fresh gas, refreshing the entire gas volume once every 12 hours. Conversely, the MDTs of multi-layer 1 in the BML1 and both multi-layers in the BML2 are equipped with serial gas distribution systems designed at the Laboratori Nazionali di Frascati. This system runs the gas through three tubes in series before the gas is flushed. The gas input tubes are collectively referred to as tube number 1, the middle tubes as tube number 2, and the output tubes as tube 3. The configuration is shown in Figure 3.4. [10]

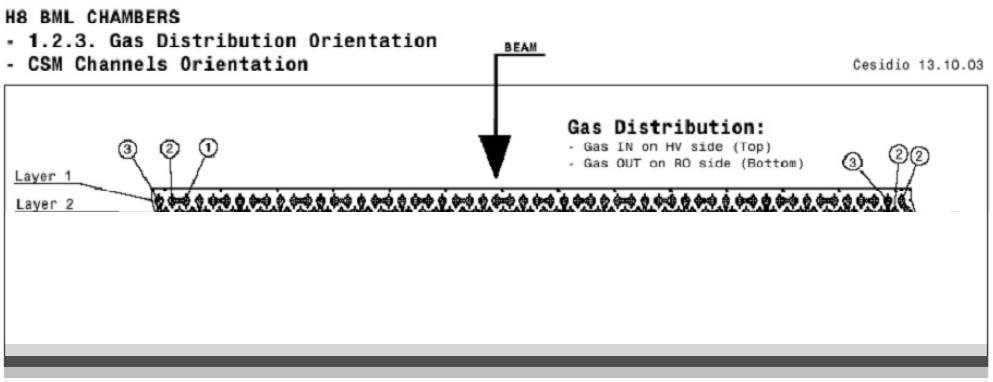

Figure 3.4

With respect to gas flow, tubes labeled 1 are input tubes, tubes labeled 2 are middle and tubes 3 are output. [10]

Each tube has an NORYL-GNF3 plastic end-plug and the three tubes are connected by inox gas jumpers or stainless steel tubes. The entire connector valve mechanism also includes a crimped aluminum cylinder and an O-ring seal (shown in figure 3.5). In the serial system the entire gas volume, through tubes 1-3, is also completely refreshed once every 12 hours. [10]

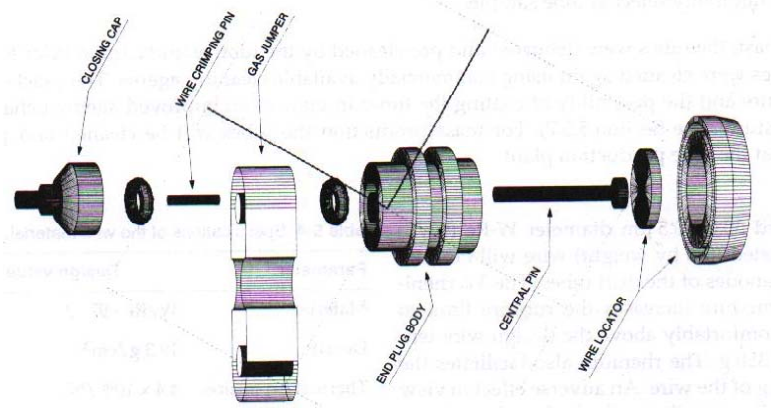

Figure 3.5

Three-dimensional schematic of the end cap design. [10]

The gas used is a mixture of Ar and CO2 (93:7). The mixture is held at a constant pressure of 3 bar and is heated to a minimum of 20°C at all times.[14]

The test beam fires clusters of muons with momentum ranging between 20 and 400 GeV/c toward the MDT chambers. A scintillator hodoscope trigger is located between the beam aperture and the MDT's. Using both fast detectors and

scintillators with very short output pulses, a hodoscope is a combination of multiple detector elements that determines when a particle passes through it. In the 2003 muon test beam a hodoscope is used to trigger when the muons are coming. [14]

Whenever a muon passes through the hodoscope, the hodoscope sends a signal instructing the MDT electronics to expect a signal from the MDT chambers. The MDT electronics are always monitoring the current coming from the MDT wires and storing its level in a memory buffer. The data is only stored in long-term memory whenever the hodoscope sends a signal, indicating that a muon is headed for the MDT chambers. The purpose of the triggering is to decrease the number of random signals from cosmic rays or other accidental sources. [10]

3.2 Objective

This analysis compared the drift times of MDT's with parallel gas distribution systems to those with serial gas distribution systems. The 2003 test beam data collected from the chambers located in the BML (Barrel Middle Large) were examined. Previous unpublished results suggested that the drift time of the ionized electrons in the MDTs varied depending on whether the tube was a gas input tube, a middle tube, or a gas output tube. This analysis was initiated in order to discover whether there is a dependence and, if a dependence exists, to understand the cause.

3.3 Methods of Analysis

One run of data generally last around ten minutes and normally yields between $2x10^5$ and $3x10^5$ events, resulting in about triple this amount of drift times per multi-layer (since there are three layers of tubes per multi-layer), depending on the run. These drift times can be plotted in histograms as shown in Figure 3.6.

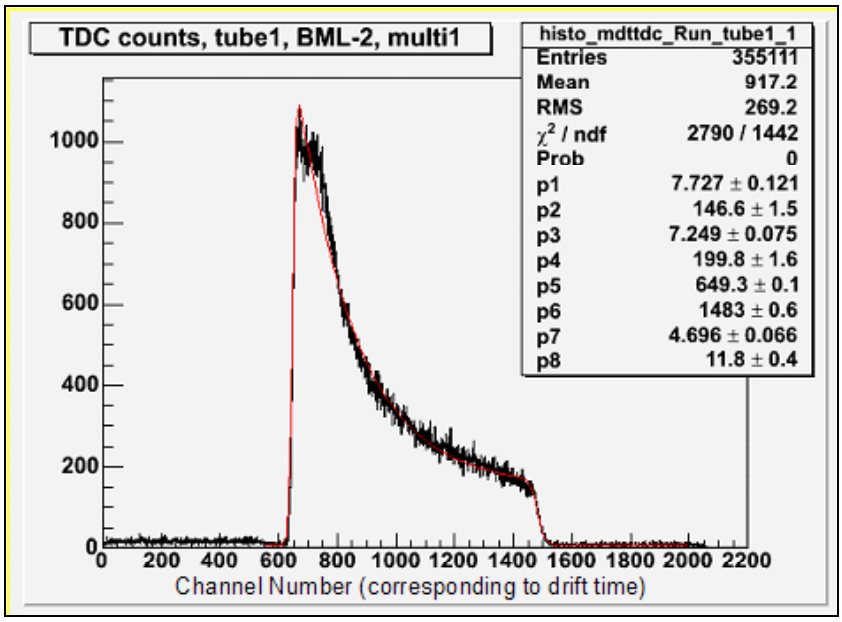

Figure 3.6

Histogram of raw MDT time-to-digital-converter counts. The red line is a fit of the data with equation 3.1.

The histograms of the spectra of drift times span a 700 ns range and can be fit with the following function:

$$f(x) = p1 + p2 \cdot \frac{\left[1 + p3 \cdot e^{\frac{-(x-p5)}{p4}}\right]}{\left[1 + e^{\frac{-(x-p5)}{p7}}\right]\left[1 + e^{\frac{(x-p6)}{p8}}\right]} \qquad \text{Eq. [3.1]}$$

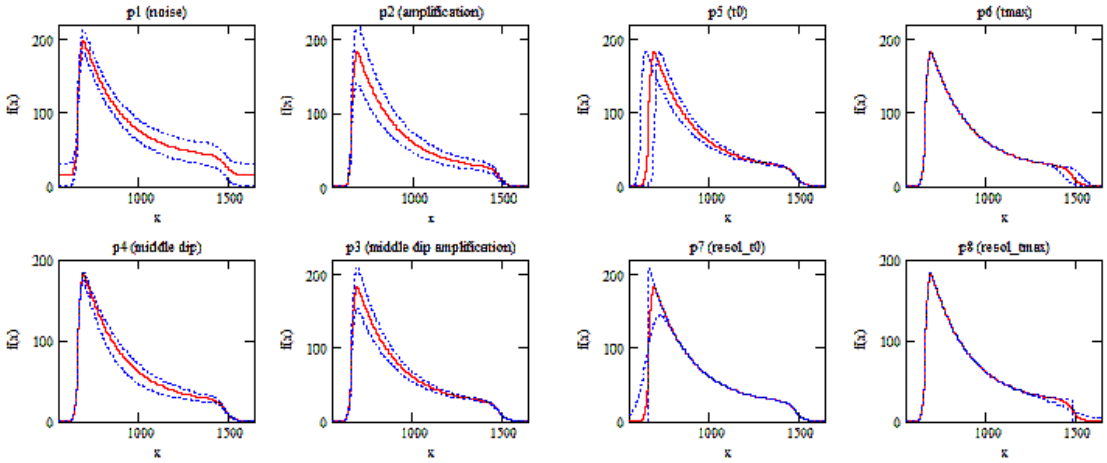

Figure 3.7

The blue lines indicate what changing the parameter does to the red-lined function.

Equation 3.1 is a combination of three Fermi-Dirac distributions and its parameters p5 and p6 describe the edges of the spectra, while p7 and p8 represent the resolution of the edges. Parameter p5, the minimum edge of the spectra, is referred to as 't_0' and p6, the maximum edge of the spectra, as 't_{max}'. The quantity t_{drift} is defined as the maximum measured time that it takes an ionized electron to drift to the wire and is found by taking the difference between t_0 and t_{max} in any given spectra. Each parameter's specific influence on the fit is shown in Figure 3.7. While this shape may seem peculiar, it is the natural result from a distribution of average drift velocities that is non-linear with respect to starting radius (Figure 3.8). The starting radius is the distance from the wire at which any given electron is separated from its atom.

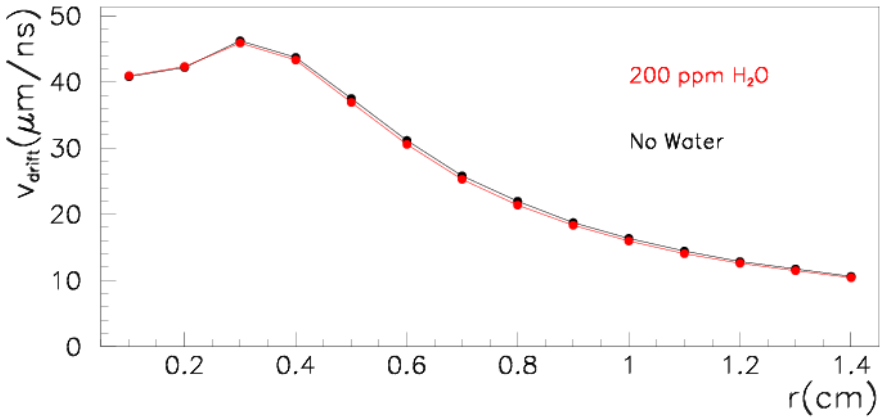

Figure 3.8

GARFIELD simulation of ionization electron drift velocities as a function of starting radius within the MDT for tubes with and without water contamination.

Integrating the distribution over time leads to a correlation between the starting radius of the ionization electron and the actual drift time (Figure 3.9). A histogram of the projection of the drift times (y-axis) leads to the shape that is found for the drift time spectra.

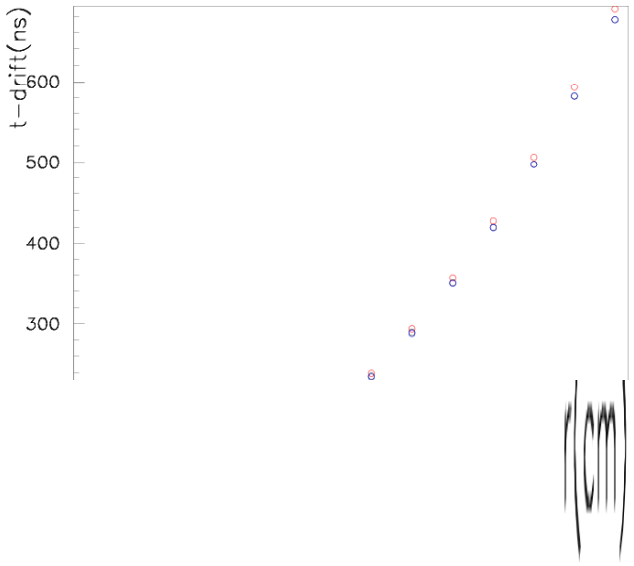

Figure 3.9
GARFIELD simulation of drift time as a function of starting radius. Blue points are 'no water', red points are with 200 ppm water contamination.

Due to the complex electronics involved in the triggering system, the trigger signal from the hodoscope can potentially actually arrive *after* (or even sometime before) the muons have passed through the drift tube. Therefore, the drift time cannot be known as it would be measured in real-time, only in relative time with respect to the trigger. This is acceptable since, in this analysis, the only quantity of interest is t_{drift}, the longest possible time that it might take an electron to reach the wire; t_{drift} is only dependant on the relative difference between t_0 and t_{max}.

In order to analyze the test beam data a framework was written in C^{++} using the CERN application ROOT as a foundation (Appendix 1). This framework is divided into four distinct macros or scripts.

3.3.1 Macro 1 (*fill_ntup_allRuns.C)*

First the raw data trees had to be converted from PAW (Physics Analysis Workstation) files to ROOT. This data is initially sorted by run number. Macro 1 takes the created ROOT data trees and first groups all of the data by multi-layer and tube number. The drift times are represented by channels of the time-to-digital-converter (tdc) in the original data trees. Each channel represents 25/32 ns. This conversion arises from the 11-bit system used in the electronics leading to a total of 2^{11} or 2048 channels. Since the full measurable time range of the particular TDCs that are used is 1600 ns, this requires a conversion of

$$\frac{1600}{2048}\frac{\text{ns}}{\text{channel}} = \frac{25}{32}\frac{\text{ns}}{\text{channel}} = 0.78125\frac{\text{ns}}{\text{channel}} \qquad \text{Eq. [3.2]}$$

This conversion from channel to time is accounted for in Macros 2,3,4.

After Macro 1 groups the mdttdc channel counts by multi-layer and tube number, the histograms are plotted and fit with equation 3.1. An ntuple is created for each run that contains the following variables:

1) Run number
2) Multi-layer ID
3) Tube number
4) Number of entries
5) p5 (t_0)
6) p6 (t_{max})
7) p7 (resolution of t_0)
8) p8 (resolution of t_{max})
9) error of p5
10) error of p6
11) error of p7
12) error of p8
13) χ^2 value of fit
14) tdrift
15) error of tdrift

After being filled, each ntuple for each run is saved in a file to be accessed by Macros 2, 3 and 4. This analysis results in a total of 144 t_{drift} values, 12 for each run.

