Modeling of Scintillation Dynamics

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science with Honors in
Physics from the College of William and Mary in Virginia

by

Andrew Hucke

Accepted for _____

_____

Advisor: Dr. Kane

_____

Dr. Hoatson

_____

Dr. Rublein

_____

Dr. Eckhause

The College of William and Mary

Williamsburg, Virginia

May 2003

Abstract

The model of scintillation in a polyvinyltoluene scintillator with imbedded wavelength-shifting fibers will be considered in this project. Events that occur during the scintillation process will be modeled as accurately as possible and a computer program in the C++ language that simulates these events will be created. The program will deal with the scintillation process as exactly as possible while maintaining fast run time. The program will follow each photon individually as it travels through the scintillator and create statistics based on the results. Experimental methods will be used to test and refine my program, and features will be added to the program that experimentation determines are necessary. The program will generate output data distributions such as the number of bounces off of a wall of the scintillator, the path length traveled by the photon, and the percentage of photons being detected in the photomultiplier tube. This model will benefit researchers by demonstrating the practical application of modeling tools to aid in scintillator selection.

Acknowledgments

I would like to thank my advisor Professor Kane for the time and effort he put into helping me on my thesis and the invaluable suggestions he offered. I would also like to thank Professor Zobin for helping me with some of the more technical geometrical principles.

Contents

List of Figures

List of Tables

1 Introduction

My project is to write a computer program to model muon detection by a scintillator. This is a complicated and involved process that will require consideration of many aspects of scintillators. Scintillators have been used to detect cosmic rays in particle and nuclear physics experiments for over sixty years. Cosmic rays are high energy particles emitted by stars and supernovae. When these particles enter the atmosphere, they usually collide with the nuclei of O_2 or N_2, sending a shower of pions, kaons, and nucleons toward the surface of the earth [1]. These reactions are shown by the equations:

$$pp \rightarrow pp\pi^0 \quad (1)$$

$$pp \rightarrow pn\pi^+ \quad (2)$$

$$pn \rightarrow pn\pi^0 \quad (3)$$

$$pn \rightarrow pp\pi^- \quad (4)$$

The charged pions have a proper lifetime of twenty-six nanoseconds and quickly decay into a positive (negative) muon and a muon neutrino (antineutrino).

$$\pi^+ \rightarrow \mu^+ \nu_\mu \quad (5)$$

$$\pi^- \rightarrow \mu^- \bar{\nu}_\mu \quad (6)$$

This project will focus on cosmic ray muons, which make up seventy-five percent of the cosmic particles at the earth's surface. Scintillators are useful detecting devices, because they are sensitive to the energy loss of charged particles that pass through and have a fast

time response [2]. This allows scintillators to accurately detect cosmic ray incidence, as needed in the MECO project. MECO is an experiment to test lepton flavor violation in Muon to Electron Conversions [3]. The layout for the MECO experiment is given in Fig. 1.

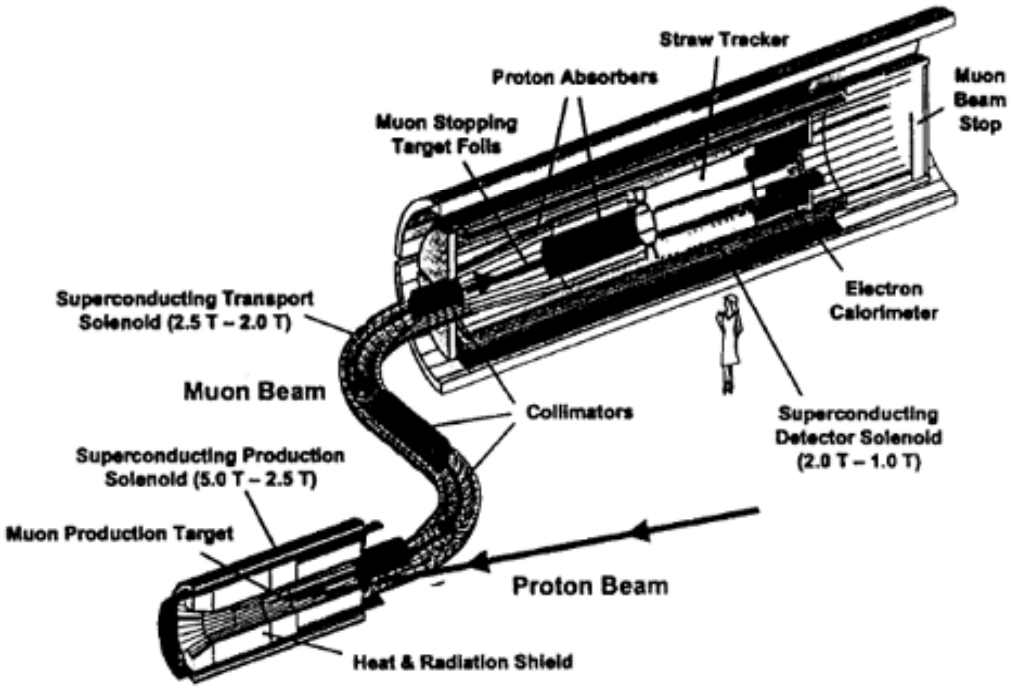

Figure 1: The layout for the MECO experiment.

The experiment utilizes a proton beam incident on a target producing low momentum pions. The pions are caught in helical orbits due to the magnetic field surrounding the target. The field's gradient also increases away from the large apparatus for the purpose of catching some of the pions that are headed out the back of the apparatus. When the

pions decay in the magnetic field the resulting muons have less energy then the pions and as a result are also caught in the magnetic field. The predicted intensity of the negative muon beam is 10^{11} muons per second, which is one thousand times larger then any previous muon beam. The resulting muons are transported via a superconducting solenoid and the magnetic field gradient to the muon stopping target foils. The muons then form muonic atoms inside the target and either decay from the 1s orbit into two neutrinos and an electron or capture on a target nucleus. These electrons have maximum energy of 53 MeV; however if lepton flavor is violated at the nucleus, a small number of electrons will have an energy of 105 MeV. This is due to the fact that in a lepton flavor violation the muon decays into an electron instead of an electron and two neutrinos. The presence of the two neutrinos limits the energy of the electron at 53 MeV, however if they are not present the energy of the electron is 105 MeV instead. The detectors are designed so that the 53 MeV electrons spiral around in the field in such a manner that they miss the detector. The 105 MeV electrons caused by the violation of lepton number would have a radius large enough to be detected, however. In the MECO project scintillators are used to veto events caused by cosmic particles. It is possible for a cosmic muon to interact with the stopping target to produce an electron of 105 MeV. This occurrence would be very rare, since the electron would have to have the momentum in the correct direction; however the number of occurrences of this is predicted to be about 10 during the lifetime of the experiment. Since, a sensitivity of $6 * 10^{-17}$ is required in this experiment, not being able to veto these erroneous events will set a background limit below which the experiment lacks sensitivity.

MINOS is another project where scintillators play a large role. MINOS is a search for neutrino oscillations [4]. This experiment is searching for two specific oscillations.

$$\nu_\mu \rightarrow \nu_\tau \quad (6)$$

$$\nu_\mu \rightarrow \nu_e \quad (7)$$

The experiment utilizes a near and far detector, the near detector being 290 meters from the decay pipe and the far detector being 730 km from the decay pipe. The near detector is designed to detect the neutrinos present in the beam before many of them have a chance to undergo oscillations. The far detector will detect the neutrinos after they have had a chance to oscillate. Neutrinos will be produced by a 120 GeV proton beam with an intensity of $4*10^{13}$ protons per pulse. When impacting a graphite target cylinder, these protons then produce neutrinos in the same manner as described for cosmic ray muons. The detector is a steel plate that the neutrinos will interact with to produce an electron which will be detected by the scintillator plates surrounding the steel target. MINOS tests the efficiency of their scintillators by testing the response of different arrangements to cosmic rays, such as muons. My program would allow simple experimentation with new ideas for scintillator designs. If the design performs well in a simulation then the experimenters would know that it would be advantageous to test that configuration experimentally. In addition, both MECO and MINOS make use of the same TiO_2 coating that I use in my simulation. Both experiments are also using parts similar to Bicron products that I will be using as defaults for several variables in my program. This makes

my program potentially valuable to both of these experiments. Despite the frequent use of scintillators few, if any, computer programs have been developed to predict the efficiency or model the behavior of scintillator configurations read out by embedded wave-shifting fibers. A program that has been developed for more basic scintillator-only simulations is GuideIt [5]. These programs can be highly useful, because scintillator materials are expensive and time is limited. While experimental research and observation is necessary; it is impossible to test all situations this way. The program allows easy testing of the possible configurations that don't have enough basis for experimental consideration and allows for comparison to the other methods being considered.

The basic scintillator configuration consists of a block of transparent material, in this case polyvinyltoluene, doped with the optically-active chemicals PPO, 2,5-Diphenyloxazole, (1%) and POPOP, 5-phenyloxazole, (.03%), in which fluorescence[1] is produced by an ionizing particle as shown in Fig. 2[6].

[1] Defined as the emission of optical photons in a substance stimulated by incident radiation.

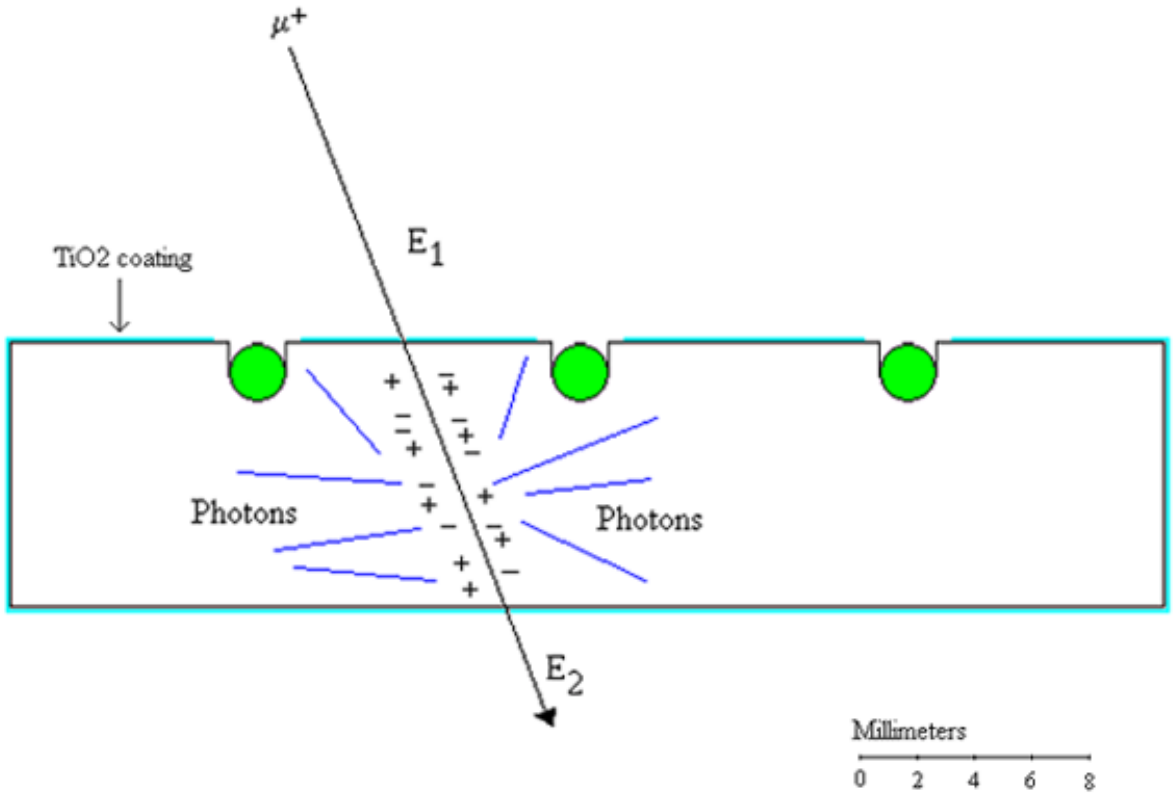

Figure 2: A cosmic ray muon depositing energy in a block of scintillator material and the resulting photons. The green circles represent an end-view of wavelength-shifting fibers, the blue lines are the photons, the light blue outline is the paint, and the plus and minus signs are the ions.