3.3.2 Macro 2 (*ana_mean_allRuns.C*), Macro 3 (*ana_tdrift_vs_time.C*), Macro 4 (*ana_subtract_tdrift_vs_time.C*)

Macros 2, 3 and 4 all share a common calibration section. First, only data from runs where the signal to noise ratio was greater than 15 was used. The drift time values from the mdttdc count fits are stored in the ntuples from Macro 1 in units of channel number instead of ns. This conversion is shown in equation 3.2. In addition to this channel-to-time calibration, there is a temperature calibration. While the temperature of the drift tubes is kept above 20°C, it is not stringently controlled beyond this. In fact, the temperature of the system ranges between 23°C and 30°C for all of the analyzed runs; however, the duration of the runs are not long enough for the temperature to change significantly throughout any given run. In the 2001 test beam, G Avolio et al found that the drift time varies with the temperature of the system. Using 27°C as the reference temperature, t_{drift} varies as

$$\Delta t_{drift} = \frac{2.4\text{ns}}{^\circ\text{C}} \cdot (\text{T} - 27^\circ\text{C}) \qquad \text{Eq. [3.3]}$$

Taking both of these calibrations into account, the t_{drift}s can be further analyzed. [15]

Macro 2 plots t_{drift} vs tube number for each run and multi-layer and prints these values to the screen. The plots from Run 1226 are shown in Figure 3.10 as an example. There is a clear dependence of t_{drift} on tube number for the multi-layers with serial gas distribution systems. The parallel system shows no such distinguishable dependence. This clearly indicates that the conditions inside of the MDTs are changing from one tube to the next in the serial system and not in the parallel system.

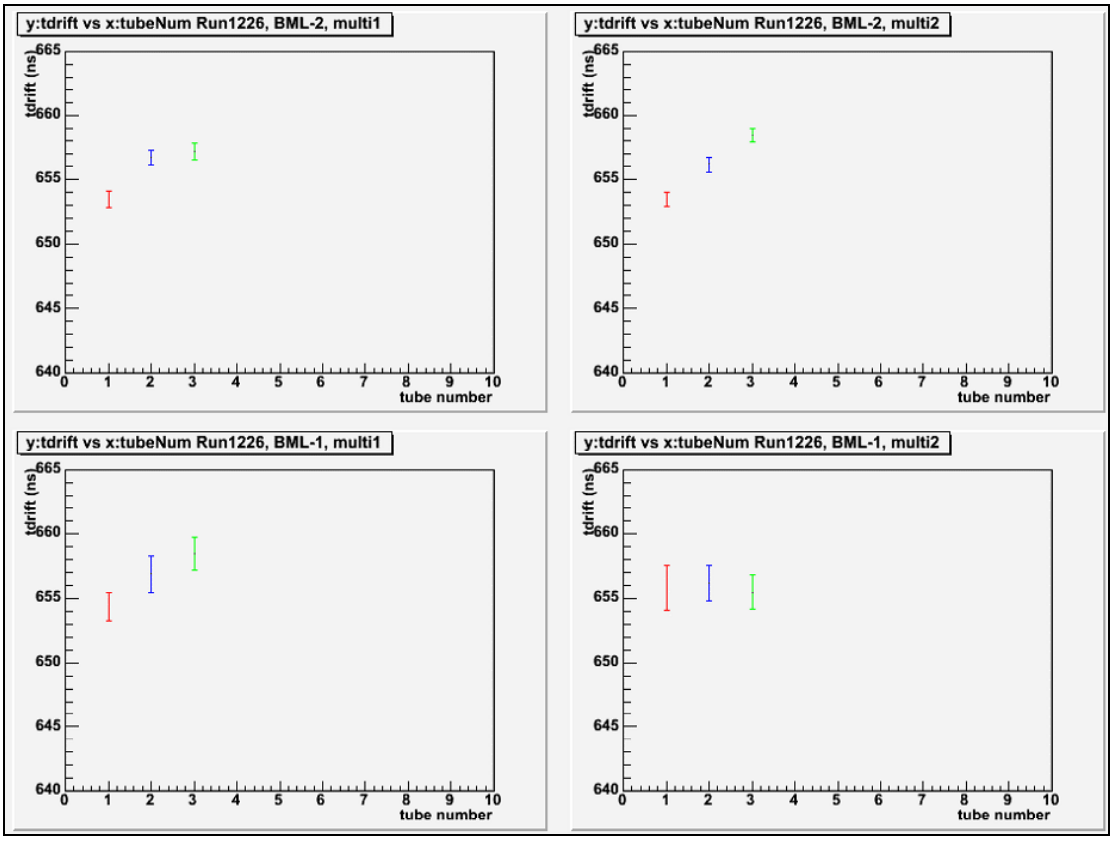

Figure 3.10

The experimental t_{drift} as a function of tube number. The lower right plot is the parallel gas distribution system, the remaining three plots are serial.

Each run number is associated with a date and time (Table 3.1).

Run	nEntries	temp (bml1)	temp (bml2)	start time	stop time	hour	date
700032	120k	23.81	24.17	7:46	8:18	4	19/7/2003
700090	160k	24.86	25.23	9:38	10:04	30	20/7/2003
700108	100k	26.88	26.77	13:55	14:08	34	20/7/2003
700172	120k	26.92	26.9	3:20	3:58	47	21/7/2003
1091	200k	27.6	27.42	0:07	0:56	428	6/8/2003
1222	280k	28.83	28.65	18:43	19:34	495	8/8/2003
1226	280k	28.18	28.06	22:41	22:37	499	8/8/2003
1235	200k	25.33	25.72	7:28	8:11	508	9/8/2003
1256	280k	26.07	26.37	10:44	11:38	535	10/8/2003
1374	240k	25.31	25.57	3:58	5:02	672	16/8/2003
1431	280k	26.02	26.17	1:30	2:36	694	17/8/2003
1559	280k	26.76	26.54	21:20	???	833	22/8/2003

Table 3.1

Therefore, a global time can be applied. Treating 4:00 am on July 19, 2003 as reference time, each run has an associated hour enabling a plot of t_{drift} as a function of time. A concern was that t_{drift} or the change in t_{drift} (Δt_{drift}) between each tube was varying with time. Macro 3 calibrates the data and then plots t_{drift} for each type of tube as a function of time separately for each multilayer (example shown in Figure 3.11).

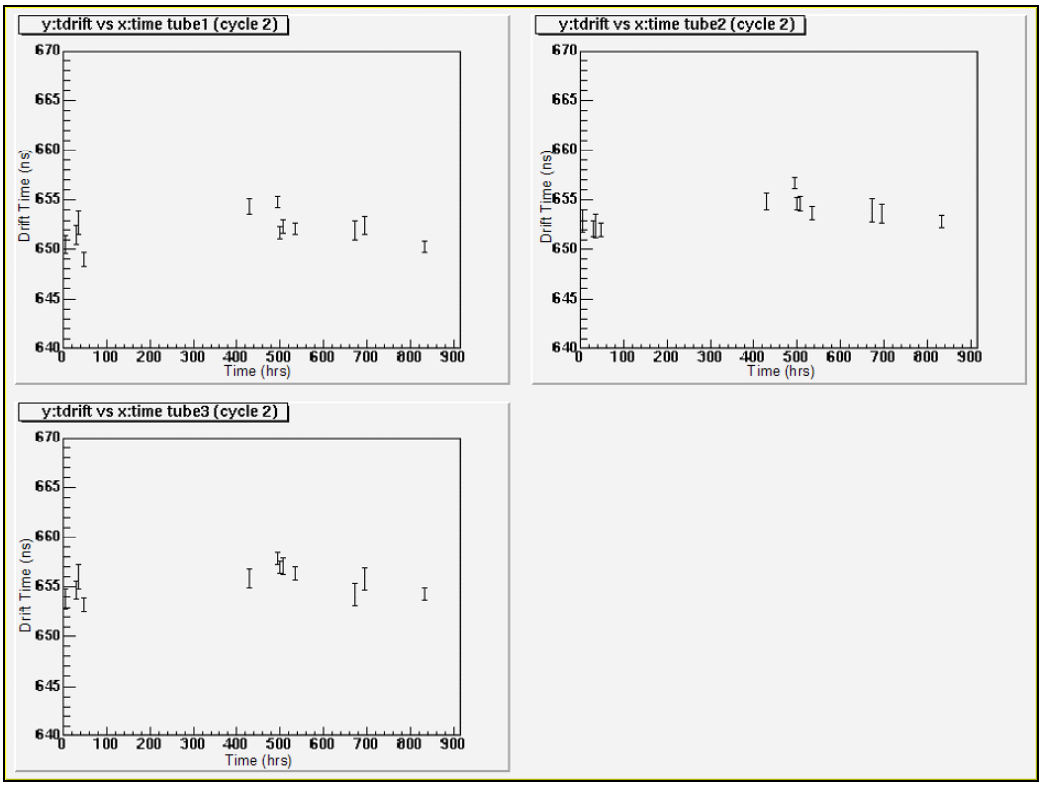

Figure 3.11

The experimental t_{drift} plotted as a function of time. Each tube number and multi-layer was plotted separately.

The data was fit to both a flat line (zero slope) and a 1st order polynomial (a sloped line). There was no appreciable difference in the error of the fit. The ratio of the χ^2 value divided by the number of degrees (NDF) of freedom of a 1st degree polynomial to a flat line, summed over all the multi-layers and tubes and averaged is essentially one:

$$\frac{\sum_{\text{multilayers}} \sum_{\text{tubes}} \left(\frac{\chi^2_{\text{pol\_1}}}{\text{NDF}_{\text{pol\_1}}} \cdot \frac{\text{NDF}_{\text{pol\_0}}}{\chi^2_{\text{pol\_0}}} \right)}{12} = 0.9 \qquad \text{Eq. [3.4]}$$

Clearly, t_{drift} does not appear to change over time.

Macro 4 does a similar analysis. This macro looks at Δt_{drift} from one tube to the next as a function of time for each multi-layer and each type of tube and plots them (example shown in Figure 3.12).

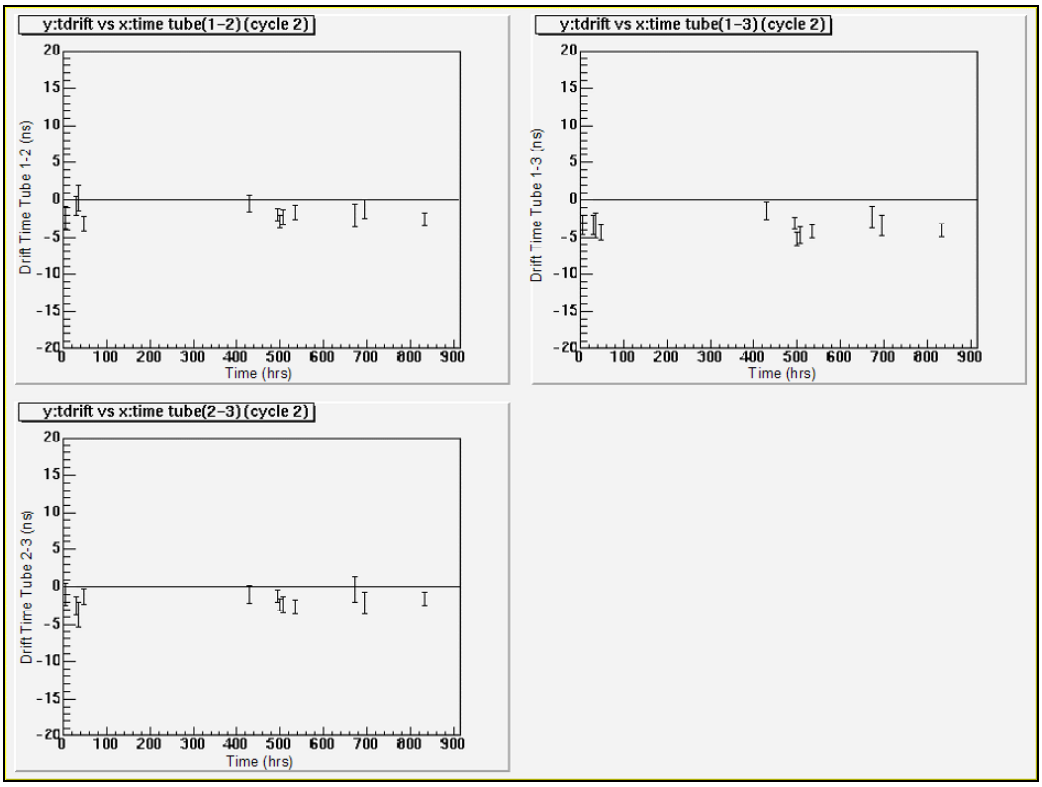

Figure 3.12

Experimental t_{drift} plotted as a function of time for each mutli-layer and each tube. The black horizontal lines are simply to reference the zero line.

Again, after fitting these plots, Δt_{drift} does not appear to vary over time. These plots do reveal, however, an average Δt_{drift} of the serial system between tubes 1 and 2 as well as between tubes 2 and 3 that is very markedly different than the Δt_{drift}s of the parallel system. The Δt_{drift} values between any two consecutive tubes (Δt_{drift} of tubes 2-1 and Δt_{drift} of tubes 3-2) are then averaged. The average Δt_{drift} of the serial system was found to be 1.90 ± 0.22 ns/tube while for the parallel system it was 0.82 ± 0.79 ns/tube. (final_data.xls – sheet5)

3.4 Hypothesis

One might naïvely expect that in the serial system the gas will become increasingly more polluted as it passes through each tube, thereby affecting the measured drift time. In both the serial and parallel systems, the entire gas volume is completely refreshed once every 12 hours. This is well within the gas purity limits which dictate that the gas be refreshed once every 24 hours.

The MDTs are identical and the gas entering the system is the same in both systems. From a gas particle's point of view, the only difference between the systems is

traveling thought the end-plug/valve mechanism that connect one tube to the next. Therefore, in order to explain the discrepancy of the serial system, it was proposed that water may be leaking into the gas through the plastic end-plugs. Since, as simulations show, a more humid gas medium would result in the electron having a lower drift velocity, this could potentially account for the drift time discrepancy.

This hypothesis has a few obvious ramifications. First, since the additional water is only leaking in between tubes, the humidity levels would be constant in any given tube. The gas flows at a steady rate so that the more humid gas would be unable to propagate backwards through the serial system. Soon after the gas starts to flow, the humidity in every part of the system will reach some equilibrium, leading to a constant t_{drift} within each tube. Second, the difference in humidity levels should be the same between tubes 1 and 2 as it is between tubes 2 and 3. This would result in $\Delta t_{drift}(2\text{-}1)$ also being the same as $\Delta t_{drift}(3\text{-}2)$. Moreover, this difference should be constant over time.

In order to examine this hypothesis, Monte Carlo (random) muon events were simulated using two computer applications, ROOT and GARFIELD, focusing specifically on the drifting electrons within the tubes. Preliminary simulations in coordination with experimental data should lead to a correction factor for the humidity levels. Subsequent, more detailed simulations can then predict actual mdttdc count spectra which can be analyzed with the ROOT framework to compare with the experimental results.

Unfortunately, this is the only feasible way to examine this hypothesis at this time. The drift chambers are already built and, therefore, cannot be easily retrofitted with humidity sensors. In addition, the amount of water in question is very small (< 100 ppm). Extracting the used gas from the system without introducing some additional contamination would be extremely difficult. This resulting contamination could easily spoil any subsequent humidity measurements. Both of these procedures would also be very costly.

3.5 Simulations

GARFIELD is program that simulates drift chamber mechanics and can calculate things like electron and ion drift lines, average drift velocities, electron avalanche amplifications, and field maps within the chambers among many other things [11]. With GARFIELD, Fabio Cerrutti (a LNF employee stationed at CERN) calculated average drift velocities with respect to different starting radii within the tube (Figure 3.13) for gas with no water as well as a humidity level of 200 ppm.

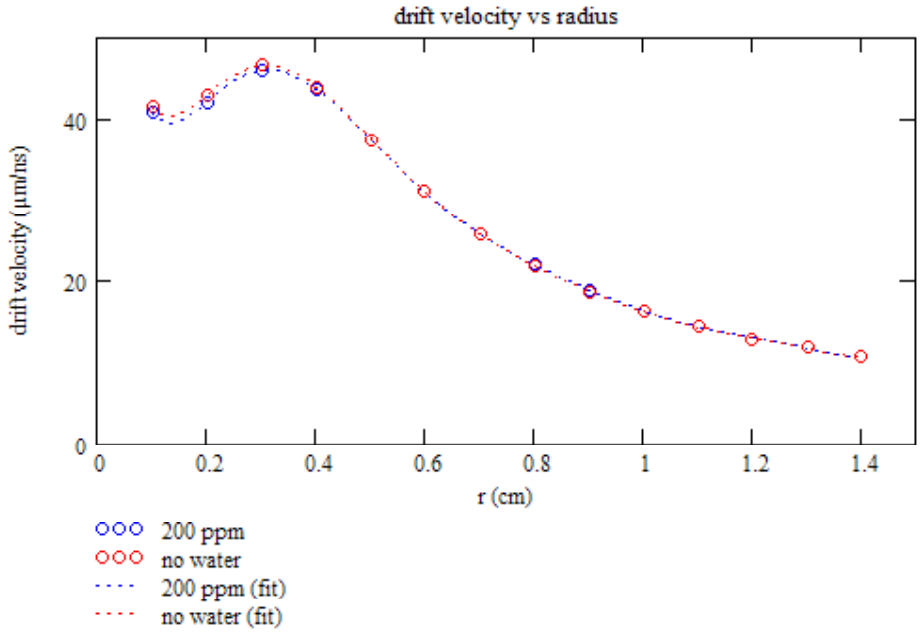

Figure 3.13

GARFIELD simulations of ionization electron drift velocity as a function of starting radius. Polynomial fits are also shown.

Integrating this graph reveals a relation between drift time and starting radius (Figure 3.14).

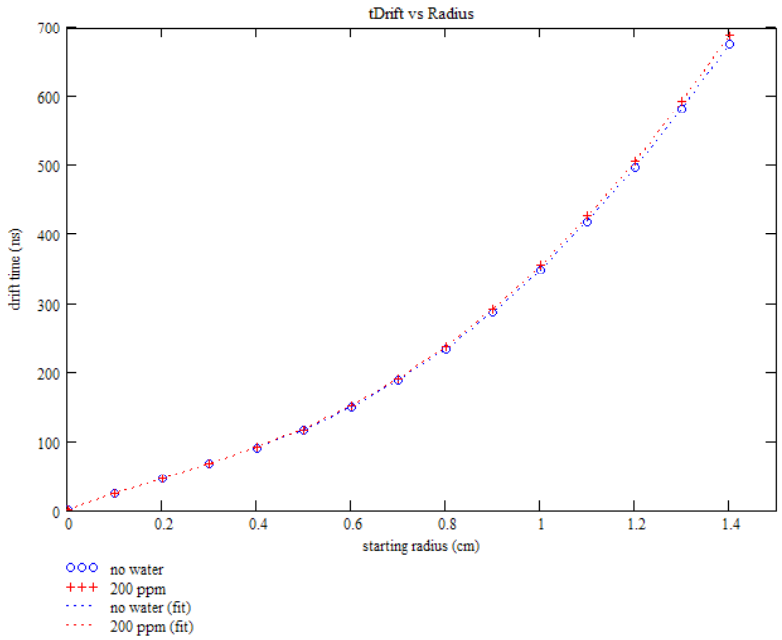

Figure 3.14

GARFIELD simulations of ionization electron drift times as a function of starting radius, 4th degree polynomial fits are shown.

This relation was fit phenomenologically with Mathcad with a 4th degree polynomial (anything less than a 4th degree revealed lousy residuals, Figure 3.15). The resulting function supplied a mechanism to translate any given starting radius to a drift time.

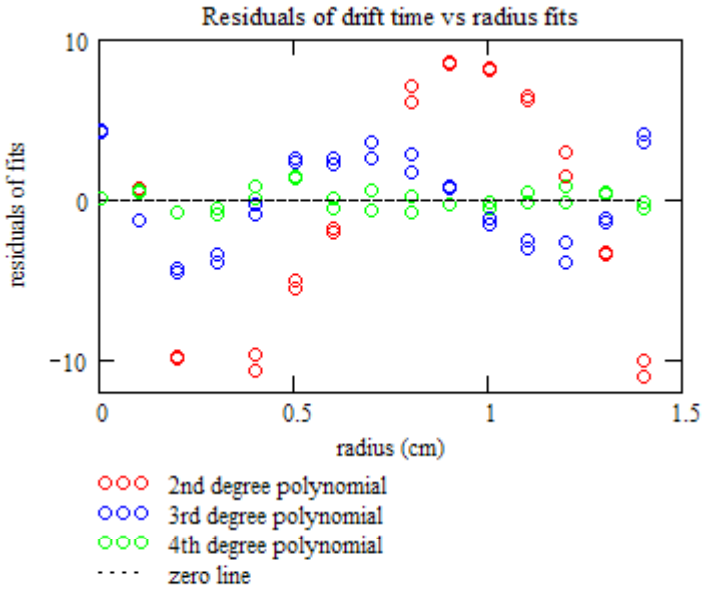

Figure 3.15

Residual plots of the 2nd, 3rd, and 4th degree polynomial fits of ionization electron drift time vs starting radius.

In order to simulate the proper distributions of electrons and propagate these distributions to mdttdc count spectra, a series of macros were written in ROOT (Appendix B).

In the ROOT simulations, intuition would indicate that the proper distribution of electrons that would correspond to a uniform distribution of muons is a circular or possibly even a cylindrical distribution since this is the shape of the MDT. However, this is a naïve misconception. Even thought there are many ionization electrons left in the wake of the muon, there is only one drift time per muon track measured in any given tube. This is due to the front-end electronics of the system. As the electrons gather on the wire, they create a measurable current. The discriminator only records a drift time when the current reaches some threshold value. Figure 3.16 shows the measured current as a function of time for a simulated track. The drift time is only measured when current begins which corresponds to when the first (or very nearly first) electron hits the wire.

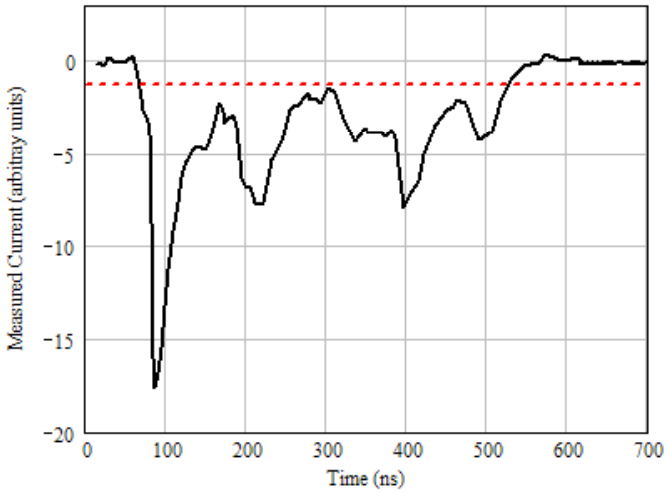

Figure 3.16

Simulation of measured current as a function of time for a muon track.
The avalanche effect has been subtracted.

The muon that leaves the track is moving at nearly the speed of light and, therefore, the wake of electrons appears, for our simulation purposes, instantly. Therefore, the electrons that will reach the wire first are the ones with the shortest distance to travel. The leading electrons will always lie on the line which is perpendicular to both the muon's path and to the length of the tube (Figure 3.17). This means that the proper distribution of electrons to simulate is a *linear* one along this line.

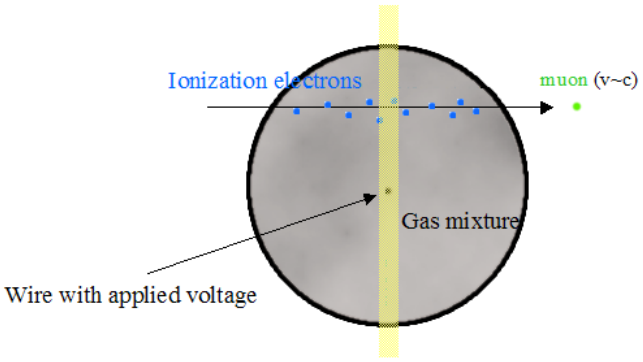

Figure 3.17

Cross section of monitored drift tube: the electrons distributed along the yellow line will always have the shortest distance to travel to the wire, thus they are the ones to simulate.

A random 'jiggle' was added to each simulated drift time that was based on a linear extrapolation of the resolutions of t_0 and t_{max} found from the experimental data. In addition, an appropriate amount of random white noise was also added to the spectra.

The fitted function from Figure 3.14 above was used to convert this linear distribution of electrons into drift times. Since the GARFIELD data only had data for no water and 200 ppm, it was necessary to develop a technique to simulate any amount of water. The differences in the two fit functions (no water and 200 ppm) are relatively small. Therefore, assuming a linear parameter dependence between the 'no water' fit and the 200 ppm fit, the parameters for any amount of water less then 200 ppm could be derived.

Assuming a linear dependence of Δt_{drift} on humidity level, Figure 3.14 leads to a correction factor of 6.45 ns/100ppm. The experimental Δt_{drift} was found to be 1.90 ± 0.22 ns/tube corresponding to a humidity level of 29.87 ± 1.70 ppm/end-plug leaking into the system between tubes.

ROOT simulations of the mdttdc spectra were run from zero to 180 ppm in 30 ppm increments, primarily to verify the linear correlation between Δt_{drift} and humidity level. The results are shown in Figure 3.18 and do indeed confirm a linear dependence.

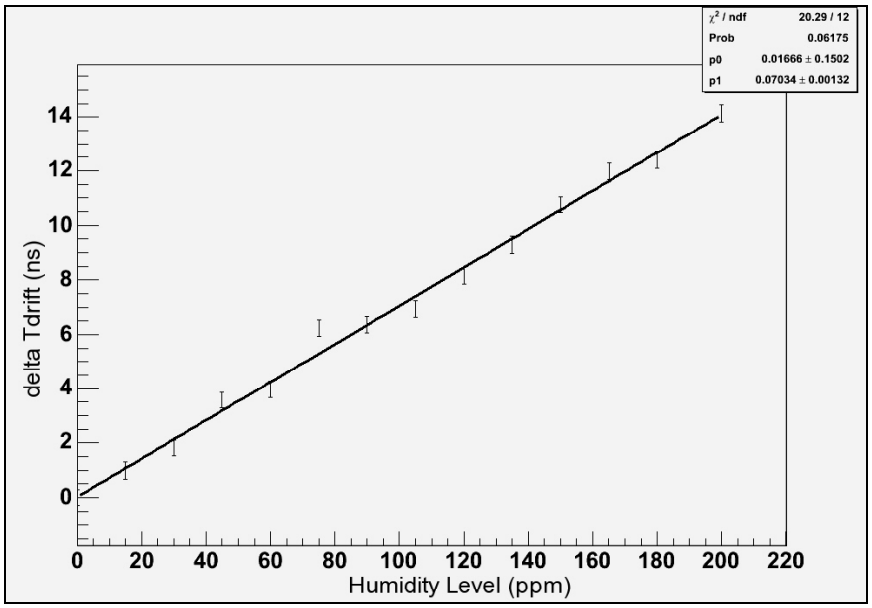

Figure 3.18

Monte Carlo simulations of Δt_{drift} for various amounts of water ranging from 0 ppm to 200 ppm, illustrating a linear dependence.

Finally, in order to explore a simulated humidity increase of 29.87 ± 1.70 ppm/end-plug, the ROOT simulation was run at 0, 30 and 60 ppm. The analysis from macros 2, 3 and 4 was then performed to on these data. These simulations predict a Δt_{drift}

of 2.10 ± 0.09 ns/tube, in perfect agreement with the experimental value of 1.90 ± 0.22 ns/tube.

3.6 Conclusions

A systematic analysis of the serial gas distribution system revealed that t_{drift} is constant with respect to time for any particular type of tube (input, middle,output) and, therefore, Δt_{drift} is also constant between any two consecutive tubes. In addition, Δt_{drift}(tubes 1-2) and Δt_{drift}(tubes 2-3) are equal indicating that there is something happening between every tube that is affecting the electron drift time. Specifically, something is increasing t_{drift} by 1.90 ± 0.22 ns/tube.

According to equation 3.5, the experimentally measured value of Δt_{drift} corresponds to a humidity influx leakage rate or water flow (WF_{exp}) of 0.221 ± 0.013 (mbar/day)/tube.

$$WF_{exp} = \frac{P \cdot w \cdot V}{t_{refresh}} \qquad \text{Eq. [3.5]}$$

where P is the air pressure inside the tube, w is the amount of water in ppm, V is the volume of one tube, and $t_{refresh}$ is the time to refresh the total volume of gas.