The scintillator design considered by MECO and MINOS involves very long scintillator strips and makes use of a TiO_2 coating that reflects nearly all the photon energy from the surfaces of the polyvinylstyrene as well as wave-shifting fibers. These fibers shift the wavelengths of the photons, from a band centered around 400 nanometers to one around

500 nanometers, and direct them to a Hamamatsu photomultiplier tube.[2] The photomultiplier tube detects the photons with a certain photoelectric efficiency. Many of the relevant values for aspects of the scintillator design, such as the TiO_2 paint, and attenuation of the wave-shifting fibers, must be determined experimentally. This creates a need to design and carry out accurate experiments to determine these relevant physical properties. Because the program is complex and has many stages an overview of the program is necessary to fully understand its many features and implications.

2 Program Overview

My program will accurately show the behavior of a photon in a scintillator through computational simulation. The user will be able to specify the dimensions of the scintillator as well as its emission and absorption properties. This program is not a small undertaking; the code itself is about sixty pages. The program will be written in the C++ programming language, which is a very powerful and versatile language. The program is complex and has many different parts or modules that are diagrammed in Fig. 3.

[2] In this case the photons emitted in the fluorescence are blue (409 nm) and the shifted photons are green (480 nm).

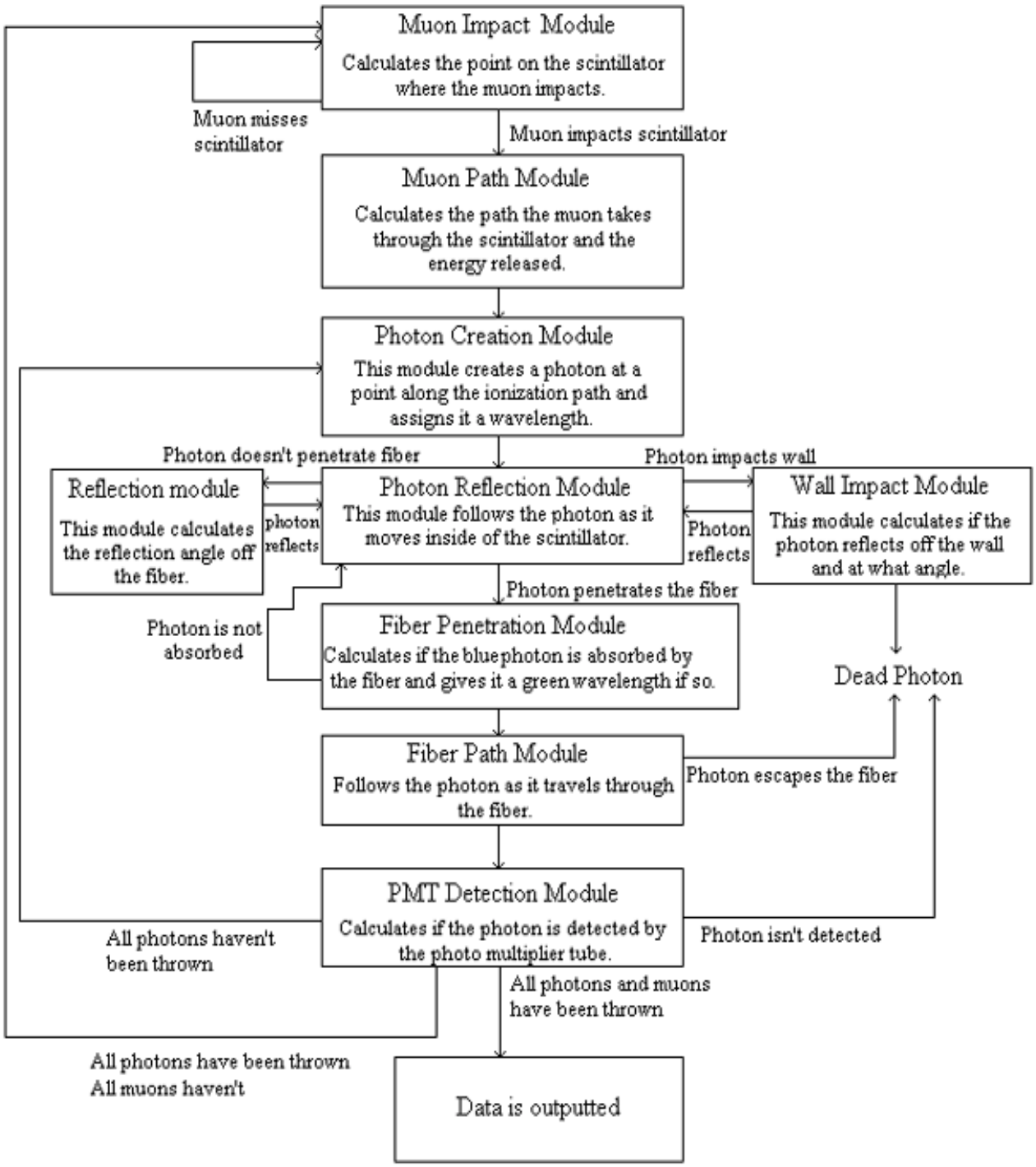

Figure 3: The flow chart describes the basic program progression as well as the connections between the many different modules.

The program starts with the Muon Impact Module. This module tracks a muon as it is incident on the block of scintillating material, the size of which is defined by the user, and records the position of impact if an impact occurs. This position of impact is then passed on to the Muon Path Module. The Muon Path Module tracks the path of the muon as it passes through the block of scintillating material. After the path of the muon inside the scintillator is calculated it will be used to calculate the number of photons produced by the impact via experimental data. This data as well as the path length and the location of the path are passed from the Muon Path Module to the Photon Creation Module. The Photon Creation Module will generate photons with a random direction vector in three dimensions to approximate uniform, isotropic emission along the muon path. The module then uses the absorption spectrum of the scintillator to calculate the wavelength of a particular photon emitted by fluorescence. This spectrum will have a default value which can also be defined by the user. The directions of these photons as well as their point of origin are passed on to the Photon Reflection Module. This module is responsible for the tracking of the photons inside the scintillator. The module must take into account the positions of the fibers, the locations of the walls and the angles of impact of the photons. In addition, a running distribution for the total distance the photon has traveled will begin at this point. This distance distribution of photon trials will be used later to calculate the time delay distribution between scintillation and detection. In addition, the number of scintillator photons produced and the number of wave-shifted photons detected will be used to calculate the percentage of photons reaching the detector. When the module determines that a photon has impacted a wall the module ends and sends the photon location and direction to the Wall Impact Module. This

module will apply the appropriate fit for reflection versus absorption given in Fig. 10 and determine the angle at which the photon reflects off of the wall. If the photon is absorbed then the module returns a "dead" photon; if the photon is reflected it returns the resultant direction vectors to the Photon Reflection Module. In addition, a count of how many bounces off the wall the photon undergoes will be kept by the Photon Reflection Module. A photon will bounce until it hits a wave-shifting fiber unless it has been absorbed in a wall bounce. If a photon hits a wave-shifting fiber its location as well as direction vector will be passed onto the Reflection Module. The positions of the wave-shifting fibers, which can be specified by the user, are used to calculate the angle of impact of the photon. This angle will be analyzed to see if the photon successfully penetrates the lower index of the fiber cladding. If the photon penetrates the fiber the Reflection Module sends the photon direction and position to the Fiber Penetration Module. If the photon reflects off the fiber the Reflection Module sends the photon direction and position back to the Photon Reflection Module. The Fiber Penetration Module determines the path the photon travels and also calculates if the photon is absorbed by the fiber after traveling a small distance. The Fiber Penetration Module will use a direction vector identical to the vector used by the Fiber Reflection Module as a small approximation. Since there is a .02 difference between the index of refraction of the scintillator and the fiber, and since this will not significantly change the path length of the photon we can make this approximation without much loss of accuracy. The chance of being absorbed depends on the wavelength of the photon and can also be specified by the user for a different fiber. If the photon is absorbed its absorption position will be passed to the Fiber Path Module and it will be reemitted, at the same location, in a random direction as a green photon.

The Fiber Path Module will determine the exact wavelength of this green photon using the spectrum for emission from the fiber by utilizing the fit to the emission spectrum given in Fig. 16. The number of bounces off the walls of the scintillator variable will stop counting for a specific photon at this point in the simulation. An example of the behavior of the photon inside the wave-shifting fiber is given in Fig. 4.

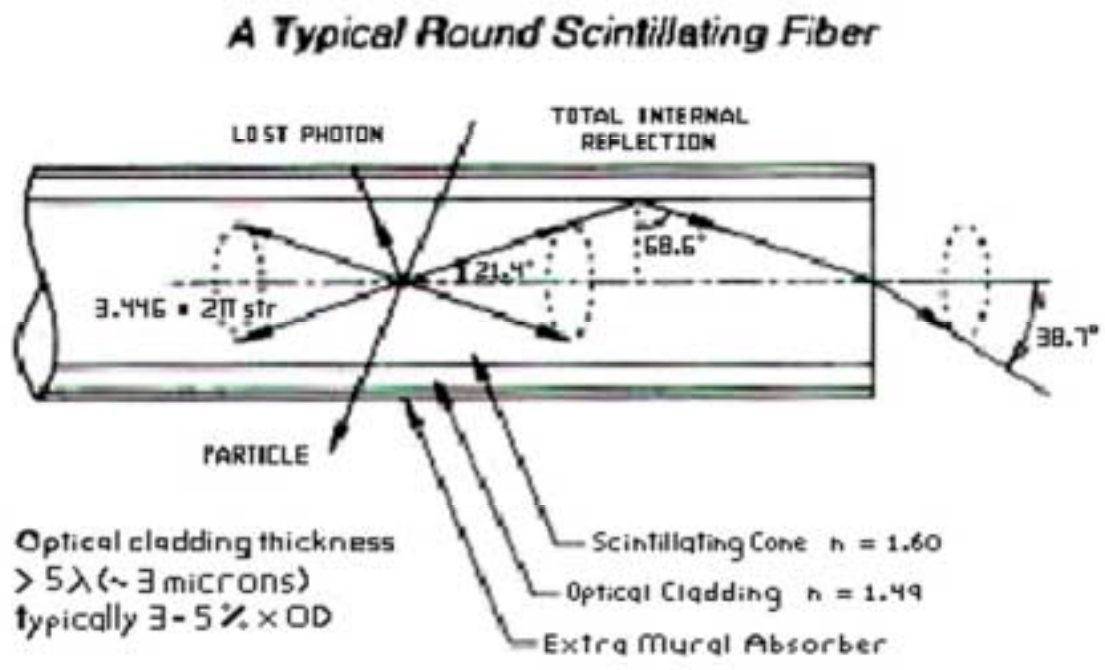

Figure 4: A diagram of photon behavior after emission by the wave-shifting fiber. The cones in this picture represent the critical angle of 68.6 degrees, below which the particle will undergo total internal reflection. In addition, the reflection angles and the outer cladding of the fiber are also shown.