A theoretical water flow (WF_{theory}) was computed using the permeability constant of NORYL-GFN3 of

$$P_W = 2.6 cm^3 \cdot \left(\frac{cm}{cm^2 \cdot day} \right) \qquad \text{at } 25°C,^{15}$$

the difference in partial pressures inside and outside the tube (ΔP), the surface of the end-plug exposed to the gas (S) and the average end-plug length between tubes (L):

$$WF_{theory} = P_W \cdot \frac{S}{L} \cdot \Delta P \qquad \text{Eq. [3.6]}$$

Assuming an outside humidity of 60% and using the average temperature of all the runs (25°C), WF_{theory} was estimated to be ~ 0.194 (mbar/day)/end-plug. This is in close agreement with WF_{exp} of 0.221 ± 0.013 (mbar/day)/tube.

Therefore, the steady increase in t_{drift} in consecutive tubes can be quantitatively explained by water contamination through the end-plugs; there is a build up of water vapor in the gas mixture during the its flow through the series of tubes. This hypothesis is further supported by the water conduction rate of the NORYL-GFN3 end-plugs.

Surprisingly, the Δt_{drift}/tube for the parallel system was 0.82 ± 0.79 ns/tube, not exactly zero. In theory, these tubes should all record the same drift times. However, while one standard deviation is slightly higher than zero (~ 0.03 ns/tube), two deviations

are clearly in agreement with zero. In addition, the statistics in BML1, multi-layer 2 were not as favorable as the other multi-layers due to its configuration with respect to the test beam line (refer to Figure 3.2). BML1, multi-layer 2 recorded, on average about ¼ less events than the either of the multi-layers from BML2. While this could have adversely affected the results from the parallel system, there are still enough recorded events to claim a reasonable level of accuracy.

4. Appendices

4.1 Appendix A

4.1.1 Macro 1

```
//  This macro groups the raw data by multi-layer and tube number,
//  plots the histos (12 per run) and fits the data to a combination
//  of three Fermi-Dirac Equations.  It then stores all of the
//  relevant data in an ntuple for later use (by the ana-prefixed macros).

#include <sstream>
#include <iostream>
#include <string>
#include "MyClass_ntup.h"
#include "TH1.h"
#include "TFile.h"
#include "TChain.h"
#include "TTree.h"
#include "TDirectory.h"
#include "TNtuple.h"
#include "TF1.h"
#include "TCanvas.h"

const int nWires = 56;
TH1F* histo_mdttdc_wire[nWires];
TH1F* histo_mdttdc_tube[3][4];
TNtuple *ntup_RunData;
TCanvas* canv[12][3];
TPad* pad_canv[12][3][4];

double myfunc(double *x, double *p){
  return p[0]+((p[1]*(1+p[2]*exp(-(x[0]-p[4])/p[3])))/
              ((1+exp((-x[0]+p[4])/p[6]))*(1+exp((x[0]-p[5])/p[7]))));
}

void fill_ntuple_allRuns()
{
  TArrayF array_RunNumber = TArrayF(12);
  array_RunNumber[0] = 700032;
  array_RunNumber[1] = 700090;
  array_RunNumber[2] = 700108;
  array_RunNumber[3] = 700172;
  array_RunNumber[4] = 1091;
  array_RunNumber[5] = 1222;
  array_RunNumber[6] = 1226;
  array_RunNumber[7] = 1235;
  array_RunNumber[8] = 1256;
  array_RunNumber[9] = 1374;
  array_RunNumber[10] = 1431;
  array_RunNumber[11] = 1559;

  // Defining cuts.....
  int cut_station = 2;
  int RunNumber, wireCount, cycleCount = 0;

  ntup_RunData = new TNtuple("ntup_RunData","Ntuple of Run
Data","station:eta:multi:nEntries:tube:t0:tmax:resol_t0:resol_tmax:err_t0:err_tmax:err_re
sol_t0:err_resol_tmax:chiSq",10000);

  //  Defining the fitting function and setting the initial parameters....
  TF1 *funct_fit = new TF1("funct_fit", myfunc, 0, 2200, 8);
  funct_fit->SetParName(0,"p1");
  funct_fit->SetParName(1,"p2");
  funct_fit->SetParName(2,"p3");
  funct_fit->SetParName(3,"p4");
  funct_fit->SetParName(4,"p5");
  funct_fit->SetParName(5,"p6");
```

```
  funct_fit->SetParName(6,"p7");
  funct_fit->SetParName(7,"p8");
  funct_fit->SetParameter(0, 0.22);
  funct_fit->SetParameter(1, 22);
  funct_fit->SetParameter(2, 8.5);
  funct_fit->SetParameter(3, 214);
  funct_fit->SetParameter(4, 656);
  funct_fit->SetParameter(5, 1480);
  funct_fit->SetParameter(6, 7.4);
  funct_fit->SetParameter(7, 22.4);

  funct_fit->SetParLimits( 0,  0,    500);// starts at 0.22
  funct_fit->SetParLimits( 1,  0.002,22000);// 22
  funct_fit->SetParLimits( 2,  0.085,8500);// 8.5
  funct_fit->SetParLimits( 3,  0.02, 20000);// 214
  funct_fit->SetParLimits( 4,  550,  800);// 656
  funct_fit->SetParLimits( 5,  1380, 1700);// 1480
  funct_fit->SetParLimits( 6,  0.5,  30);// 7.4
  funct_fit->SetParLimits( 7,  5,    50);// 22.4

  //  Declaring the histo for mdttdc for each wire.....
  for(int i=0; i<nWires; i++){
      ostringstream ost_histo_mdttdc_wire;
      ost_histo_mdttdc_wire<<"histo_mdttdc_wire"<<i+1;
      histo_mdttdc_wire[i] = new TH1F(ost_histo_mdttdc_wire.str().c_str(),
ost_histo_mdttdc_wire.str().c_str(),2200,0,2200);
      histo_mdttdc_wire[i]->SetDirectory(0);
  }

  //  Declaring the histo for mdttdc for each file and tube.....
  for(int i=0; i<4; i++){
    for(int j=0; j<3; j++){
      char title_for_tube_histos[60];
      sprintf(title_for_tube_histos,"TDC counts, tube%d, BML%d, multi%d",j+1, int(-2 +
(i>>1)), 1 + (i&1) );
      ostringstream ost_histo_mdttdc_tube1_;
      ost_histo_mdttdc_tube1_<<"histo_mdttdc_Run_tube"<<j+1<<"_"<<i+1;
      histo_mdttdc_tube[j][i] = new TH1F(ost_histo_mdttdc_tube1_.str().c_str(),
title_for_tube_histos,2200,0,2200);
      //histo_mdttdc_tube[j][i]->SetTitle(title_for_tube_histos";TDC counts");
      histo_mdttdc_tube[j][i]->SetDirectory(0);
    }
  }

  for(int CountRunNumber = 0; CountRunNumber<=11; CountRunNumber++){
    RunNumber = int(array_RunNumber[CountRunNumber]);
    ntup_RunData->Reset();
    cout << endl << "RunNumber: " << RunNumber <<endl;

  for(int i=0; i<4; i++)
    for(int j=0; j<3; j++){
      histo_mdttdc_tube[j][i]->Reset();
    }

    //  Creating the chain of data (for now we're only doing one file at a time...
    TChain *chain_data = new TChain("data");
    int maxFilePiece = 0;
    char filename_raw[80];
    if(RunNumber==1758)                                                    maxFilePiece =
2;
    if((RunNumber==700032) || (RunNumber==700108))                         maxFilePiece =
3;
    if((RunNumber==1764)   || (RunNumber==700090))                         maxFilePiece =
4;
    if((RunNumber==1091)   || (RunNumber==1235))                           maxFilePiece =
5;
    if((RunNumber==1220)   || (RunNumber==1223)   || (RunNumber==1227) ||
       (RunNumber==1270)   || (RunNumber==1374))                           maxFilePiece =
6;
    if((RunNumber==1222)   || (RunNumber==1226)   || (RunNumber==1256) ||
       (RunNumber==1263)   || (RunNumber==1431)   || (RunNumber==1559) ||
```

```
      (RunNumber==700172))                                                maxFilePiece =
7;
    if(maxFilePiece==0){
      cout << "This file doesn't exist..." << endl;
    }
    for(int filePiece = 1; filePiece <= maxFilePiece; filePiece++){

sprintf(filename_raw,"/scratch/nfs/data/athen/rootdata/run_%d_%d.root/MUTB/h10",RunNumber
,filePiece);
      chain_data->Add(filename_raw);
    }
    MyClass_ntup t(chain_data);
    cout<<"Number of events is "<<t.fChain->GetEntries()<<endl; // I wonder how many
events we have...

    for (cut_station = 2; cut_station == 2; cut_station++){
      for (int eventCount = 0; eventCount < t.fChain->GetEntries(); eventCount++) // Loop
over all events
       {
         t.GetEntry(eventCount);
         for (int hitCount = 0; hitCount < t.nmdtdig; hitCount++) // Loop over hits for
each event
          {
            wireCount = (int(t.mwire[hitCount])-1); // "wireCount" is easier to type
                                                    // (and read) than
"int(t.mwire[hitCount])-1".

             //  Let's make some cuts...
             if ((t.mstation[hitCount] == cut_station) && (wireCount < nWires)) {
               (histo_mdttdc_wire[wireCount])->Fill(t.mdttdc[hitCount]);  // Fill
histo_mdttdc
               int cycle = int(2*(2 + t.meta[hitCount]) +  t.mmulti[hitCount] - 1);
               if( cycle<0 || cycle>3 ) continue;
               int tubenumber = -1;
               if(  t.mmulti[hitCount] == 2) tubenumber = wireCount%3;
               if(  t.mmulti[hitCount] == 1) tubenumber = 2 - wireCount%3;

               if( ( (t.mmulti[hitCount] == 1) && (t.mlayer[hitCount] == 1) &&
(t.mwire[hitCount] == 55) ) ||
                   ( (t.mmulti[hitCount] == 1) && (t.mlayer[hitCount] == 3) &&
(t.mwire[hitCount] == 55) ) ||
                   ( (t.mmulti[hitCount] == 2) && (t.mlayer[hitCount] == 1) &&
(t.mwire[hitCount] == 56) ) ||
                   ( (t.mmulti[hitCount] == 2) && (t.mlayer[hitCount] == 2) &&
(t.mwire[hitCount] == 56) ) ){
                 tubenumber = 2;}
               if( ( (t.mmulti[hitCount] == 1) && (t.mlayer[hitCount] == 1) &&
(t.mwire[hitCount] == 56) ) ||
                   ( (t.mmulti[hitCount] == 1) && (t.mlayer[hitCount] == 2) &&
(t.mwire[hitCount] == 55) ) ||
                   ( (t.mmulti[hitCount] == 2) && (t.mlayer[hitCount] == 2) &&
(t.mwire[hitCount] == 55) ) ||
                   ( (t.mmulti[hitCount] == 2) && (t.mlayer[hitCount] == 3) &&
(t.mwire[hitCount] == 56) ) ){
                 tubenumber = 1;}
               if( ( (t.mmulti[hitCount] == 1) && (t.mlayer[hitCount] == 2) &&
(t.mwire[hitCount] == 56) ) ||
                   ( (t.mmulti[hitCount] == 1) && (t.mlayer[hitCount] == 3) &&
(t.mwire[hitCount] == 56) ) ||
                   ( (t.mmulti[hitCount] == 2) && (t.mlayer[hitCount] == 1) &&
(t.mwire[hitCount] == 55) ) ||
                   ( (t.mmulti[hitCount] == 2) && (t.mlayer[hitCount] == 3) &&
(t.mwire[hitCount] == 55) ) ){
                 tubenumber = 0;}

               if( tubenumber<0) continue;

               histo_mdttdc_tube[tubenumber][cycle]->Fill(t.mdttdc[hitCount]-
t.mtrigger[hitCount]);
             }
           } // ending hitCount loop
```

```
        } // ending eventCount loop
       double init_par[] = {0.22, 22, 8.5, 214, 656, 1480, 7.4, 22.4};

       for(int tubeCount = 0;tubeCount <3;tubeCount++){
         char name_for_canvas[50];
         sprintf(name_for_canvas,"canv_tube%d_Run%d",tubeCount,RunNumber);
         char title_for_canvas[50];
         sprintf(title_for_canvas,"histo of mdttdc Run %d (Tube
%d)",RunNumber,tubeCount+1);
         //  To take a look at things as we go
         canv[CountRunNumber][tubeCount] = new
TCanvas(name_for_canvas,title_for_canvas,1200,900);

         for(cycleCount = 0; cycleCount<4; cycleCount++){
           char name_for_pad[50];

sprintf(name_for_pad,"pad_canv_tube%d_Run%d_cycle%d",tubeCount+1,RunNumber,cycleCount+1);
           char title_for_pad[50];
           sprintf(title_for_pad,"histo of mdttdc Run %d (Tube %d, Cycle
%d)",RunNumber,tubeCount+1,cycleCount+1);

           if(cycleCount == 0){
             pad_canv[CountRunNumber][tubeCount][0] = new
TPad(name_for_pad,title_for_pad,0.01,0.51,0.49,0.99);}// top left pad
           if(cycleCount == 1){
             pad_canv[CountRunNumber][tubeCount][1] = new
TPad(name_for_pad,title_for_pad,0.51,0.51,0.99,0.99);}// top right pad
           if(cycleCount == 2){
             pad_canv[CountRunNumber][tubeCount][2] = new
TPad(name_for_pad,title_for_pad,0.01,0.01,0.49,0.49);}// bot left pad
           if(cycleCount == 3){
             pad_canv[CountRunNumber][tubeCount][3] = new
TPad(name_for_pad,title_for_pad,0.51,0.01,0.99,0.49);}// bot right pad

           funct_fit->SetParameters(init_par);
           histo_mdttdc_tube[tubeCount][cycleCount]->Fit("funct_fit","Q0","",550,2000);

           pad_canv[CountRunNumber][tubeCount][cycleCount]->Draw();
           pad_canv[CountRunNumber][tubeCount][cycleCount]->cd();
           histo_mdttdc_tube[tubeCount][cycleCount]->Draw();
           pad_canv[CountRunNumber][tubeCount][cycleCount]->Modified();
           canv[CountRunNumber][tubeCount]->cd();

           // And finally filling the ntuple...
           ntup_RunData->Fill(2,                              // station
                              -2 + (cycleCount>>1), // eta
                               1 + (cycleCount &1), // multi
                              histo_mdttdc_tube[tubeCount][cycleCount]->GetEntries(), //
number of entries
                              tubeCount + 1,              // tube number
                              funct_fit->GetParameter(4), // array_t0[cycleCount],
                              funct_fit->GetParameter(5), // array_tmax[cycleCount],
                              funct_fit->GetParameter(6), // array_resol_t0[cycleCount],
                              funct_fit->GetParameter(7), // array_resol_tmax[cycleCount],
                              funct_fit->GetParError(4),  // array_err_t0[cycleCount],
                              funct_fit->GetParError(5),  //array_err_tmax[cycleCount],
                              funct_fit->GetParError(6),  //array_err_resol_t0[cycleCount],
                              funct_fit->GetParError(7),
//array_err_resol_tmax[cycleCount],
                              (funct_fit->GetChisquare())/(funct_fit->GetNDF()));  //
array_chiSq[cycleCount]);
         }// ending cycleCount

         //  Saving the histo images...
         char histo_filename_ps[100];
         char histo_filename_gif[100];
         sprintf(histo_filename_ps,"immagini/histo_spectra_fits/histo_Run%d_Tube%d.ps",RunN
umber,tubeCount+1);
         sprintf(histo_filename_gif,"immagini/histo_spectra_fits/histo_Run%d_Tube%d.gif",Ru
nNumber,tubeCount+1);
         canv[CountRunNumber][tubeCount]->SaveAs(histo_filename_ps);
```

```
        canv[CountRunNumber][tubeCount]->SaveAs(histo_filename_gif);
        canv[CountRunNumber][tubeCount]->Close();

      }// ending tubeCount
    } // ending cut_station loop

    //  Writing the ntuple to a file...
    char newFilename[100];
    char description_of_ntuple[30];
    sprintf(newFilename,"ntup_files/file_ntup_Run%d.root",RunNumber);
    sprintf(description_of_ntuple,"Ntuple of Run %d",RunNumber);

    TFile *hfile = new TFile(newFilename,"RECREATE",description_of_ntuple);
    ntup_RunData->SetDirectory(hfile);
    hfile->Write();

    delete chain_data;

  }//ending CountRunNumber loop

}
```