If the photon contacts the walls of the fiber within a certain angle it will be totally internally reflected inside the fiber until it is either self-absorbed, resulting in a dead

photon and the frequency of which can be predicted by equation 28, or reaches one of the fiber ends. If the photon escapes the fiber it will have no practical chance of being detected by the detector and is therefore counted as a dead photon. If the photon is self-absorbed, it will also be counted as a dead photon. The Fiber Path Module also assumes one end of the fiber is painted with a black paint to eliminate photons that travel in that direction. This is to prevent the delayed signal that would occur if the photons instead reflected. Only photons traveling toward the detector end of the fiber will be counted; the others will be counted as dead photons. Once the photon reaches the detector the Fiber Path Module will then pass the photon's wavelength to the PMT Detection Module which uses the PMT quantum efficiency curve to determine if the photon is detected. The length of the photon's travel will end when the photon reaches the detector. The program will then return to the Photon Creation Module and throw another photon until all photons produced by the muon have been tracked or the user terminates the simulation. Once all the photons have been thrown another muon will be thrown and the process will repeat. The raw output will be the length of the photon's travel, the number of reflections off the walls of the scintillator, and the percentage of photons reaching the detector.

3 Theory

3.1 Random Number Generation

A major aspect of my program is the use of random numbers to make decisions on photon wavelength, surface reflections, and absorption and emission by the fibers. As a result of this, good random numbers are important to my project. While computers can't produce true random numbers they can produce a sequence of pseudo random

numbers with a long period [7]. In addition, a good random number algorithm will not degenerate into a short repeating sequence of numbers. The C++ compiler uses a random number generation method similar to a special case of the Linear Congruential Method [7]. This method makes use of the formula

$$\begin{aligned} &X_{n+1} = (a * X_n + c) \bmod (m) \\ &m : 0 < m \\ &a : 0 \le a < m \\ &c : 0 \le c < m \\ &X_0 : 0 \le X_0 < m \end{aligned} \tag{8}$$

where m is the modulus, a is the multiplier, c is the increment, and X_0 is the starting value. To produce a good random number generator using this method appropriate starting values for m, a, c, and X_0 must be chosen. According to Knuth the optimal value for m is the word size of the computer [7] which is stated as about 2^{32} for an IBM machine. Choosing a correct value of the modulus is important because an inconvenient value will greatly increase processing time, whereas a convenient value such as the word size of the computer will make calculations much faster. It is very likely that the C++ knows the word size of the computer, since it is a Microsoft specific version; giving it a very good value for the modulus m. In addition, the multiplier, a, must be chosen so the period is as long as possible while maintaining randomness. According to Knuth for a computer with a word size of 2^{32}, the best value for a is 3 or 5 (modulo 8) [7]. This will result in a random sequence with a period of .25m. This period is just about one billion elements long before it starts repeating itself which is more then sufficient for this program since, even in long simulations, it is unlikely much more then one hundred million random numbers will need to be generated. Within this context many values of c

and X_0 will produce satisfactory results and there are probably further theories that determine which values are best. However, the basic period with a = 3, 5 (mod(8)) and $m = 2^{32}$ is long enough to be satisfactory in our application. Since the C++ compiler makes use of a process similar to this method; I will use the random numbers generated by the random() function in the C++ library to generate random numbers.

3.2 Simulation of Particle Movement

One of the most important issues in doing particle simulations is accurately simulating the transport of the particle without using up too many system resources. Most computers nowadays have impressive clock speeds of over one Gigahertz; however operations can still take hundreds of nanoseconds for the computer to evaluate. When you consider a loop which easily contains ten thousand of these calculations, the speed has dropped to milliseconds. As a result it is important to design loops that can be computed as fast as possible without sacrificing much accuracy. To this end my program's movement simulation checks the condition of each photon after a certain path length has been covered as shown in Fig. 5. A single increment of path length represents 1.25 mm. The photon is then given x, y, and z speeds based on these blocks.

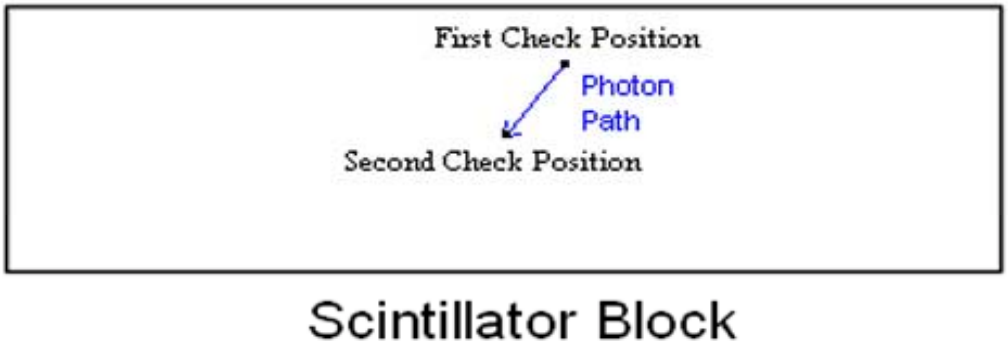

Figure 5: The movement of the photon after a position check has been run to the point at which the next position check will be run. The program does not know what occurs between the two checks.

As a result the loop controlling the photon movement only has to run a maximum of a few hundred times at the longest. This greatly increases calculation speed but could easily impact the accuracy. For example, what if the photon was moved so it ended up outside the scintillator before the program did a check to see if it was at the boundary? My solution to this problem was to do a correction to the location of the photon. Once a photon escapes, the program calculates at what point that photon had impacted the wall of the scintillator and moves the photon back to that position. This result is also taken into account in the photon's total travel distance. This is a reasonable approximation and still allows the user to place objects at each eighth centimeter block in the scintillator, such as wave-shifting fibers, with no difficulty. Calculations for the incident muons are similar yet much simpler. Since the muon must have a velocity in the z direction it is automatically given a constant value. The angle of the muon's path is controlled entirely

by the x and y velocities. This allows an easy simulation of muon movement inside the scintillator with error checking only necessary in two directions. In addition to this, initial impact of the muon requires no error checking at all, simply a check to see if the muon has impacted the scintillator after each cycle of the movement simulation loop. This simulation method should be more then sufficient to produce quick, accurate simulations of particle movement; if this is not the case more short cuts can still be implemented with only a small loss on simulation accuracy.

3.3 Normalization of Curves

This experiment uses a lot of graphical data, most of which is in the form of a Bell-Shaped curve. These plots are not normalized; however I will be using a method to transform them into probabilities for use in my simulations. First, points on the curve will be plotted in the curve-fitting program Kaleidagraph [7a]. Then a high order polynomial, up to ten orders, will be fit to these points until a satisfactory fit to the original graph is obtained. At that point the graph generated by the polynomials will be integrated over between a starting and ending point that includes as much of the area of the original curve as possible. The curve generated by the polynomial will then be divided by this value to produce a probability curve. This curve will then be used in my program whenever the corresponding probability needs to be calculated. The reason the curve must be normalized is to assure that the maximum value for the curve will be less then one. We can be assured that there will not be a spike significantly exceeding one because the emission and absorption curves that this method will be used for are spread over many wavelengths. The maximum value needs to be normalized to unity because if

the maximum value is significantly less then the upper bound on the random number generation for the y coordinate, the program will have to generate significantly more photons before one gets under the area of the curve. This would greatly hamper the running speed of the program. In addition if the maximum of the curve was greater then the upper bound, the curve will not be represented by the program properly.

3.4 Incident Particles

The incident particles that will be dealt with in this experiment are cosmic ray muons. A muon passing through the scintillator will deposit a certain amount of energy per unit length in the scintillator material. Using an energy loss formula, this energy can be accurately predicted, based on the amount of scintillator the particle has traversed [8]. The data for the ionization energy of a muon passing through a certain thickness of material, as given by reference 8, shows that a minimum amount of energy will be deposited in a length of scintillator via ionization of the atoms along that path. Using a minimum is useful because it shows how effective the scintillator will be with the least number of photons incident on the PMT. As a result actual efficiencies can be expected to be higher then what the program calculates. This energy will then be converted into photons, as the electrons fall back into their ground states, with an energy loss of about 100 electron volts per photon [9]. The minimum amount of energy deposited by a muon

is 1.78 MeV per gram per centimeter squared as given by Lawerence [8]. This result can also be calculated from the Bethe-Bloch formula [2].[3]

$$-\frac{dE}{dx} = 2\pi n_a r_e^2 m_e c^2 \rho \frac{Z}{A}\frac{z^2}{\beta^2}\left[\ln\left(\frac{2m_e\gamma^2 u^2 W_{max}}{I^2}\right) - 2\beta^2\right] \quad (9)$$

in which $\frac{dE}{dx}$ is the differential change in the energy after the particle travels a short length dx through the scintillator in MeV per centimeter. In addition, n_a is Avogadro's number, r_e is the classical electron radius, m_e is the mass of the electron, c is the speed of light, ρ is the density of the absorbing material, Z is the atomic number of the absorbing material, A is the atomic weight of the absorbing material, z is the charge of incident particle in units of e, β is the velocity of the particle divided by the speed of light, γ is the inverse of the square root of $(1 - \beta^2)$, u is the velocity of the incident particle, I is the mean excitation potential, and W_{max} is the maximum energy transfer in a single collision. The Bethe-Bloch formula is not limited to minimum energy, is heavily based on relativistic formulae, and is appropriate for objects with high energy. A muon is the particle that is assumed incident when using this formula since my program doesn't deal with any other particles. Since only a certain number of blue photons are created for a value of deposited energy, the efficiency of this process must be calculated. This can easily be done using the formulae:

$$N_{photons} = \frac{\Delta E_{min} * 1 * 10^6}{\varepsilon} \quad (10)$$

[3] See Leo for a more in-depth description of the Bethe-Bloch formula.

$$\Delta E_{min} = (\frac{dE}{dt})_{min} * h * \rho \quad (11)$$

in which N is the number of photons released, ε is the efficiency for converting ionization energy to photons, ΔE is the deposited energy, h is the path length, and ρ is the density of the scintillator material. The value for ΔE that will be used in this experiment is

$1.837*10^{6} * h$ eV. This value for ΔE is calculated by multiplying together everything that is known for polyvinyltoluene and writing ΔE in terms of the unknown h. The value of N for polyvinyltoluene is given by the manufacturer as 68 percent of anthracene. As a result ε can be calculated for polyvinyltoluene by the equation:

$$\frac{N_{photons\ anth}}{N_{photons\ PVT}} = \frac{\varepsilon_{anth}}{.68\varepsilon_{anth}} \quad (12)$$

$$\varepsilon_{PVT} = .68\varepsilon_{anth} \quad (13)$$

The result is 100 electron volts per photon, as compared to anthracene, which has 68 electron volts per photon [2]. The number of photons produced will be calculated and each photon will be simulated one at a time. This is doable in a reasonable amount of time due to the speed of the current computer processors. A distribution of the wavelengths produced by the conversion of energy to photons for polyvinyltoluene is given by the manufacturer in Fig. 6 [10].