4.1.2 Macro 2

```
//  This macro takes the ntuple data from macro 1, attaches the proper
//  temps and times to the data and then plots tdrift vs tube number
//  for each of the runs and stations.  It also prints the values out to the screen.
//  In addtion, it calibrates the data for temp and bit correction.


#include <sstream>
#include <iostream>
#include "MyClass_ana.h"
#include "TChain.h"
#include "TFile.h"
#include "TH1.h"
#include "TH2.h"
#include "TCanvas.h"
#include "TPad.h"
#include "TGraphErrors.h"


void ana_mean_allRuns()
{
  // Defining all of our graphs and arrays and more:
  TGraphErrors* graph_tdrift_nEntries_tube[12][3][4];
  TGraphErrors* graph_tdrift_tubeNum_tube[12][3][4];
  float cut_nEntries = 1; // should be unnecessary
  float cut_error = 50; // should be unnecessary
  TCanvas* c1[12];
  TPad* pad_c1[12][4];


  for(int i=0; i<4; i++)
    for(int j=0; j<3; j++)
      for(int k=0; k<12; k++){
        graph_tdrift_nEntries_tube[k][j][i] = new TGraphErrors(24);
        graph_tdrift_tubeNum_tube[k][j][i]  = new TGraphErrors(24);
  }

  TArrayF array_RunNumber = TArrayF(12);
  array_RunNumber[0] = 700032;
  array_RunNumber[1] = 700090;
```

```
  array_RunNumber[2] = 700108;
  array_RunNumber[3] = 700172;
  array_RunNumber[4] = 1091;
  array_RunNumber[5] = 1222;
  array_RunNumber[6] = 1226;
  array_RunNumber[7] = 1235;
  array_RunNumber[8] = 1256;
  array_RunNumber[9] = 1374;
  array_RunNumber[10] = 1431;
  array_RunNumber[11] = 1559;
  int RunNumber;

  for(int CountRunNumber = 0; CountRunNumber<=11; CountRunNumber++){
    RunNumber = int(array_RunNumber[CountRunNumber]);
    cout << "RunNumber: " << RunNumber <<endl;


    // Defining the chain, importing the data, setting up a MyClass_ana called "data"...
    char input_filename[120];
    TChain *chain_data = new TChain("data");

sprintf(input_filename,"/afs/lnf/user/j/jkamin/work/atlas/root/final_macros/ntup_files/fi
le_ntup_Run%d.root/ntup_RunData",RunNumber);
    chain_data->Add(input_filename);
    MyClass_ana data(chain_data);
    int nEntries = int(data.fChain->GetEntries());
    cout<<"Number of events is "<< nEntries << endl << endl;

    //  Defining some floats (for our graphs) and ints (for our loops)...
    float tdrift[12][3][4],err_tdrift[12][3][4];
    int cut_eta,cut_multi,cycle;
    static const float ref_temperature = 27.0;
    static const float bit_correction = 0.78125;
    float temperature = 27.0;
    float ore = 0;
    float temperature_correction = 0;

    //  Looping over all the cuts...
    for(cut_eta = -2; cut_eta <= -1; cut_eta++){
      for(cut_multi = 1; cut_multi <=2; cut_multi++){

        //  Setting the appropriate tempuratura and ore(time)...
        if(RunNumber == 1091   && cut_multi == 1){
          temperature = 27.60;
          ore = 428;}
        if(RunNumber == 1091   && cut_multi == 2){
          temperature = 27.42;
          ore = 428;}
        if(RunNumber == 1220   && cut_multi == 1){
          temperature = 28.48;
          ore = 492;}
        if(RunNumber == 1220   && cut_multi == 2){
          temperature = 28.38;
          ore = 492;}
        if(RunNumber == 1222   && cut_multi == 1){
          temperature = 28.83;
          ore = 495;}
        if(RunNumber == 1222   && cut_multi == 2){
          temperature = 28.65;
          ore = 495;}
        if(RunNumber == 1223   && cut_multi == 1){
          temperature = 28.85;
          ore = 496;}
        if(RunNumber == 1223   && cut_multi == 2){
          temperature = 28.66;
          ore = 496;}
        if(RunNumber == 1226   && cut_multi == 1){
          temperature = 28.18;
          ore = 499;}
        if(RunNumber == 1226   && cut_multi == 2){
          temperature = 28.06;
```

```
        ore = 499;}
      if(RunNumber == 1235   && cut_multi == 1){
        temperature = 25.33;
        ore = 508;}
      if(RunNumber == 1235   && cut_multi == 2){
        temperature = 25.72;
        ore = 508;}
      if(RunNumber == 1256   && cut_multi == 1){
        temperature = 26.07;
        ore = 535;}
      if(RunNumber == 1256   && cut_multi == 2){
        temperature = 26.37;
        ore = 535;}
      if(RunNumber == 1263){
        ore = 544;
        cout << "We don't have a temperature measurement for this run!!!" << endl;}
      if(RunNumber == 1270){
        ore = 554;
        cout << "We don't have a temperature measurement for this run!!!" << endl;}
      if(RunNumber == 1374   && cut_multi == 1){
        temperature = 25.31;
        ore = 672;}
      if(RunNumber == 1374   && cut_multi == 2){
        temperature = 25.57;
        ore = 672;}
      if(RunNumber == 1431   && cut_multi == 1){
        temperature = 26.02;
        ore = 694;}
      if(RunNumber == 1431   && cut_multi == 2){
        temperature = 26.17;
        ore = 694;}
      if(RunNumber == 1559   && cut_multi == 1){
        temperature = 26.76;
        ore = 833;}
      if(RunNumber == 1559   && cut_multi == 2){
        temperature = 26.54;
        ore = 833;}
      if(RunNumber == 700032 && cut_multi == 1){
        temperature = 23.81;
        ore = 4;}
      if(RunNumber == 700032 && cut_multi == 2){
        temperature = 24.17;
        ore = 4;}
      if(RunNumber == 700090 && cut_multi == 1){
        temperature = 24.86;
        ore = 30;}
      if(RunNumber == 700090 && cut_multi == 2){
        temperature = 25.23;
        ore = 30;}
      if(RunNumber == 700108 && cut_multi == 1){
        temperature = 26.88;
        ore = 34;}
      if(RunNumber == 700108 && cut_multi == 2){
        temperature = 26.77;
        ore = 34;}
      if(RunNumber == 700172 && cut_multi == 1){
        temperature = 26.92;
        ore = 47;}
      if(RunNumber == 700172 && cut_multi == 2){
        temperature = 26.90;
        ore = 47;}
      if(ore==0) cout <<"The gas was changed for this run so we can't compare the time
with the rest (or some source was on)!!!" << endl;


      for(int entryCount = 0; entryCount < nEntries; entryCount++){
        data.GetEntry(entryCount);

        //  Let's make some cuts...
        if(data.eta == cut_eta){
          if(data.multi == cut_multi){
```

```
            if( data.eta == -2 && data.multi == 1) cycle = 1;
            if( data.eta == -2 && data.multi == 2) cycle = 2;
            if( data.eta == -1 && data.multi == 1) cycle = 3;
            if( data.eta == -1 && data.multi == 2) cycle = 4;

            if(data.tube==1)  cout <<"cycle: "<<cycle<<endl;

            temperature_correction = (temperature-ref_temperature)*(-2.4); // correcting
for temperature.
            float tdrift_temp = ((data.tmax-data.t0)*bit_correction)-
(temperature_correction);
            cout <<"Tube#: "<<data.tube<<", tdrift: "<<tdrift_temp<<endl;
            float err_tdrift_temp =
sqrt((data.err_t0*data.err_t0)+(data.err_tmax*data.err_tmax));
            // we ignore the
            // temp for the errors.

            //  This is just to clean up the few large errored tdrifts
            //  that somehow snuck past the other cuts...
            if(err_tdrift_temp<2000 &&
               !((data.eta == -1) && (RunNumber == 700032)))
              if(err_tdrift_temp<2000 &&
                 !((data.eta == -1) && (RunNumber == 1374)))
                if(err_tdrift_temp<2000 &&
                   !((data.eta == -1) && (RunNumber == 1431))){

                  tdrift[CountRunNumber][int(data.tube)-1][cycle-1] = tdrift_temp;
                  err_tdrift[CountRunNumber][int(data.tube)-1][cycle-1] =
err_tdrift_temp;
                }// ending if err_tdrift<200

          }// ending if cut_multi
        }// ending if cut_eta
      }// ending entryCount

      //  Also, setting the points on the new graphs for those:
      for(int i = 0; i<3; i++){
        graph_tdrift_tubeNum_tube[CountRunNumber][i][cycle-1]-
>SetPoint(0,i+1,tdrift[CountRunNumber][i][cycle-1]);
        graph_tdrift_tubeNum_tube[CountRunNumber][i][cycle-1]-
>SetPointError(0,0,err_tdrift[CountRunNumber][i][cycle-1]);
      }
    }// ending cut_multi
  }// ending cut_eta

 // Creating the Canvas and declaring Pads...
   char name_for_canvas[10];
   char title_for_canvas[50];
   sprintf(name_for_canvas,"c1_%d",CountRunNumber);
   sprintf(title_for_canvas,"mean values of t drift Run %d",RunNumber);
   c1[CountRunNumber] = new TCanvas(name_for_canvas,title_for_canvas);

   for(cycle = 1; cycle < 5; cycle++){

     char name_for_pad[10];
     sprintf(name_for_pad,"pad_c1_cycle:%d_RunNum:%d",cycle,RunNumber);
     //  Setting up the pads in the four corners of the canvas:
     pad_c1[CountRunNumber][0] = new TPad("pad_c1_0", "eta-2,m1
means",0.01,0.51,0.49,0.99);
     pad_c1[CountRunNumber][1] = new TPad("pad_c1_1", "eta-2,m2
means",0.51,0.51,0.99,0.99);
     pad_c1[CountRunNumber][2] = new TPad("pad_c1_2", "eta-1,m1
means",0.01,0.01,0.49,0.49);
     pad_c1[CountRunNumber][3] = new TPad("pad_c1_3", "eta-1,m2
means",0.51,0.01,0.99,0.49);

     c1[CountRunNumber]->cd();
     pad_c1[CountRunNumber][cycle-1]->Draw();
     pad_c1[CountRunNumber][cycle-1]->cd();
```

```
      (graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->GetYaxis())-
>SetLimits(640,665);
      (graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->GetXaxis())-
>SetLimits(0,10);
      (graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->GetYaxis())->UnZoom();
      (graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->GetXaxis())->UnZoom();

      (graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->GetYaxis())-
>SetTitle("tdrift (ns)");
      (graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->GetXaxis())->SetTitle("tube
number");

      graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->SetLineColor(2);
      graph_tdrift_tubeNum_tube[CountRunNumber][1][cycle-1]->SetLineColor(4);
      graph_tdrift_tubeNum_tube[CountRunNumber][2][cycle-1]->SetLineColor(3);
      char title_for_graphs[50];
      sprintf(title_for_graphs,"y:tdrift vs x:tubeNum Run%d, BML%d, multi%d",RunNumber, -
2+((cycle-1)>>1), 1+((cycle-1)&1) );
      graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->SetTitle(title_for_graphs);

      graph_tdrift_tubeNum_tube[CountRunNumber][0][cycle-1]->Draw("AP");
      graph_tdrift_tubeNum_tube[CountRunNumber][1][cycle-1]->Draw("P");
      graph_tdrift_tubeNum_tube[CountRunNumber][2][cycle-1]->Draw("P");

      pad_c1[CountRunNumber][cycle-1]->Modified();
      c1[CountRunNumber]->cd();
    }

       char output_ps_filename[70];
       char output_gif_filename[70];
       sprintf(output_ps_filename,"immagini/tdrift_vs_tubeNum/TdriftVsTubeNum_run%d.ps",R
unNumber);
       sprintf(output_gif_filename,"immagini/tdrift_vs_tubeNum/TdriftVsTubeNum_run%d.gif"
,RunNumber);

       c1[CountRunNumber]->SetCanvasSize(950,712);
       c1[CountRunNumber]->SaveAs(output_ps_filename);
       c1[CountRunNumber]->SaveAs(output_gif_filename);

  }//ending CountRunNumber loop

  cout << "We made cuts: " << endl;
  cout << "nEntries >= "<<cut_nEntries<<" && error <= "<<cut_error<<" ns"<<endl<<endl;
  cout << "------ Cycle 1: BML 2, Multi 1 ------" << endl;
  cout << "------ Cycle 2: BML 2, Multi 2 ------" << endl;
  cout << "------ Cycle 3: BML 1, Multi 1 ------" << endl;
  cout << "------ Cycle 4: BML 1, Multi 2 ------" << endl;
}
```