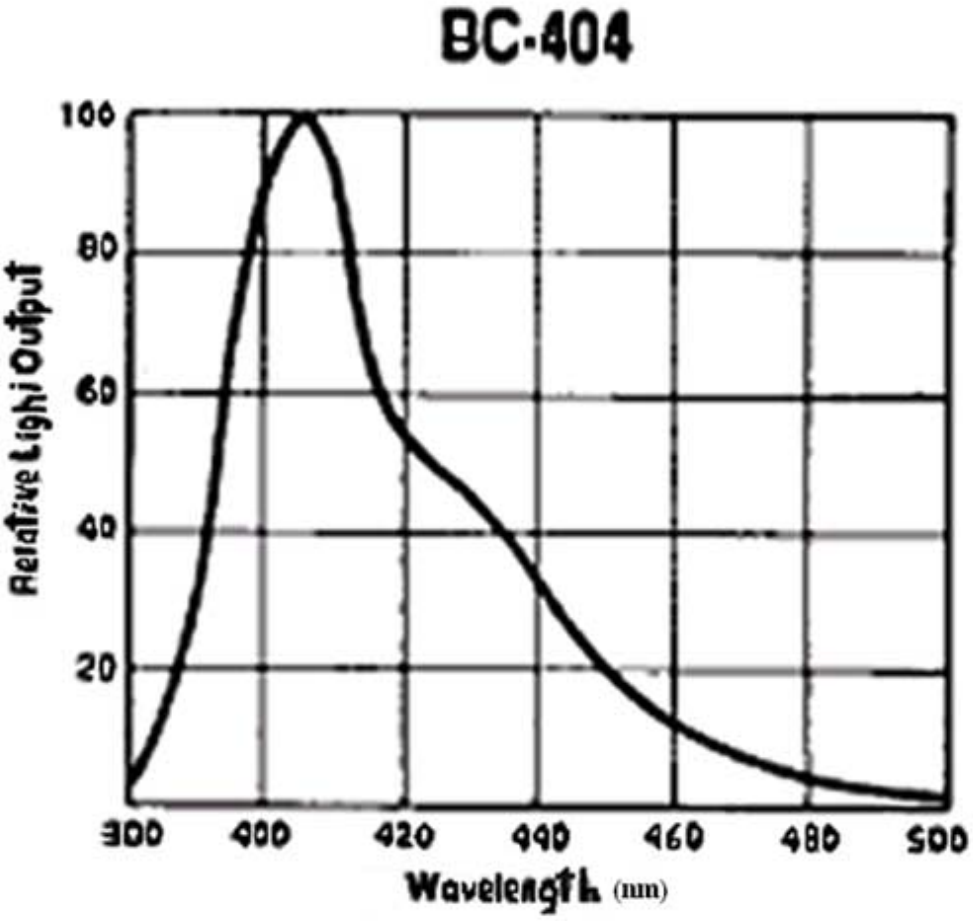

Figure 6: The relative light output plotted against the wavelength for Polyvinyltoluene. The wavelengths of the photons emitted are sharply peaked around 410 nanometers.

Though this curve peaks fairly sharply at about 410 nanometers, there is still a good deal of area corresponding to wavelengths in the 300s. This will become a concern because the reflectivity of the TiO_2 paint will drop off markedly below 400 nanometers. I fit a curve to this graph using Kaleidagraph and use the curve as a default value for the emission spectrum of the scintillator in my program. I normalized the fit by dividing by the maximum value of the fit. As a result I have a curve with a maximum of one. This is

important because of how the program determines random numbers for the purposes of assigning weighted photon wavelength. If the area above the curve is large then the program will have to throw significantly more random numbers before it gets one under the curve, leading to a large increase in processing time. Since no curve resembles a delta function it is then safe to assume that the maximum value will be finite. As a result I can set my upper limit for throwing random y values at one and as a result, make the program more efficient. It is much better for the program speed to ask the user to normalize their curves, by dividing by the maximum value, instead of trying to guess at a value that would fit everyone's curves. The fit I used to approximate the curve in Fig. 6 is given in Fig 7.

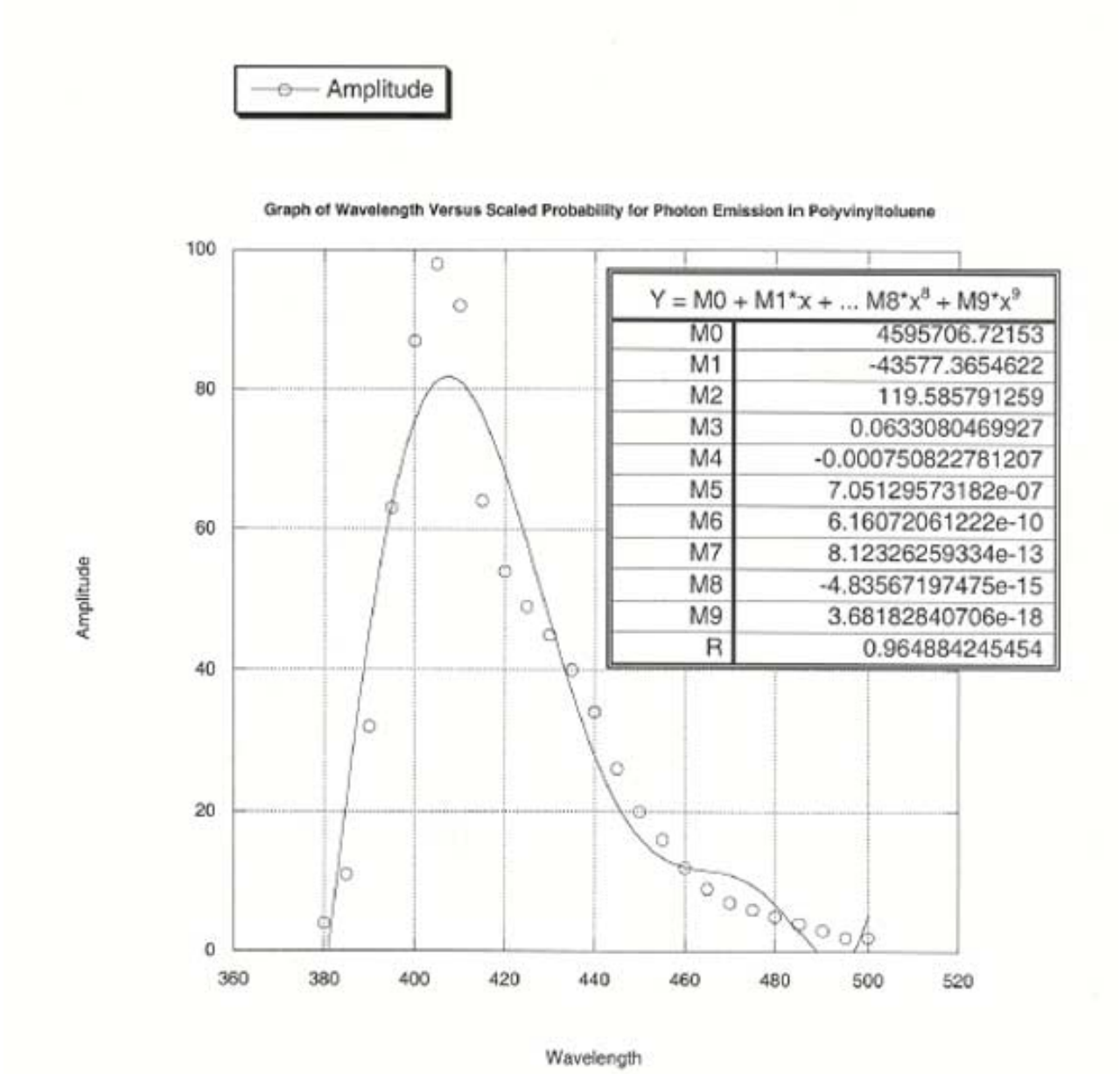

Figure 7: This is a graph of the wavelength versus a scaled probability for scintillator emission. The solid line is the fit and the points are points on the scintillator emission distribution.

This graph would be cut off at the values 382 and 488 providing an area that approximates the curve to a reasonable degree. The curve doesn't accurately fit the

uppermost three points, and accuracy can be improved using a piecewise fit. The problem with doing this, however is that it would be hard to allow the user to enter a piecewise fit if the user wanted to change the spectrum of the scintillator. It is much easier to approximate it as a higher order polynomial and allow the user to define a different fit using a polynomial as well due to the versatility of polynomials.

3.5 Reflection inside the scintillator

The outside of the scintillator will be coated with TiO_2 paint except for the fiber grooves, to keep the blue photons inside the scintillator as long as possible until they are absorbed by a wave shifting fiber. This paint is composed of many tiny particles of TiO_2 and we speculate it will behave like a rough surface. It is suggested that photons encountering rough surfaces undergoing random reflection can still have reflection patterns that follow formulae [9]. The roughness of the surface plays a large part in determining how the particle will reflect. Lekner suggests that the rougher the surface, the more diffuse the reflection [11]. In addition, Stover suggests that the measure of roughness is related to how much greater the average surface-height deviations are than the wavelength of the incident particle [12]. Stover also provides a plot showing scattering from an isotropic surface.[4]

I have experimentally determined the scattering distribution of the TiO_2 paint by hitting the paint with a blue light of known wavelength close to the peak wavelength emitted by the scintillator. By using a mercury source and shining it through a small slit and then an angled diffraction grating I was able to spread the spectral lines out. This allows me to

[4] An isotropic surface is one that would measure the same average roughness in all directions.

easily select the 408 nanometer line I want to use for the experiment. After the line is isolated from the others via a slit, it is reflected off a front surface mirror onto the paint. The reflection of the light off of the paint can be detected using a rotating detector. More details on this experiment are given in appendix A. A distribution of reflected light was taken over the different angles normal to the plane of the paint surface. This pattern represents a Gaussian distribution. After doing experiments to determine the characteristics of this Gaussian, I found that the Gaussian portion of the reflection from the paint is minute. I will be approximating reflections from the paint by assuming the photon has an equal probability of reflecting at any angle for simplicity. This will be done by throwing random values for the sine and cosine of the angle and using them as the change in direction. If we assume we have a randomly generated angle θ and φ we can use the equations

$$\sin(\theta) = \nu_i \tag{14}$$

$$\cos(\theta) = \nu_j \tag{15}$$

$$\cos(\varphi) = \nu_k \tag{16}$$

where ν_i, ν_j, ν_k are the direction vectors for each Cartesian coordinate. Sine is not included for φ because it is the same as cosine for θ.

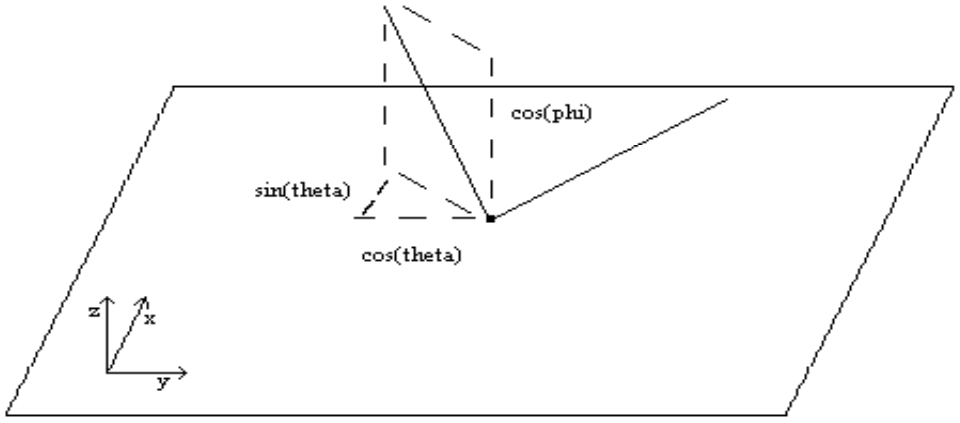

Figure 8: This figure shows how the Cartesian cords of the reflected velocities will be calculated from the random angles thrown by the program.