4.1.3 Macro 3

```
//  This macro takes the ntuple data from macro 1, attaches the proper
//  temps and times to the data and then plots tdrift vs time
//  for each tube number and station.  It also prints the values out to the screen.
//  In addtion, it calibrates the data for temp and bit correction.


#include <sstream>
#include <iostream>
#include "MyClass_ana.h"
#include "TChain.h"
#include "TFile.h"
#include "TH1.h"
#include "TH2.h"
#include "TCanvas.h"
#include "TPad.h"
#include "TGraphErrors.h"
```

```
void ana_tdrift_vs_time()
{
  // Defining all of our graphs and arrays and more:
  TGraphErrors* graph_tdrift_nEntries_tube[12][3][4];
  TGraphErrors* graph_tdrift_tubeNum_tube[12][3][4];
  TGraphErrors* graph_tdrift_vs_time[3][4];
  float cut_nEntries = 1; // should be unnecessary
  float cut_error = 50; // should be unnecessary
  TCanvas* c1[4];
  TPad* pad_c1[3][4];

  for(int i=0; i<4; i++)
    for(int j=0; j<3; j++){
      graph_tdrift_vs_time[j][i] = new TGraphErrors(12);
      for(int k=0; k<12; k++){
        graph_tdrift_nEntries_tube[k][j][i] = new TGraphErrors(24);
        graph_tdrift_tubeNum_tube[k][j][i]  = new TGraphErrors(24);
      }
    }

  TArrayF array_RunNumber = TArrayF(12);
  array_RunNumber[0] = 700032;
  array_RunNumber[1] = 700090;
  array_RunNumber[2] = 700108;
  array_RunNumber[3] = 700172;
  array_RunNumber[4] = 1091;
  array_RunNumber[5] = 1222;
  array_RunNumber[6] = 1226;
  array_RunNumber[7] = 1235;
  array_RunNumber[8] = 1256;
  array_RunNumber[9] = 1374;
  array_RunNumber[10] = 1431;
  array_RunNumber[11] = 1559;
  int RunNumber;

  for(int CountRunNumber = 0; CountRunNumber<=11; CountRunNumber++){
    RunNumber = int(array_RunNumber[CountRunNumber]);
    cout << "RunNumber: " << RunNumber <<endl;

    // Defining the chain, importing the data, setting up a MyClass_ana called "data"...
    char input_filename[120];
    TChain *chain_data = new TChain("data");

sprintf(input_filename,"/afs/lnf/user/j/jkamin/work/atlas/root/final_macros/ntup_files/fi
le_ntup_Run%d.root/ntup_RunData",RunNumber);
    chain_data->Add(input_filename);
    MyClass_ana data(chain_data);
    int nEntries = int(data.fChain->GetEntries());
    cout<<"Number of events is "<< nEntries << endl << endl;

    //  Defining some floats (for our graphs) and ints (for our loops)...
    float tdrift[12][3][4],err_tdrift[12][3][4];
    int cut_eta,cut_multi,cycle;
    static const float ref_temperature = 27.0;
    static const float bit_correction = 0.78125;
    float temperature = 27.0;
    float ore = 0;
    float temperature_correction = 0;

    //  Looping over all the cuts...
    for(cut_eta = -2; cut_eta <= -1; cut_eta++){
      for(cut_multi = 1; cut_multi <=2; cut_multi++){

        //  Setting the appropriate tempuratura and ore(time)...
        if(RunNumber == 1091   && cut_multi == 1){
          temperature = 27.60;
          ore = 428;}
        if(RunNumber == 1091   && cut_multi == 2){
          temperature = 27.42;
          ore = 428;}
        if(RunNumber == 1220   && cut_multi == 1){
```

```
  temperature = 28.48;
  ore = 492;}
if(RunNumber == 1220   && cut_multi == 2){
  temperature = 28.38;
  ore = 492;}
if(RunNumber == 1222   && cut_multi == 1){
  temperature = 28.83;
  ore = 495;}
if(RunNumber == 1222   && cut_multi == 2){
  temperature = 28.65;
  ore = 495;}
if(RunNumber == 1223   && cut_multi == 1){
  temperature = 28.85;
  ore = 496;}
if(RunNumber == 1223   && cut_multi == 2){
  temperature = 28.66;
  ore = 496;}
if(RunNumber == 1226   && cut_multi == 1){
  temperature = 28.18;
  ore = 499;}
if(RunNumber == 1226   && cut_multi == 2){
  temperature = 28.06;
  ore = 499;}
if(RunNumber == 1227   && cut_multi == 1){
  temperature = 27.81;
  ore = 500;}
if(RunNumber == 1227   && cut_multi == 2){
  temperature = 27.76;
  ore = 500;}
if(RunNumber == 1235   && cut_multi == 1){
  temperature = 25.33;
  ore = 508;}
if(RunNumber == 1235   && cut_multi == 2){
  temperature = 25.72;
  ore = 508;}
if(RunNumber == 1256   && cut_multi == 1){
  temperature = 26.07;
  ore = 535;}
if(RunNumber == 1256   && cut_multi == 2){
  temperature = 26.37;
  ore = 535;}
if(RunNumber == 1263){
  ore = 544;
  cout << "We don't have a temperature measurement for this run!!!" << endl;}
if(RunNumber == 1270){
  ore = 554;
  cout << "We don't have a temperature measurement for this run!!!" << endl;}
if(RunNumber == 1374   && cut_multi == 1){
  temperature = 25.31;
  ore = 672;}
if(RunNumber == 1374   && cut_multi == 2){
  temperature = 25.57;
  ore = 672;}
if(RunNumber == 1431   && cut_multi == 1){
  temperature = 26.02;
  ore = 694;}
if(RunNumber == 1431   && cut_multi == 2){
  temperature = 26.17;
  ore = 694;}
if(RunNumber == 1559   && cut_multi == 1){
  temperature = 26.76;
  ore = 833;}
if(RunNumber == 1559   && cut_multi == 2){
  temperature = 26.54;
  ore = 833;}
if(RunNumber == 700032 && cut_multi == 1){
  temperature = 23.81;
  ore = 4;}
if(RunNumber == 700032 && cut_multi == 2){
  temperature = 24.17;
  ore = 4;}
```

```
        if(RunNumber == 700090 && cut_multi == 1){
          temperature = 24.86;
          ore = 30;}
        if(RunNumber == 700090 && cut_multi == 2){
          temperature = 25.23;
          ore = 30;}
        if(RunNumber == 700108 && cut_multi == 1){
          temperature = 26.88;
          ore = 34;}
        if(RunNumber == 700108 && cut_multi == 2){
          temperature = 26.77;
          ore = 34;}
        if(RunNumber == 700172 && cut_multi == 1){
          temperature = 26.92;
          ore = 47;}
        if(RunNumber == 700172 && cut_multi == 2){
          temperature = 26.90;
          ore = 47;}
        if(ore==0) cout <<"The gas was changed for this run so we can't compare the time
with the rest!!!" << endl;

        for(int entryCount = 0; entryCount < nEntries; entryCount++){
          data.GetEntry(entryCount);

          //  Let's make some cuts...
          if(data.eta == cut_eta){
            if(data.multi == cut_multi){

              if( data.eta == -2 && data.multi == 1) cycle = 1;
              if( data.eta == -2 && data.multi == 2) cycle = 2;
              if( data.eta == -1 && data.multi == 1) cycle = 3;
              if( data.eta == -1 && data.multi == 2) cycle = 4;

              if(data.tube==1)  cout <<"cycle: "<<cycle<<endl;

              temperature_correction = (temperature-ref_temperature)*(-2.4); // correcting
for temperature.
              float tdrift_temp = ((data.tmax-data.t0)*bit_correction)-
(temperature_correction);
              cout <<"Tube#: "<<data.tube<<", tdrift: "<<tdrift_temp<<endl;
              float err_tdrift_temp =
sqrt((data.err_t0*data.err_t0)+(data.err_tmax*data.err_tmax));
              // we ignore the
              // temp for the errors.

              //  This is just to clean up the few large errored tdrifts
              //  that somehow snuck past the other cuts…
              if(err_tdrift_temp<2000 &&
                 !((data.eta == -1) && (RunNumber == 700032)))
                if(err_tdrift_temp<2000 &&
                   (!((data.eta == -1) && (RunNumber == 1374))))
                  if(err_tdrift_temp<2000 &&
                     (!((data.eta == -1) && (RunNumber == 1431)))){
                    tdrift[CountRunNumber][int(data.tube)-1][cycle-1] = tdrift_temp;
                    err_tdrift[CountRunNumber][int(data.tube)-1][cycle-1] =
err_tdrift_temp;
                  }// ending if err_tdrift<200
            }// ending if cut_multi
          }// ending if cut_eta
        }// ending entryCount


        //  Also, setting the points on the new graphs for those:
        for(int i = 0; i<3; i++){
          graph_tdrift_tubeNum_tube[CountRunNumber][i][cycle-1]-
>SetPoint(0,i+1,tdrift[CountRunNumber][i][cycle-1]);
          graph_tdrift_tubeNum_tube[CountRunNumber][i][cycle-1]-
>SetPointError(0,0,err_tdrift[CountRunNumber][i][cycle-1]);
          graph_tdrift_vs_time[i][cycle-1]-
>SetPoint(CountRunNumber,ore,tdrift[CountRunNumber][i][cycle-1]);
```

```
         graph_tdrift_vs_time[i][cycle-1]-
>SetPointError(CountRunNumber,0,err_tdrift[CountRunNumber][i][cycle-1]);
       }
      }// ending cut_multi
    }// ending cut_eta

  }//ending CountRunNumber loop

  for(int cycle = 0; cycle < 4; cycle++){

    // Creating the Canvas...
    char name_for_canvas[10];
    char title_for_canvas[50];
    sprintf(name_for_canvas,"c1_cycle%d",cycle+1);
    sprintf(title_for_canvas,"mean values of tdrift vs time cycle%d",cycle+1);
    c1[cycle] = new TCanvas(name_for_canvas,title_for_canvas);

    for(int tube = 0; tube < 3; tube++){

      char name_for_pad[10];
      char title_for_pad[50];
      sprintf(name_for_pad,"pad_c1_tube%d_cycle%d",tube+1,cycle+1);
      sprintf(title_for_pad,"tdrift vs time, tube%d cycle%d",tube+1,cycle+1);
      // Setting the pads in the corners…
      if(tube==0){  pad_c1[tube][cycle] = new
TPad(name_for_pad,title_for_pad,0.01,0.51,0.49,0.99);}
      if(tube==1){  pad_c1[tube][cycle] = new
TPad(name_for_pad,title_for_pad,0.51,0.51,0.99,0.99);}
      if(tube==2){  pad_c1[tube][cycle] = new
TPad(name_for_pad,title_for_pad,0.01,0.01,0.49,0.49);}

      c1[cycle]->cd();
      pad_c1[tube][cycle]->Draw();
      pad_c1[tube][cycle]->cd();

      (graph_tdrift_vs_time[tube][cycle]->GetYaxis())->SetLimits(644,664);
      (graph_tdrift_vs_time[tube][cycle]->GetYaxis())->UnZoom();

      (graph_tdrift_vs_time[tube][cycle]->GetYaxis())->SetTitle("tdrift (ns)");
      (graph_tdrift_vs_time[tube][cycle]->GetXaxis())->SetTitle("time (hours)");

      char title_for_graphs[50];
      sprintf(title_for_graphs,"y:tdrift vs x:time tube%d, BML%d, multi%d",tube+1, -
2+(cycle>>1), 1+(cycle&1) );
      graph_tdrift_vs_time[tube][cycle]->SetTitle(title_for_graphs);
      graph_tdrift_vs_time[tube][cycle]->Draw("AP");

      pad_c1[tube][cycle]->Modified();
      c1[cycle]->cd();
    }// ending tube loop for displaying images

    char output_ps_filename[70];
    char output_gif_filename[70];

sprintf(output_ps_filename,"immagini/tdrift_vs_time/TdriftVsTime_cycle%d.ps",cycle+1);

sprintf(output_gif_filename,"immagini/tdrift_vs_time/TdriftVsTime_cycle%d.gif",cycle+1);

    c1[cycle]->SetCanvasSize(950,712);
    c1[cycle]->SaveAs(output_ps_filename);
    c1[cycle]->SaveAs(output_gif_filename);
  }// ending cycle loop for displaying images

  cout << "We made cuts: " << endl;
  cout << "nEntries >= "<<cut_nEntries<<" && error <= "<<cut_error<<" ns"<<endl<<endl;
  cout << "------ Cycle 1: BML 2, Multi 1 ------" << endl;
  cout << "------ Cycle 2: BML 2, Multi 2 ------" << endl;
  cout << "------ Cycle 3: BML 1, Multi 1 ------" << endl;
  cout << "------ Cycle 4: BML 1, Multi 2 ------" << endl << endl;
}
```