Random numbers will be used to determine the actual angle the photon leaves the surface. In addition to the formula for reflections off the reflector coated inner walls of the scintillator, a formula that represents the efficiency of the reflections is also necessary. Some of the photons are lost through absorption by this surface [9]. A curve representing this is given by Bicron and is available on their website [13]. See Fig. 9.

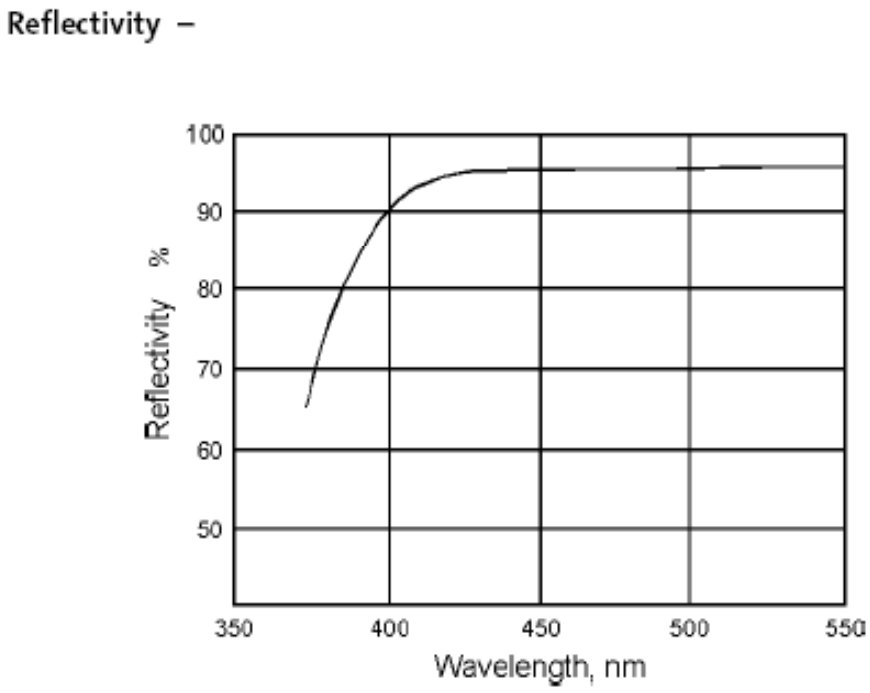

Figure 9: The wavelength plotted against the percentage of light reflected from the TiO_2 paint. The graph begins to fall off drastically below 400 nm.

A curve was fit to this curve using Kaleidagraph and the result will be used to determine the probability of a reflection when the photon impacts the paint. The curve that Kaleidagraph fits to Fig. 9 is given in Fig. 10.

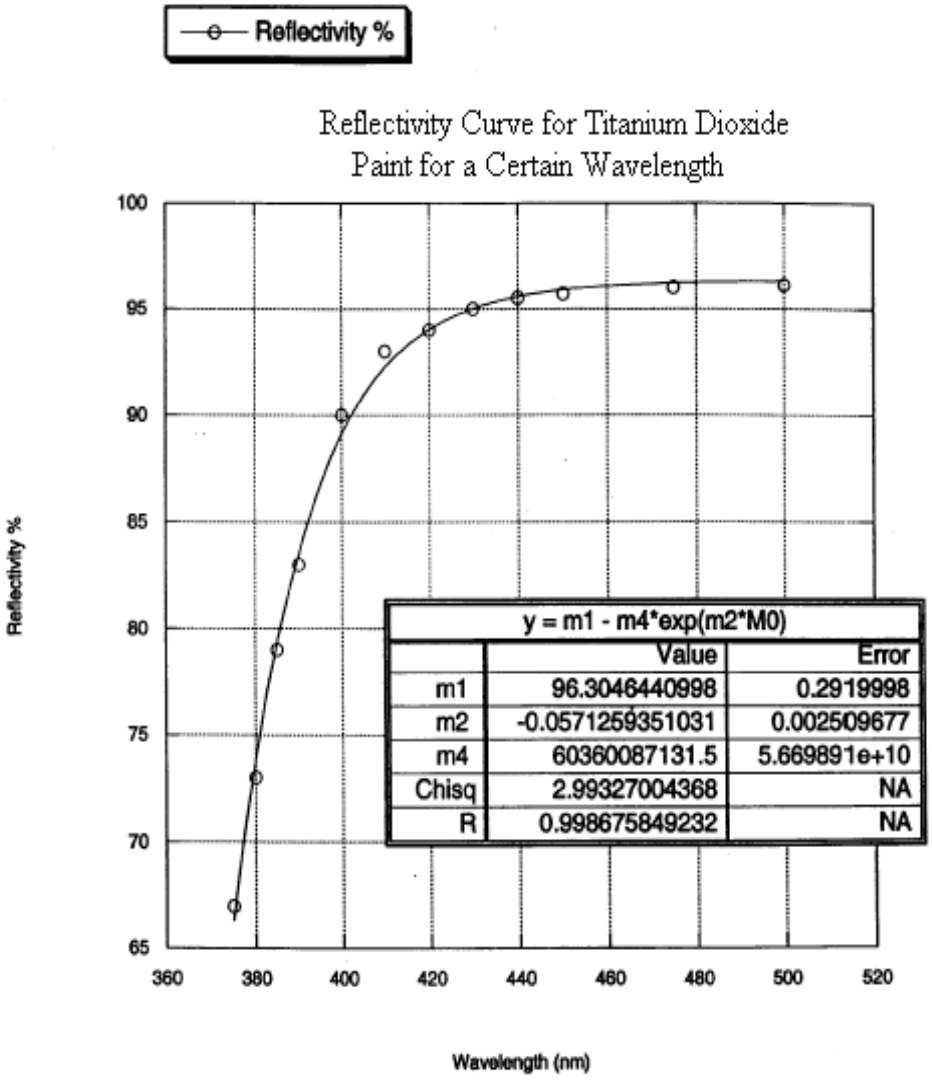

Figure 10: The fit given by Kaleidagraph to the reflectivity curve plotted versus wavelength for the TiO_2 paint.

Every time a particle impacts the coating it will be given a new random direction vector.

The angles θ and φ defining the solid angles, $-\pi/2 < \theta < \pi/2$, $0 < \varphi < 2\pi$; are randomly generated by my program. The random angles will be defined by using a common Monte Carlo method. A normalized function will be plotted and then a box with a height of one and width equal to a value designated by the user will be drawn overtop of the function as shown in Fig. 11. This method is a good way to generate random x values that are weighted by the shape of the curve.

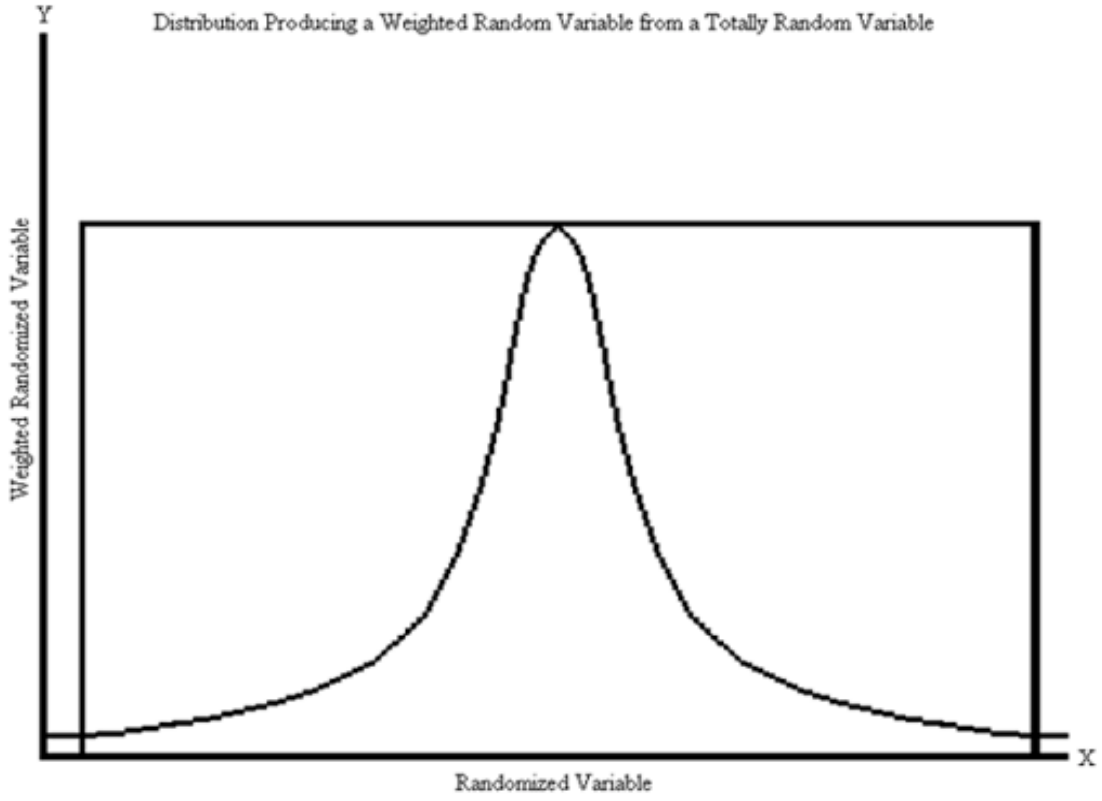

Figure 11: A rough outline of how a box can be put on top of a Gaussian, or any other, curve to allow random numbers to represent a random weighted distribution.

The computer will then generate random values for θ and φ. If these values fall within the area of the Gaussian, they will become the new reflection angles for the photon. If these values fall outside the area of the Gaussian, they will be eliminated and new values

will be generated. The photon will be allowed to reflect inside the scintillator until it penetrates the TiO_2 coating or it is successfully absorbed into a wave-shifting fiber. The box technique will also be used to weight random wavelengths generated and assigned to the photons when they are emitted from the scintillator and emitted from the wave-shifting fiber. The interface between the wavelength-shifting fiber and the scintillator is a standard Snell's Law boundary. The angle of impact of the particle and the walls of the fiber must be calculated. This is done by using a vector pointing from the midpoint of the circle to the point of impact of the particle. In addition, the direction vector of the photon will be used. A diagram of this geometry is given in Fig. 12.