4.1.4 Macro 4

```
//  This macro takes the ntuple data from macro 1, attaches the proper
//  temps and times to the data and then plots subtracted tdrifts vs tube number
//  for each of the stations.  It also prints the values out to the screen.
//  In addtion, it calibrates the data for temp and bit correction.


#include <sstream>
#include <iostream>
#include "MyClass_ana.h"
#include "TChain.h"
#include "TFile.h"
#include "TH1.h"
#include "TH2.h"
#include "TCanvas.h"
#include "TPad.h"
#include "TGraphErrors.h"

void ana_subtract_tdrift_vs_time()
{
  // Defining all of our graphs and arrays and more:
  TGraphErrors* graph_tdrift_nEntries_tube[12][3][4];
  TGraphErrors* graph_tdrift_tubeNum_tube[12][3][4];
  TGraphErrors* graph_tdrift_vs_time[3][4];
  TGraphErrors* graph_subtract_tdrift_vs_time[3][4];
  float cut_nEntries = 1; // should be unnecessary
  float cut_error = 50; // should be unnecessary
  TCanvas* c1[4];
  TPad* pad_c1[3][4];

  for(int i=0; i<4; i++)
    for(int j=0; j<3; j++){
      graph_tdrift_vs_time[j][i] = new TGraphErrors(12);
      graph_subtract_tdrift_vs_time[j][i] = new TGraphErrors(12);
      for(int k=0; k<12; k++){
        graph_tdrift_nEntries_tube[k][j][i] = new TGraphErrors(24);
        graph_tdrift_tubeNum_tube[k][j][i]  = new TGraphErrors(24);
      }
    }

  TArrayF array_RunNumber = TArrayF(12);
  array_RunNumber[0] = 700032;
  array_RunNumber[1] = 700090;
  array_RunNumber[2] = 700108;
  array_RunNumber[3] = 700172;
  array_RunNumber[4] = 1091;
  array_RunNumber[5] = 1222;
  array_RunNumber[6] = 1226;
  array_RunNumber[7] = 1235;
  array_RunNumber[8] = 1256;
  array_RunNumber[9] = 1374;
  array_RunNumber[10] = 1431;
  array_RunNumber[11] = 1559;
  int RunNumber;

  for(int CountRunNumber = 0; CountRunNumber<=11; CountRunNumber++){
    RunNumber = int(array_RunNumber[CountRunNumber]);
    cout << "RunNumber: " << RunNumber <<endl;

    // Defining the chain, importing the data, setting up a MyClass_ana called "data"...
    char input_filename[120];
    TChain *chain_data = new TChain("data");
```

```
sprintf(input_filename,"/afs/lnf/user/j/jkamin/work/atlas/root/final_macros/ntup_files/fi
le_ntup_Run%d.root/ntup_RunData",RunNumber);
    chain_data->Add(input_filename);
    MyClass_ana data(chain_data);
    int nEntries = int(data.fChain->GetEntries());
    cout<<"Number of events is "<< nEntries << endl << endl;

    //  Defining some floats (for our graphs) and ints (for our loops)...
    float tdrift[12][3][4],err_tdrift[12][3][4];
    int cut_eta,cut_multi,cycle;
    static const float ref_temperature = 27.0;
    static const float bit_correction = 0.78125;
    float temperature = 27.0;
    float ore = 0;
    float temperature_correction = 0;

    //  Looping over all the cuts...
    for(cut_eta = -2; cut_eta <= -1; cut_eta++){
      for(cut_multi = 1; cut_multi <=2; cut_multi++){

        //  Setting the appropriate tempuratura and ore(time)...
        if(RunNumber == 1091   && cut_multi == 1){
          temperature = 27.60;
          ore = 428;}
        if(RunNumber == 1091   && cut_multi == 2){
          temperature = 27.42;
          ore = 428;}
        if(RunNumber == 1220   && cut_multi == 1){
          temperature = 28.48;
          ore = 492;}
        if(RunNumber == 1220   && cut_multi == 2){
          temperature = 28.38;
          ore = 492;}
        if(RunNumber == 1222   && cut_multi == 1){
          temperature = 28.83;
          ore = 495;}
        if(RunNumber == 1222   && cut_multi == 2){
          temperature = 28.65;
          ore = 495;}
        if(RunNumber == 1223   && cut_multi == 1){
          temperature = 28.85;
          ore = 496;}
        if(RunNumber == 1223   && cut_multi == 2){
          temperature = 28.66;
          ore = 496;}
        if(RunNumber == 1226   && cut_multi == 1){
          temperature = 28.18;
          ore = 499;}
        if(RunNumber == 1226   && cut_multi == 2){
          temperature = 28.06;
          ore = 499;}
        if(RunNumber == 1227   && cut_multi == 1){
          temperature = 27.81;
          ore = 500;}
        if(RunNumber == 1227   && cut_multi == 2){
          temperature = 27.76;
          ore = 500;}
        if(RunNumber == 1235   && cut_multi == 1){
          temperature = 25.33;
          ore = 508;}
        if(RunNumber == 1235   && cut_multi == 2){
          temperature = 25.72;
          ore = 508;}
        if(RunNumber == 1256   && cut_multi == 1){
          temperature = 26.07;
          ore = 535;}
        if(RunNumber == 1256   && cut_multi == 2){
          temperature = 26.37;
          ore = 535;}
        if(RunNumber == 1263){  ore = 544;
```

```
          cout << "We don't have a temperature measurement for this run!!!" << endl;}
        if(RunNumber == 1270){  ore = 554;
          cout << "We don't have a temperature measurement for this run!!!" << endl;}
        if(RunNumber == 1374   && cut_multi == 1){
          temperature = 25.31;
          ore = 672;}
        if(RunNumber == 1374   && cut_multi == 2){
          temperature = 25.57;
          ore = 672;}
        if(RunNumber == 1431   && cut_multi == 1){
          temperature = 26.02;
          ore = 694;}
        if(RunNumber == 1431   && cut_multi == 2){
          temperature = 26.17;
          ore = 694;}
        if(RunNumber == 1559   && cut_multi == 1){
          temperature = 26.76;
          ore = 833;}
        if(RunNumber == 1559   && cut_multi == 2){
          temperature = 26.54;
          ore = 833;}
        if(RunNumber == 700032 && cut_multi == 1){
          temperature = 23.81;
          ore = 4;}
        if(RunNumber == 700032 && cut_multi == 2){
          temperature = 24.17;
          ore = 4;}
        if(RunNumber == 700090 && cut_multi == 1){
          temperature = 24.86;
          ore = 30;}
        if(RunNumber == 700090 && cut_multi == 2){
          temperature = 25.23;
          ore = 30;}
        if(RunNumber == 700108 && cut_multi == 1){
          temperature = 26.88;
          ore = 34;}
        if(RunNumber == 700108 && cut_multi == 2){
          temperature = 26.77;
          ore = 34;}
        if(RunNumber == 700172 && cut_multi == 1){
          temperature = 26.92;
          ore = 47;}
        if(RunNumber == 700172 && cut_multi == 2){
          temperature = 26.90;
          ore = 47;}
        if(ore==0) cout <<"The gas was changed for this run so we can't compare the time
with the rest!!!" << endl;

        for(int entryCount = 0; entryCount < nEntries; entryCount++){
          data.GetEntry(entryCount);

          //  Let's make some cuts...
          if(data.eta == cut_eta){
            if(data.multi == cut_multi){

              if( data.eta == -2 && data.multi == 1) cycle = 1;
              if( data.eta == -2 && data.multi == 2) cycle = 2;
              if( data.eta == -1 && data.multi == 1) cycle = 3;
              if( data.eta == -1 && data.multi == 2) cycle = 4;

              if(data.tube==1)  cout <<"cycle: "<<cycle<<endl;

              temperature_correction = (temperature-ref_temperature)*(-2.4); // correcting
for temperature.
              float tdrift_temp = ((data.tmax-data.t0)*bit_correction)-
(temperature_correction);
              cout <<"Tube#: "<<data.tube<<", tdrift: "<<tdrift_temp<<endl;
              float err_tdrift_temp =
sqrt((data.err_t0*data.err_t0)+(data.err_tmax*data.err_tmax));
              // we ignore the
              // temp for the errors.
```

```
            //  This is just to clean up the few large errored tdrifts
            //  that somehow snuck past the other cuts...
            if(err_tdrift_temp<2000 &&
               !((data.eta == -1) && (RunNumber == 700032)))
              if(err_tdrift_temp<2000 &&
                 !((data.eta == -1) && (RunNumber == 1374)))
                if(err_tdrift_temp<2000 &&
                   !((data.eta == -1) && (RunNumber == 1431))){
                  tdrift[CountRunNumber][int(data.tube)-1][cycle-1] = tdrift_temp;
                  err_tdrift[CountRunNumber][int(data.tube)-1][cycle-1] =
err_tdrift_temp;
                }// ending if err_tdrift<200
           }// ending if cut_multi
         }// ending if cut_eta
       }// ending entryCount


       //  Also, setting the points on the new graphs for those:
       for(int i = 0; i<3; i++){
         graph_tdrift_tubeNum_tube[CountRunNumber][i][cycle-1]-
>SetPoint(0,i+1,tdrift[CountRunNumber][i][cycle-1]);
         graph_tdrift_tubeNum_tube[CountRunNumber][i][cycle-1]-
>SetPointError(0,0,err_tdrift[CountRunNumber][i][cycle-1]);
         graph_tdrift_vs_time[i][cycle-1]-
>SetPoint(CountRunNumber,ore,tdrift[CountRunNumber][i][cycle-1]);
         graph_tdrift_vs_time[i][cycle-1]-
>SetPointError(CountRunNumber,0,err_tdrift[CountRunNumber][i][cycle-1]);
       }
      }// ending cut_multi
    }// ending cut_eta

  }//ending CountRunNumber loop

  for(int CountRunNumber = 0; CountRunNumber < 12; CountRunNumber++)
    for(int cycle = 0; cycle < 4; cycle++)
      for(int padNumber = 0; padNumber < 3; padNumber++){

       if((!(((cycle-1)>0) && (array_RunNumber[CountRunNumber] == 1374))))
         if((!(((cycle-1)>0) && (array_RunNumber[CountRunNumber] == 1431)))){

       char title_for_graphs[50];

       double ore_temp[3];
       double tdrift_temp[3];
       double err_tdrift_temp[3];
       graph_tdrift_vs_time[0][cycle]-
>GetPoint(CountRunNumber,ore_temp[0],tdrift_temp[0]);
       graph_tdrift_vs_time[1][cycle]-
>GetPoint(CountRunNumber,ore_temp[1],tdrift_temp[1]);
       graph_tdrift_vs_time[2][cycle]-
>GetPoint(CountRunNumber,ore_temp[2],tdrift_temp[2]);
       err_tdrift_temp[0] = graph_tdrift_vs_time[0][cycle]->GetErrorY(CountRunNumber);
       err_tdrift_temp[1] = graph_tdrift_vs_time[1][cycle]->GetErrorY(CountRunNumber);
       err_tdrift_temp[2] = graph_tdrift_vs_time[2][cycle]->GetErrorY(CountRunNumber);

       if(padNumber==0){
         sprintf(title_for_graphs,"y:tdrift vs x:time tube(1-2) BML%d, multi%d",-
2+(cycle>>1), 1+(cycle&1) );

         graph_subtract_tdrift_vs_time[0][cycle]-
>SetPoint(CountRunNumber,ore_temp[0],(tdrift_temp[0]-tdrift_temp[1]));
         graph_subtract_tdrift_vs_time[0][cycle]->SetPointError(CountRunNumber,0,

sqrt((err_tdrift_temp[0]*err_tdrift_temp[0])+

(err_tdrift_temp[1]*err_tdrift_temp[1])));
       }
       if(padNumber==1){
         sprintf(title_for_graphs,"y:tdrift vs x:time tube(1-3) BML%d, multi%d",-
2+(cycle>>1), 1+(cycle&1) );
```

```
          graph_subtract_tdrift_vs_time[1][cycle]-
>SetPoint(CountRunNumber,ore_temp[1],(tdrift_temp[0]-tdrift_temp[2]));
          graph_subtract_tdrift_vs_time[1][cycle]->SetPointError(CountRunNumber,0,

sqrt((err_tdrift_temp[0]*err_tdrift_temp[0])+

(err_tdrift_temp[2]*err_tdrift_temp[2])));
        }
        if(padNumber==2){
          sprintf(title_for_graphs,"y:tdrift vs x:time tube(2-3) BML%d, multi%d",-
2+(cycle>>1), 1+(cycle&1) );

          graph_subtract_tdrift_vs_time[2][cycle]-
>SetPoint(CountRunNumber,ore_temp[2],(tdrift_temp[1]-tdrift_temp[2]));
          graph_subtract_tdrift_vs_time[2][cycle]->SetPointError(CountRunNumber,0,

sqrt((err_tdrift_temp[1]*err_tdrift_temp[1])+

(err_tdrift_temp[2]*err_tdrift_temp[2])));
        }

        graph_subtract_tdrift_vs_time[padNumber][cycle]->SetTitle(title_for_graphs);

          }
      }

      for(int cycle = 0; cycle < 4; cycle++){

      // Creating the Canvas...
      char name_for_canvas[10];
      char title_for_canvas[50];
      sprintf(name_for_canvas,"c1_cycle%d",cycle+1);
      sprintf(title_for_canvas,"difference of tdrift vs time cycle%d",cycle+1);
      c1[cycle] = new TCanvas(name_for_canvas,title_for_canvas);

      for(int padNumber = 0; padNumber < 3; padNumber++){

        char name_for_pad[10];
        char title_for_pad[50];

        if(padNumber==0){
          sprintf(name_for_pad,"pad_c1_tube1minus2_cycle%d",cycle+1);
          sprintf(title_for_pad,"difference of tdrift vs time, tube(1-2)
cycle%d",cycle+1);
          pad_c1[padNumber][cycle] = new
TPad(name_for_pad,title_for_pad,0.01,0.51,0.49,0.99);  //  top left pad
        }
        if(padNumber==1){
          sprintf(name_for_pad,"pad_c1_tube1minus3_cycle%d",cycle+1);
          sprintf(title_for_pad,"difference of tdrift vs time, tube(1-3)
cycle%d",cycle+1);
          pad_c1[padNumber][cycle] = new
TPad(name_for_pad,title_for_pad,0.51,0.51,0.99,0.99);  //  top right pad
        }
        if(padNumber==2){
          sprintf(name_for_pad,"pad_c1_tube2minus3_cycle%d",cycle+1);
          sprintf(title_for_pad,"difference of tdrift vs time, tube(2-3)
cycle%d",cycle+1);
          pad_c1[padNumber][cycle] = new
TPad(name_for_pad,title_for_pad,0.01,0.01,0.49,0.49);  //  bot left pad
        }

        c1[cycle]->cd();
        pad_c1[padNumber][cycle]->Draw();
        pad_c1[padNumber][cycle]->cd();

        (graph_subtract_tdrift_vs_time[padNumber][cycle]->GetYaxis())->SetLimits(-13,13);
        (graph_subtract_tdrift_vs_time[padNumber][cycle]->GetYaxis())->UnZoom();
```

```
       (graph_subtract_tdrift_vs_time[padNumber][cycle]->GetYaxis())-
>SetTitle("difference in tdrift (ns)");
       (graph_subtract_tdrift_vs_time[padNumber][cycle]->GetXaxis())->SetTitle("time
(hours)");

       graph_subtract_tdrift_vs_time[padNumber][cycle]->Draw("AP");

       pad_c1[padNumber][cycle]->Modified();
       c1[cycle]->cd();

      }// ending padNumber loop for displaying images

     char output_ps_filename[70];
     char output_gif_filename[70];

sprintf(output_ps_filename,"immagini/tdrift_sub_vs_time/diff_of_tdrift_vs_time_cycle%d.ps
",cycle+1);

sprintf(output_gif_filename,"immagini/tdrift_sub_vs_time/diff_of_tdrift_vs_time_cycle%d.g
if",cycle+1);

     c1[cycle]->SetCanvasSize(950,712);
     c1[cycle]->SaveAs(output_ps_filename);
     c1[cycle]->SaveAs(output_gif_filename);

   }// ending cycle loop for displaying images

   cout << "We made cuts: " << endl;
   cout << "nEntries >= "<<cut_nEntries<<" && error <= "<<cut_error<<" ns"<<endl<<endl;
}
```