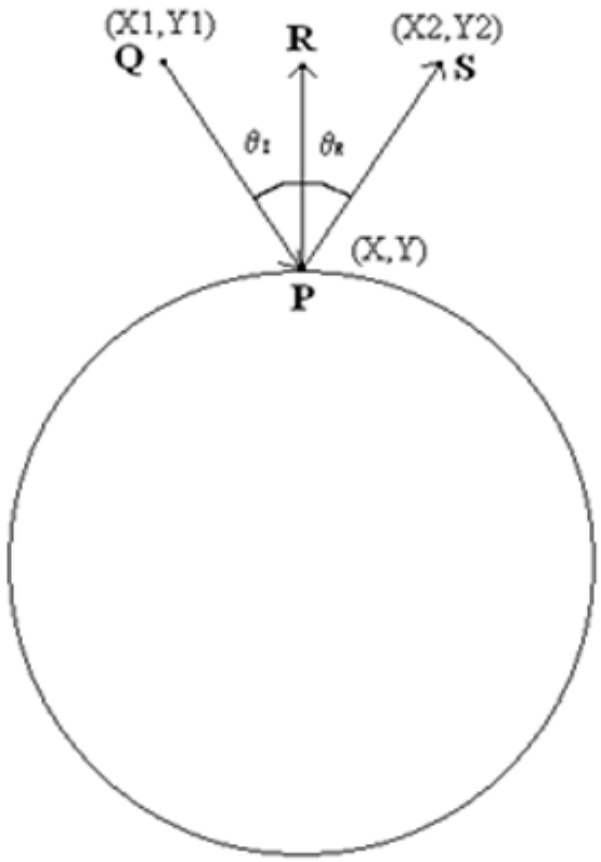

Figure 12: The reflection of a photon off a wave shifting fiber. The reflected angle can be determined and then a new direction vector can be assigned to the photon.

The radial vector will then be normalized via the formula

$$\hat{R} = \frac{\vec{R}}{\left|\vec{R}\right|} \tag{17}$$

where r is the vector heading away form the midpoint of the circle. This allows us to calculate the angle of impact of the particle. Since the direction vector is heading toward the circle and the radial vector is heading away form the circle we must flip the sign of the direction vector to get a correct angle of impact. This yields the equation

$$\theta = \cos^{-1}\left(\frac{\vec{r}\cdot(-\vec{v})}{|\vec{r}|*|\vec{v}|}\right) \tag{18}$$

where r is the vector heading away form the midpoint of the circle, v is the direction vector of the incident particle, and θ is the angle of impact of the incident particle.

The reflection direction vector can be calculated next. This can be done by equating two angle formulae. If we let $(X_1,Y_1) = Q$, $(X,Y) = P$, $(X_2,Y_2) = S$, and the head of the radial vector equal R. We get the equation

$$\frac{PQ \bullet PR}{\|PQ\|\|PR\|} = \frac{PS \bullet PR}{\|PQ\|\|PR\|} \tag{19}$$

where PQ,PR, and PS are vectors. We require that ||PR|| = 1 and ||PQ|| = ||PS|| which we can do without loss of generality. Solving this equation in terms of the coordinates yields the set of equations

$$\begin{aligned} X_2^2 + Y_2^2 &= X_1^2 + Y_1^2 \\ X_2X + Y_2Y &= X_1X + Y_1Y \end{aligned} \tag{20}$$

Solving these equations for X_2 and Y_2 yields the direction vectors after reflection off the fiber. The scintillator material has an index of refraction of 1.58, while the wave-shifting fiber has an index of refraction of 1.6. In addition, the wavelength-shifting fiber is clad with a material having an index of refraction of 1.49. This leads to a dual Snell's Law calculation.

$$\sin(\theta_c) = \frac{n_2}{n_1} \tag{21}$$

$$\sin(\theta_2) = \frac{n_1}{n_2}\sin(\theta_1) \tag{22}$$

$$\sin(\theta_3) = \frac{n_2}{n_3}\sin(\theta_2) \tag{23}$$

in which n_1 is the index of refraction for the scintillator, n_2 is the index of refraction for the cladding, n_3 is the index of refraction for the wave-shifting fiber, θ_c is the critical angle. Beyond the critical angle, photons incident on the cladding of the wave-shifting fiber will experience total reflection. Since the photon is going from a lower index to a higher index in penetrating the fibers from the cladding, there is no critical angle.

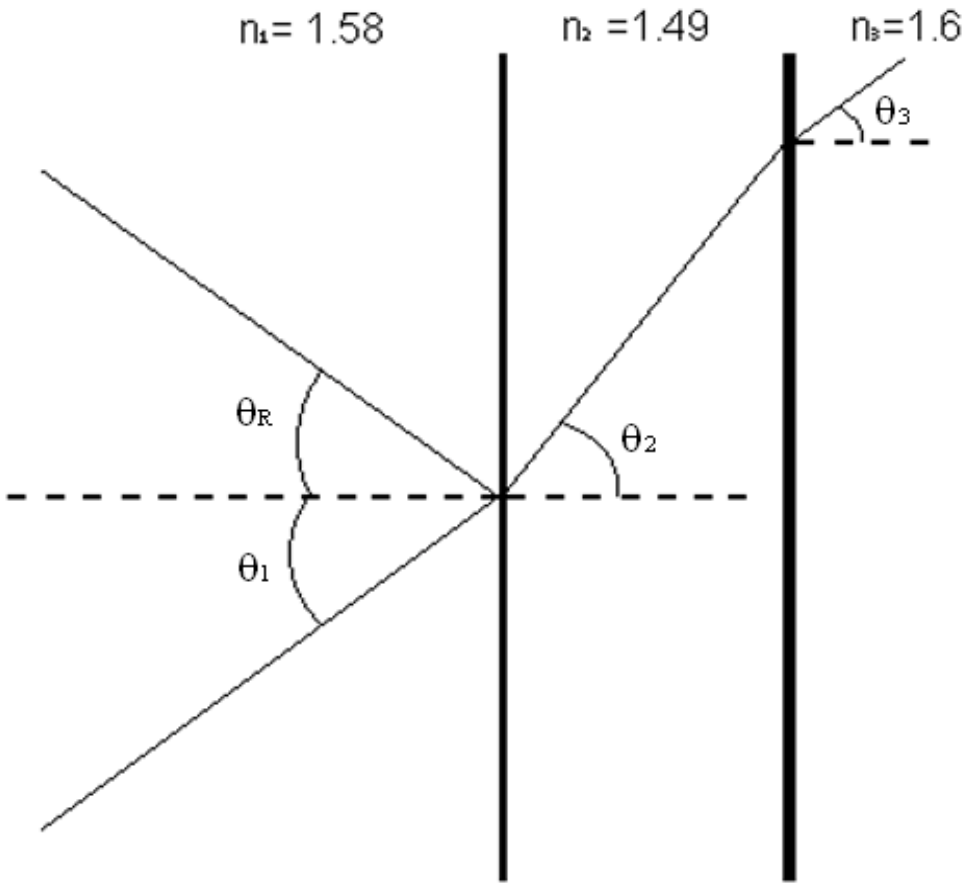

Figure 13: A cross section of reflection and refraction from a Snell's Law surface. If the incident angle is greater then the critical angle, the photon will be reflected. If the incident angle is less then the critical angle the photon will be transmitted.

For a particle to have a chance of being absorbed in the wave-shifting fiber, the condition

$$\theta_c \geq \theta_1 \qquad (24)$$

must be met. If this condition is satisfied, the photon successfully passes into the wave-shifting fiber; if it is not satisfied, the particle undergoes total reflection from the wave shifting fiber. We will approximate total internal reflection by assuming the incident particle is reflected so that $\theta_{1i} = \theta_{1r}$, in which θ_{1i} is the incident angle and θ_{1r} is the reflected angle.

3.6 Photon Behavior Inside the Wave-Shifting Fiber

Once a photon successfully enters the wave-shifting fiber, there is a certain probability that it will be absorbed. The polyvinyltoluene releases photons of many different wavelengths in the blue area of the spectrum; as a result, absorption cannot be easily calculated. The absorption and emission are based on the efficiency of curves of Bicron's BCF-92 as shown in Fig. 14 [14].

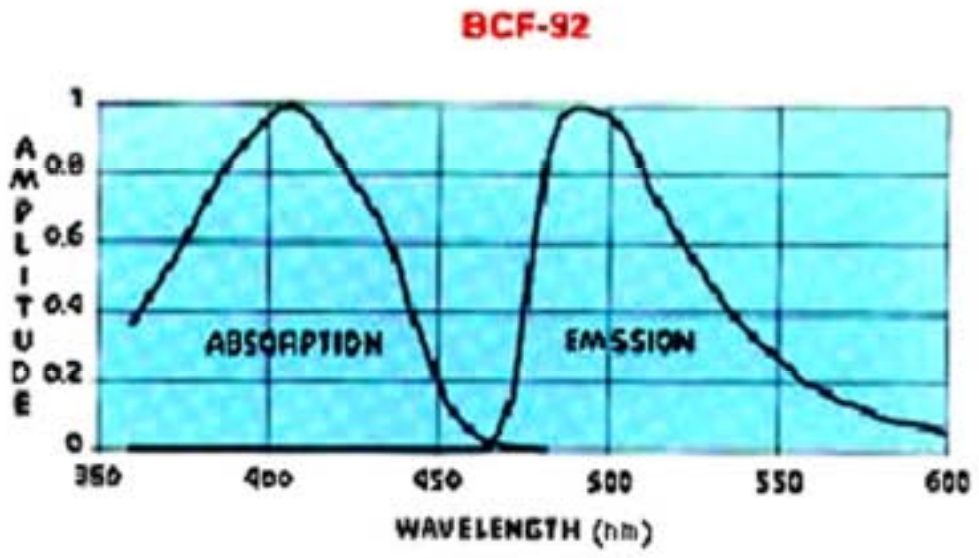

Figure 14: The relative amplitude plotted against the wavelength in nanometers of the absorption and the emission spectrum of the wave-shifting fibers.

The absorption curve in this plot can be compared to the emission curve of the scintillator in Fig. 6. These curves show reasonably good overlap, from wavelengths of 350 nanometers to wavelengths of 450 nanometers, as is required for the photons generated inside the scintillator to be absorbed effectively into the fiber. Using Kaleidagraph we can fit a curve to the absorption and emission spectra of the fiber. The absorption fit is given in Fig 15 and the emission fit is given in Fig 16.

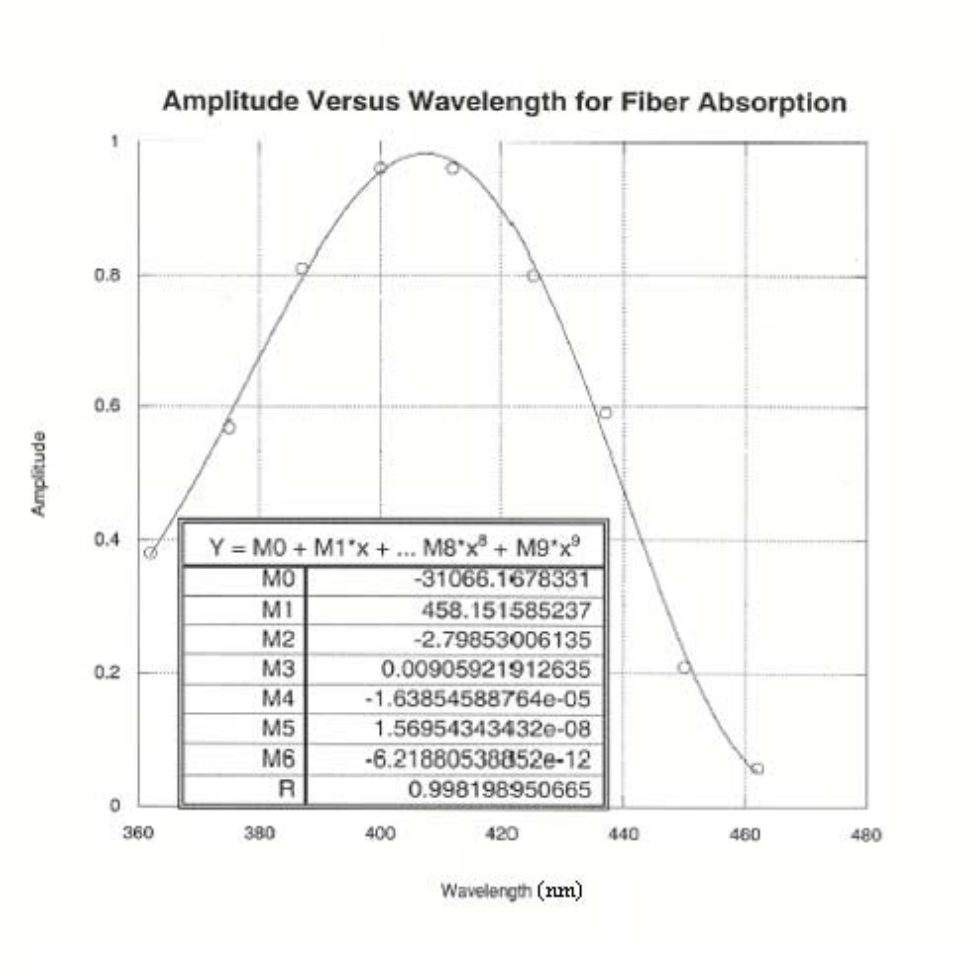

Figure 15: A curve fitted to the absorption spectrum of the wave-shifting fiber. Since the curve is only well behaved in a certain area, it will be bounded to assure it accurately represents the original curve.

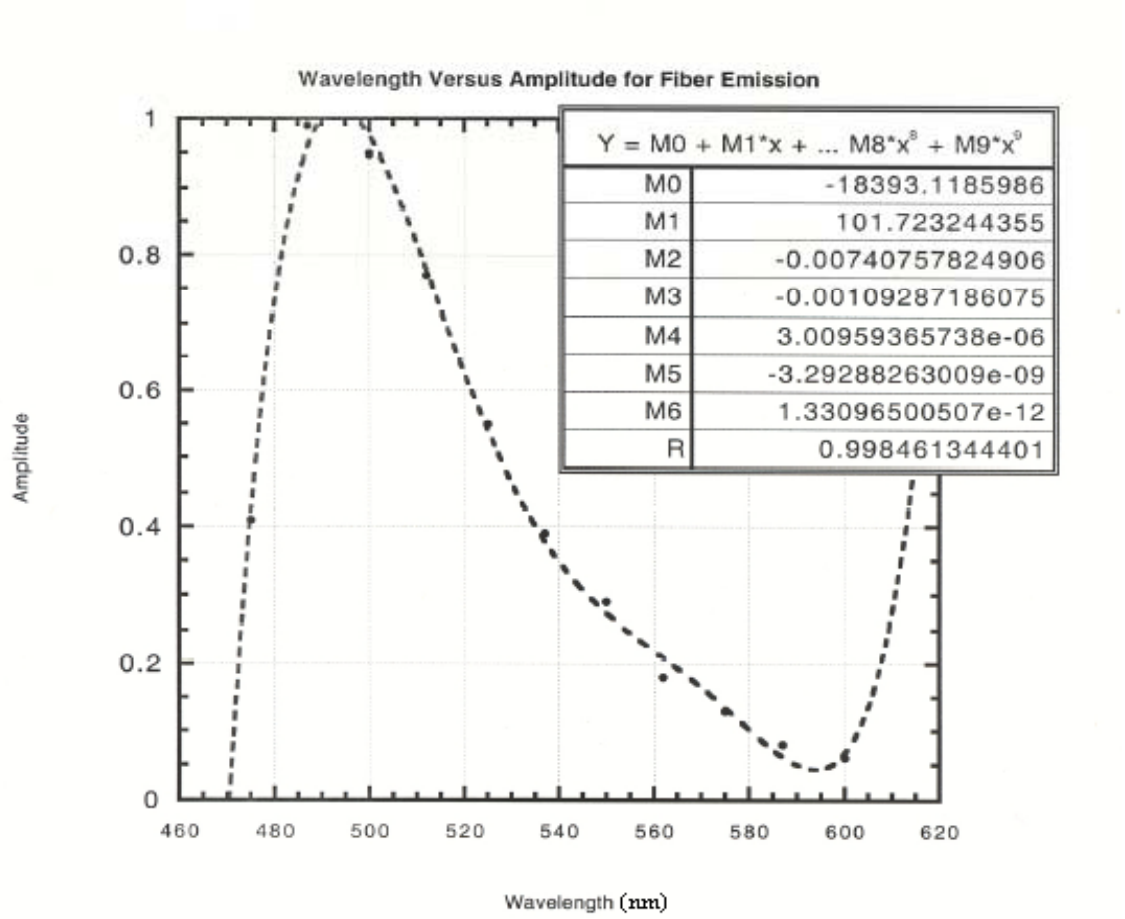

Figure 16: A curve fitted to the emission spectrum of the wave-shifting fiber BCF-92. All curve fits make use of polynomial curves because they are the most versatile type of curve fit that accurately models the original fiber spectra.

In addition to the absorption spectrum for the wave-shifting fiber, it is important to know the percentage of absorption of light per unit length. Thus it becomes necessary to allow the user to input fibers of different diameter. I measured this in the lab by shining the 408 nanometer violet spectral line of mercury through different lengths of wave shifting fiber. Data points were then collected showing the percentage of light reaching the

detector after. This was plotted versus the length of the fiber in millimeters and a curve was fit to the plot shown in Fig 17.

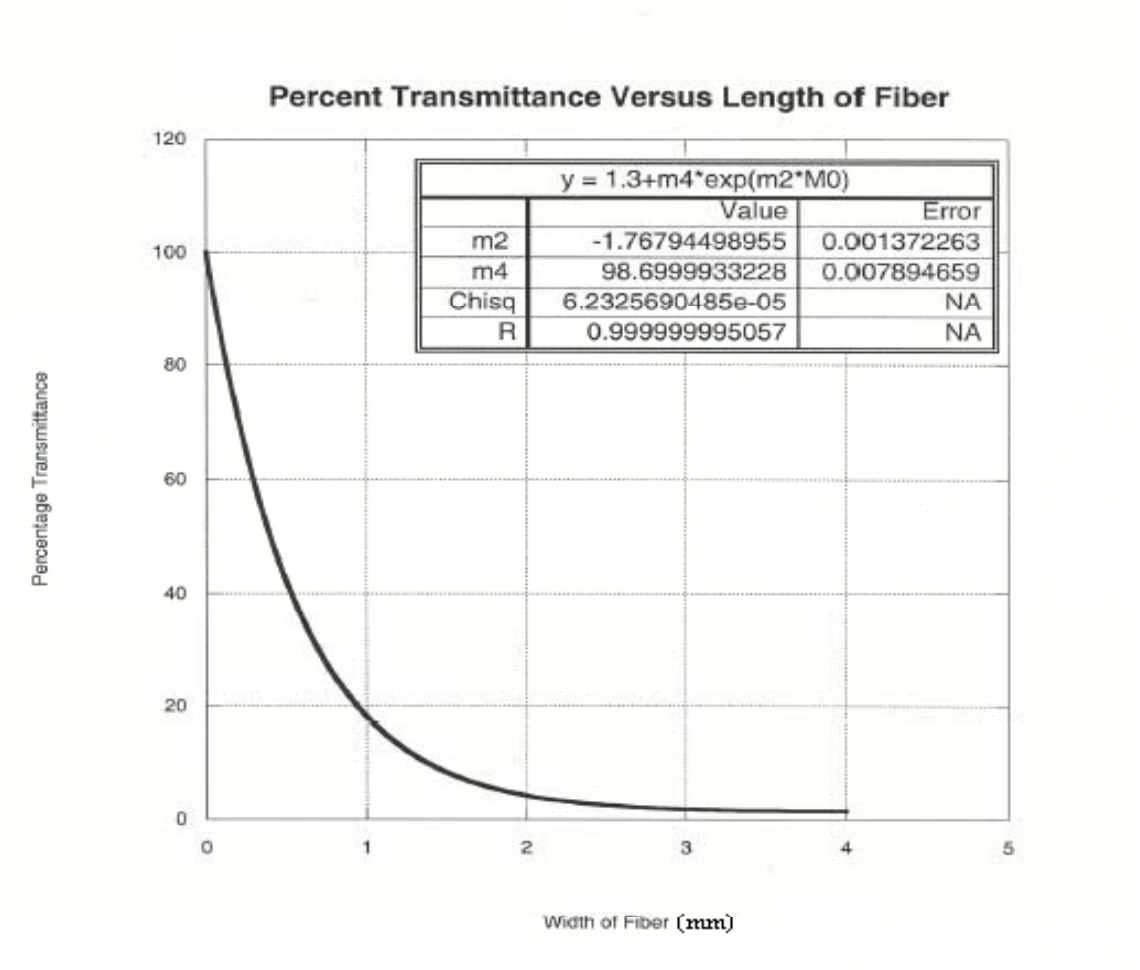

Figure 17: A plot of the percentage of light reaching the detector versus the thickness of the fiber passed through for 408 nanometer light.

This curve in conjunction with the absorption spectrum of the fiber can determine the possibility for a BC-404 photon of any wavelength to be absorbed after traveling through

an arbitrary thickness of fiber. Once a photon has been absorbed, a wave-shifted photon is emitted at the absorption point. This green photon will be captured within the wave-shifting fiber core as long as it is within a certain solid angle. The manufacturer gives this solid angle as 3.44% of the total 4π solid angle [14]. This number can also be calculated since the indexes of refraction are known. The formula:

$$\sin(\theta_c) = \frac{n_2}{n_1} \tag{25}$$

gives the critical angle θ_c as 68.63 degrees, which is the same as in the manufacturer's specifications. The rest of the calculations are:

$$\Omega = 2 * \pi * 1(1 - \frac{n_2}{n_1}) \tag{26}$$

$$f = \frac{\Omega}{4\pi} \tag{27}$$

in which Ω is the maximum solid angle for total internal reflection, and f is the fractional solid angle, which equals 3.4375% or 3.44% after rounding. If a green photon escapes from the wave-shifting fiber, it can be ignored: the fiber will not reabsorb it, and chances are low that it would hit a fiber at an angle permitting total internal reflection. One end of the wave-shifting fiber will be coated with black to absorb all photons incident on it. This effectively cuts the percentage of light that reaches the photomultiplier tube in half. The photon path length inside the fiber will be calculated using the previous formulae for reflection off a circle. This value will become relevant later on in the program output. One final calculation is necessary to accurately predict the number of photons reaching the photomultiplier tube. The fiber has self-attenuation where the photon is scattered

while traveling through the fiber. A formula giving the number of photons traveling through the fiber after a certain length is

$$N(x) = N_0 e^{\frac{-x}{\lambda}} \tag{28}$$

where $N(x)$ is the number of photons after traveling a length x, N_0 is the number of photons traveling through the fiber initially, λ is a constant determined to be about 5 meters.

3.7 The Photomultiplier Tube

While the workings of photomultiplier tubes can be quite complicated, the only aspect we are interested in is the efficiency of the tube's photocathode to convert a photon of wavelength λ to a photo-electron. Our current photomultiplier tube, the Hamamatsu R580-17, has 10 dynodes[5] and a photocathode[6] with enhanced efficiency in the green. The efficiency of these tubes is give by Fig. 18 [15]:

[5] Dynodes are the metal plates that cause the cascading effect used to increase the number of electrons reaching the detector.
[6] This is the area on the anode of the photomultiplier tube that admits the photons into the photoelectron cascading network.