4.2 Appendix B

4.2.1 Macro 5

```
// This macro simulates the mdttdc spectra for various amounts of water.

#include <sstream>
#include <iostream>
#include <string>
#include "TH1.h"
#include "TFile.h"
#include "TChain.h"
#include "TTree.h"
#include "TDirectory.h"
#include "TNtuple.h"
#include "TF1.h"
#include "TCanvas.h"

TF1  *functy=0;
TF1  *funct4_000=0;
TF1  *functNoise=0;
TF1  *functGaus=0;
TF1  *funct_fit=0;
TF1*  funct_pars[5];
TH1F *hist_mdttdc;

void sim_water(int water_level=000, int nEvent=100000, int rand_seed = 192837465)
{
  gStyle->SetOptFit(1111);
  gRandom = new TRandom3(rand_seed);
```

```
  char function[40];

  //  Defining constants...
  float min_radius     = 0.062865;
  float trigger        = 513;
  float bit_correction = 0.78125;

  //  Defining histos and functs...
  hist_mdttdc    = new TH1F("hist_mdttdc","hist_mdttdc",2200,0,2200);
  functNoise     = new TF1("functNoise","1",0,2000);
  funct_fit      = new TF1("funct_fit","[0]+(([1]*(1+[2]*exp(-(x-[4])/[3])))/((1+exp((-
x+[4])/[6]))*(1+exp((x-[5])/[7]))))",0,2200);
  functy         = new TF1("functy","1",min_radius,1.4);

  funct_pars[0] = new TF1("funct_pars0","0.000445  *x + 0.034"  ,-1000,1000);
  funct_pars[1] = new TF1("funct_pars1","-0.019135 *x + 269.048",-1000,1000);
  funct_pars[2] = new TF1("funct_pars2","0.10092   *x - 274.528",-1000,1000);
  funct_pars[3] = new TF1("funct_pars3","-0.093225 *x + 480.013",-1000,1000);
  funct_pars[4] = new TF1("funct_pars4","0.03964   *x - 125.26" ,-1000,1000);

  float pars[5];
  float pars[0] = funct_pars[0]->Eval(water_level);
  float pars[1] = funct_pars[1]->Eval(water_level);
  float pars[2] = funct_pars[2]->Eval(water_level);
  float pars[3] = funct_pars[3]->Eval(water_level);
  float pars[4] = funct_pars[4]->Eval(water_level);

  for (int i=0; i<nEvent; i++ )
    {
      if (i%10000 == 0)  cout<<"Event #: "<< i <<endl;

      float y          = functy->GetRandom();
      float y_squared  = y*y;

      float mdttdc = pars[0] + pars[1]*y + pars[2]*y_squared + pars[3]*y*y_squared +
pars[4]*y_squared*y_squared;
      float resol  = 0.004*mdttdc + 8;
      sprintf(function,"exp(-(x*x)/(2*%f*%f))",resol,resol);
      functGaus        = new TF1("functGaus",function,-1000,1000);
      float randy      = functGaus->GetRandom();
      hist_mdttdc->Fill((mdttdc + randy + trigger)/bit_correction);
    }

  for (int k=0; k<int(nEvent*0.15); k++)  hist_mdttdc->Fill(functNoise->GetRandom());

  int file_part;
  cout << "which part of the file is this (1,2,3)? " << endl;
  cin >> file_part;
  char filename[120];
  if (file_part == 1){
    sprintf(filename,"files_combining/file%d_1.root",water_level);
    TFile *file_1 = new TFile(filename,"RECREATE","Description");
    hist_mdttdc->SetDirectory(file_1);
    file_1->Write();
  }
  if (file_part == 2){
    sprintf(filename,"files_combining/file%d_2.root",water_level);
    TFile *file_2 = new TFile(filename,"RECREATE","Description");
    hist_mdttdc->SetDirectory(file_2);
    file_2->Write();
  }
  if (file_part == 3){
    sprintf(filename,"files_combining/file%d_3.root",water_level);
    TFile *file_3 = new TFile(filename,"RECREATE","Description");
    hist_mdttdc->SetDirectory(file_3);
    file_3->Write();
  }
}
```

4.2.2 Macro 6

```
//  This macro takes all of the histos from the files_combining folder
//  and combines them, fits them, and plots a graph of delta_Tdrift vs
//  humidity.  It is mainly to show that there is a linear relation
//  between the two.


#include <sstream>
#include <iostream>
#include <string>
#include "TH1.h"
#include "TFile.h"
#include "TChain.h"
#include "TTree.h"
#include "TDirectory.h"
#include "TNtuple.h"
#include "TF1.h"
#include "TCanvas.h"

TH1F* hist_mdttdc_local[14]; // to differentiate from the histos that we are pulling out
of the files.
TF1 *funct_fit;
TGraphErrors *graph_tdrift_sub_vs_water;
float tdrift[14];
float tdrift_err[14];
float diff_tdrift[14][3];

void combining_all3()
{
  int water_levels[] = {0,15,30,45,60,75,90,105,120,135,150,165,180,200};
  char hist_name[40];

  for (int k = 0; k < 14; k++)
    {
      sprintf(hist_name,"hist_mdttdc_local_%d",water_levels[k]);
      hist_mdttdc_local[k]  = new TH1F(hist_name,hist_name,2200,0,2200);
    }

  TFile *file_1 = TFile::Open("files_combining/file0_1.root");
  hist_mdttdc_local[0]->Add(hist_mdttdc,1);
  TFile *file_2 = TFile::Open("files_combining/file0_2.root");
  hist_mdttdc_local[0]->Add(hist_mdttdc,1);

  TFile  *file15 = TFile::Open("files_combining/file15_1.root");
  hist_mdttdc_local[1]->Add(hist_mdttdc,1);
  TFile  *file15_2 = TFile::Open("files_combining/file15_2.root");
  hist_mdttdc_local[1]->Add(hist_mdttdc,1);

  TFile  *file30 = TFile::Open("files_combining/file30_1.root");
  hist_mdttdc_local[2]->Add(hist_mdttdc,1);
  TFile  *file30_2 = TFile::Open("files_combining/file30_2.root");
  hist_mdttdc_local[2]->Add(hist_mdttdc,1);

  TFile  *file45 = TFile::Open("files_combining/file45_1.root");
  hist_mdttdc_local[3]->Add(hist_mdttdc,1);
  TFile  *file45_2 = TFile::Open("files_combining/file45_2.root");
  hist_mdttdc_local[3]->Add(hist_mdttdc,1);

  TFile  *file60 = TFile::Open("files_combining/file60_1.root");
  hist_mdttdc_local[4]->Add(hist_mdttdc,1);
  TFile  *file60_2 = TFile::Open("files_combining/file60_2.root");
  hist_mdttdc_local[4]->Add(hist_mdttdc,1);

  TFile  *file75 = TFile::Open("files_combining/file75_1.root");
  hist_mdttdc_local[5]->Add(hist_mdttdc,1);
  TFile  *file75_2 = TFile::Open("files_combining/file75_2.root");
  hist_mdttdc_local[5]->Add(hist_mdttdc,1);

  TFile  *file90 = TFile::Open("files_combining/file90_1.root");
```

```
  hist_mdttdc_local[6]->Add(hist_mdttdc,1);
  TFile  *file90_2 = TFile::Open("files_combining/file90_2.root");
  hist_mdttdc_local[6]->Add(hist_mdttdc,1);

  TFile *file105 = TFile::Open("files_combining/file105_1.root");
  hist_mdttdc_local[7]->Add(hist_mdttdc,1);
  TFile *file105_2 = TFile::Open("files_combining/file105_2.root");
  hist_mdttdc_local[7]->Add(hist_mdttdc,1);

  TFile *file120 = TFile::Open("files_combining/file120_1.root");
  hist_mdttdc_local[8]->Add(hist_mdttdc,1);
  TFile *file120_2 = TFile::Open("files_combining/file120_2.root");
  hist_mdttdc_local[8]->Add(hist_mdttdc,1);

  TFile *file135 = TFile::Open("files_combining/file135_1.root");
  hist_mdttdc_local[9]->Add(hist_mdttdc,1);
  TFile *file135_2 = TFile::Open("files_combining/file135_2.root");
  hist_mdttdc_local[9]->Add(hist_mdttdc,1);

  TFile *file150 = TFile::Open("files_combining/file150_1.root");
  hist_mdttdc_local[10]->Add(hist_mdttdc,1);
  TFile *file150_2 = TFile::Open("files_combining/file150_2.root");
  hist_mdttdc_local[10]->Add(hist_mdttdc,1);

  TFile *file165 = TFile::Open("files_combining/file165_1.root");
  hist_mdttdc_local[11]->Add(hist_mdttdc,1);
  TFile *file165_2 = TFile::Open("files_combining/file165_2.root");
  hist_mdttdc_local[11]->Add(hist_mdttdc,1);

  TFile *file180 = TFile::Open("files_combining/file180_1.root");
  hist_mdttdc_local[12]->Add(hist_mdttdc,1);
  TFile *file180_2 = TFile::Open("files_combining/file180_2.root");
  hist_mdttdc_local[12]->Add(hist_mdttdc,1);

  TFile *file200 = TFile::Open("files_combining/file200_1.root");
  hist_mdttdc_local[13]->Add(hist_mdttdc,1);
  TFile *file200_2 = TFile::Open("files_combining/file200_2.root");
  hist_mdttdc_local[13]->Add(hist_mdttdc,1);

  funct_fit = new TF1("funct_fit","[0]+(([1]*(1+[2]*exp(-(x-[4])/[3])))/((1+exp((-
x+[4])/[6]))*(1+exp((x-[5])/[7]))))",0,2200);
  funct_line_000_200 = new TF1("funct_line_000_200","0.0683*x",0,210);
  gStyle->SetOptFit(1111);

  // Setting Par names, limits and fitting 000ppm...
  funct_fit->SetParName(0,"noise");
  funct_fit->SetParName(1,"amp");
  funct_fit->SetParName(2,"dip_amp");
  funct_fit->SetParName(3,"dip");
  funct_fit->SetParName(4,"t0");
  funct_fit->SetParName(5,"tmax");
  funct_fit->SetParName(6,"resol_t0");
  funct_fit->SetParName(7,"resol_tmax");
  funct_fit->SetParLimits( 0, 0,     500);
  funct_fit->SetParLimits( 1, 0.002, 22000);
  funct_fit->SetParLimits( 2, 0.085, 8500);
  funct_fit->SetParLimits( 3, 0.02,  20000);
  funct_fit->SetParLimits( 4, 550,   800);
  funct_fit->SetParLimits( 5, 1380,  1700);
  funct_fit->SetParLimits( 6, 0.5,   70);
  funct_fit->SetParLimits( 7, 0.5,   100);
  double pars_init[] = {0.22, 26, 8.5, 180, 660, 1480, 20, 22.4};

  graph_tdrift_sub_vs_water = new TGraphErrors(14);

  // Fitting the rest of the simulated mdttdc histos (30 - 200 ppm):
  for (int k = 0; k < 14; k++)
    {
      funct_fit->SetParameters(pars_init);
      hist_mdttdc_local[k]->Fit("funct_fit","NQ","",550,1650);
      tdrift[k] = (funct_fit->GetParameter(5))-(funct_fit->GetParameter(4));
```

```
      tdrift_err[k] = sqrt((funct_fit->GetParError(5))*(funct_fit->GetParError(5))-
(funct_fit->GetParError(4))*(funct_fit->GetParError(4)));

      diff_tdrift[k][0] = water_levels[k];
      diff_tdrift[k][1] = (tdrift[k]-tdrift[0]) *0.78125;
      diff_tdrift[k][2] = (tdrift_err[k]) *0.78125;
      //cout << water_levels[k] << " " << diff_tdrift[k][1] << endl;

      // Setting the points on the graph:
      graph_tdrift_sub_vs_water->SetPoint(k, water_levels[k], diff_tdrift[k][1]);
      graph_tdrift_sub_vs_water->SetPointError(k, 0, diff_tdrift[k][2]);

      cout << "k: " << k << " water_level: " << water_levels[k] << endl;
      cout << "     tdrift:      " << tdrift[k] << endl;
      cout << "     tdrift_err:  " << tdrift_err[k] << endl;
      cout << "     diff_tdrift: " << diff_tdrift[k][1] << endl << endl;
    }

  //for (int j=1; j<14; j++)  cout << diff_tdrift[j][1]-diff_tdrift[j-1][1] << endl;


  TCanvas *c1 = new TCanvas("c1","c1");
  graph_tdrift_sub_vs_water->Draw("AP");
  funct_line_000_200->SetLineColor(2);
  funct_line_000_200->Draw("same");
  graph_tdrift_sub_vs_water->Fit("pol1");

}
```

5. References

5.1 Works Cited

[1] Halzen, Francis, and Alan D. Martin. Quarks and Leptons: An Introductory Course in Modern Particle Physics. New York: John Wiley & Sons, Inc., 1984.

[2] Tipler, Paul A. Elementary Modern Physics. New York: Worth Publishers, Inc., 1992.

[3] http://pdg.lbl.gov/2002/contents_tables.html

[4] Fernow, Richard. Introduction to Experimental Particle Physics. New York: Cambridge University Press, 1986.

[5] Povh, Bogdan, et al., Particles and Nuclei: An Introduction to the Physical Concepts. New York: Springer, 2002.

[6] Leo, William R. Techniques for Nuclear and Particle Physics Experiments: A How-to Approach, 2nd Revised Edition. New York: Springer, 1994.

[7] http://www.phenix.bnl.gov/index.html

[8] http://www.phys.ufl.edu/~rfield/cdf/chgjet/etaphi.html

[9] http://atlas.web.cern.ch/Atlas/

[10] ATLAS Muon Collaboration. ATLAS Muon Spectrometer: Technical Design Report.Geneva: CERN/LHCC 97-22, 1997.

[11] http://physics.bu.edu/ATLAS/ATLAS-info/withframes.html

[12] ATLAS Collaboration. ATLAS Technical Proposal for a General-Purpose pp Experiment at the Large Hadron Collider at CERN. Geneva: CERN/LHCC 94-43, 1994.

[13] http://ppewww.ph.gla.ac.uk/preprints/1999/17/SaxonPSD5web.rtf

[14] *First Results of the 2001 MDT chambers beam test*, G. Avolio et al., CERN internal note: ATL-MUON-2003-001.

[15] http://www.geplastics.com/resins/global/pdf/design_guides/propertiessec2.pdf

[16] Kamin, Jason. Lifetime of Muon. PHYSICS 350 Advanced Physics Laboratory, Smith College with Piotr Decowski, 2002.

5.2 General References

[16] Carroll, Bradley W. and Dale A. Ostlie. An Introduction to Modern Astrophysics. New York: Addison-Wesley Publishing Company, Inc., 1996.

[17] Halliday, David, Robert Resnick, and Jearl Walker. Fundamentals of Physics 6th Edition. New York: John Wiley & Sons, Inc., 2001.

[18] http://garfield.web.cern.ch/garfield/

[19] K. Hagiwara *et al.,* Phys. Rev. D **66**, 010001 (2002)

[20] Taylor, John R. Introduction to Error Analysis. New York: University Science Books, 1997.