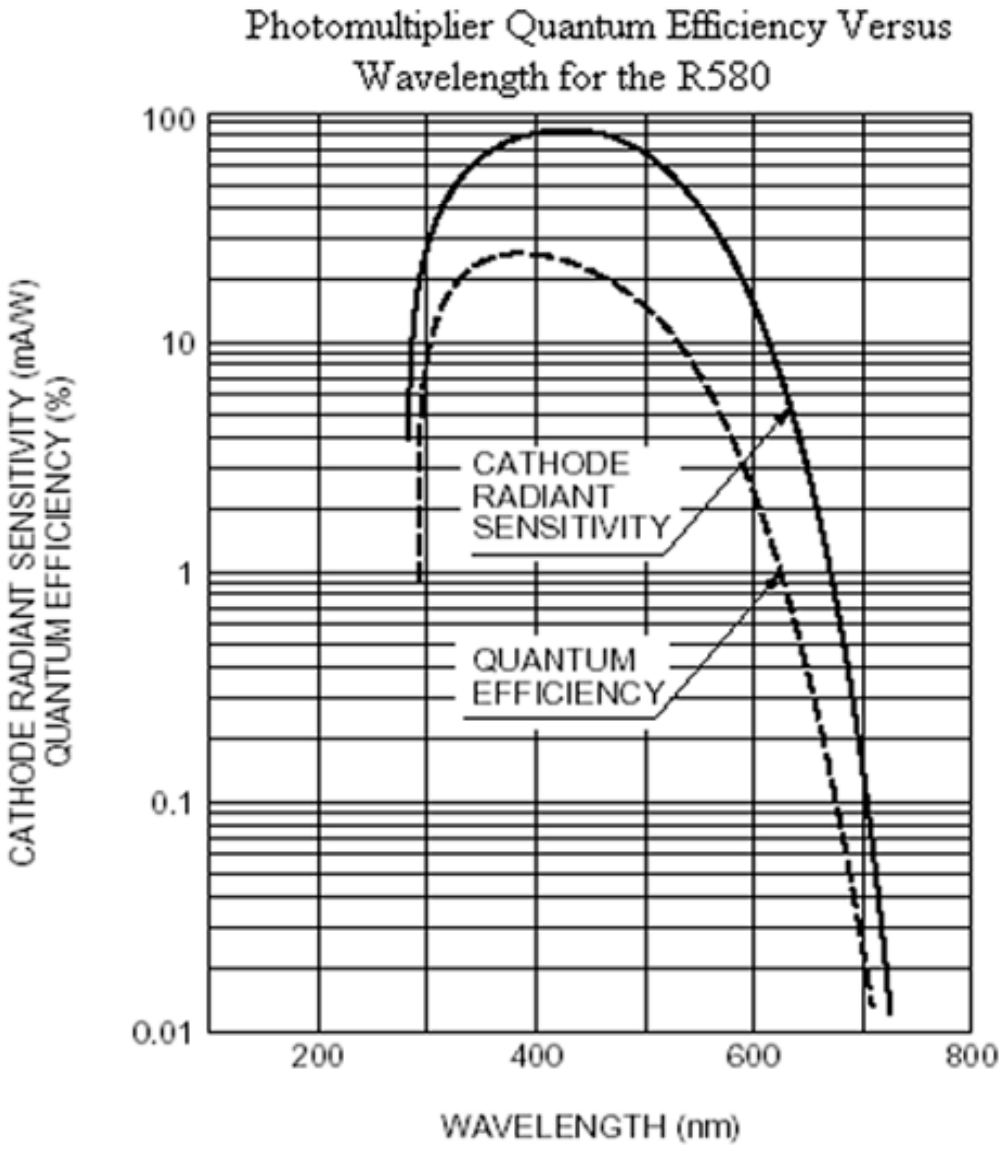

Figure 18: This is a semi log plot of the efficiency of the Hamamatsu R580-17 photomultiplier tube in detecting photons. Quantum efficiency as a percent probability is plotted versus the wavelength of the incident photons in nanometers.

This wavelength dependant efficiency will be converted to a fraction and multiplied by the current number incident photons at this wavelength, giving the final number of photons that are actually detected. Using Kaleidagraph I have fit a curve to the efficiency of the photomultiplier tube. A polynomial can accurately approximate this curve so the user will also be able to enter a polynomial curve in the program. The fit is given in Fig. 18a.

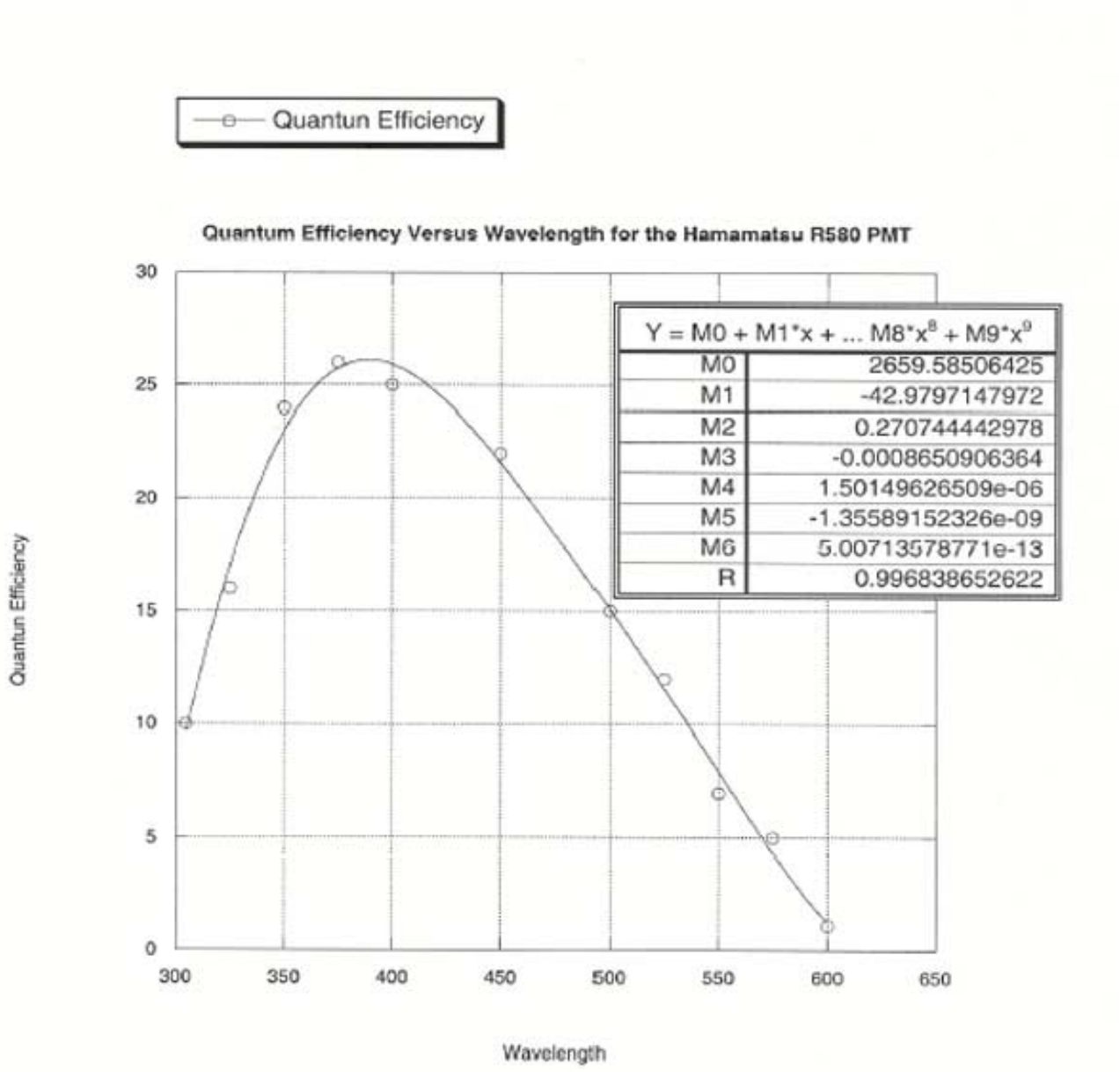

Figure 19: The polynomial curve fit to the photomultiplier tube quantum efficiency versus wavelength as given by Kaleidagraph.

In addition to the default fit used by the program the user will be able to define their own fit for the curve if they are using a different photomultiplier tube. This will be done by

allowing the user to enter the coefficients of a polynomial fit to the photomultiplier tube efficiency curve. Measuring the number of photoelectrons produced on average in the photomultiplier tube is a good way of comparing predicted results to actual results. The photomultiplier tubes will be arranged with bundles of either 3 wave-shifting fibers or 9 wave-shifting fibers in contact with each quadrant of the photocathode. A photomultiplier tube that we plan on using in the future is the R5900U which is a tube with four anodes per cathode. This tube has a slightly different curve making it important for the user to be able to define their own curve for the photomultiplier tube.

4. Program Testing

It is very important when writing computer simulations to test your program periodically to make sure the program is working correctly. This makes it easier to tell where errors are if they occur in the building process and fix them quickly and effectively. It is also important to document the methods taken to debug a program to provide users with evidence for the program working correctly. My first major module is the Muon Impact Module, which is responsible for randomly generating a muon at a random point above the scintillator and determining where that muon will impact the scintillator. Since most of the muons that are incident on the scintillator will have very large negative z velocities it is an adequate approximation to generate muons at a random position overtop of the scintillator plate. This saves a lot of computation time and allows the user to specify any size scintillator without a large amount of muons missing it. This allows the user to specify the number of muons to be fired at the scintillator with reasonable assurance that most of them will end up hitting the scintillator plate. Some

values that were produced when testing this module are given in table 1. The scintillator for this test case had a length of 125 cm, a width of 62.5 cm, and a height of 6.25 cm. The scintillator runs from 0 to the given values when determining if an impact occurred. Many more trials were run then the number displayed, but these trials give a good example of the kind of results the module is generating.

Table 1. Data Points Collected from testing of the muon impact module.

Muon Impact X Position (⅛cm)	**Muon Impact Y Position (⅛cm)**	**Muon Impact Z Position (⅛cm)**	**Did Program See a Muon Impact?**
349.5	252	50	Yes
635	201	50	Yes
817.5	-7	50	No
40	123	50	Yes
282	201.5	50	Yes
153	38.5	50	Yes
483	376.5	50	Yes
928	141	50	Yes
-18.5	224.5	50	No
638	-39.5	50	No
784	449.5	50	Yes
293	546	50	No
515.5	680.5	50	No
713	-83.5	50	No
588.5	253.5	50	Yes
448	131	50	Yes
645.5	317	50	Yes
868	451.5	50	Yes
50.5	198	50	Yes
292.5	272	50	Yes

In all cases the program was able to correctly determine if a muon impact was detected. In addition, the program stops at exactly at the spot where it can be determined whether the muon has impacted or missed the scintillator. As the data shows if a muon ends up outside the bounds for the scintillator at the top of the scintillator it is considered a miss. Since the muon is only generated overtop of the scintillator the case where the muon impacts the scintillator below z = 50, or the top of the scintillator, cannot occur. This method has been implemented to increase program run speed. The Muon Path Module is the module that controls the determination of the length of the muon path in the scintillator. The program must subtract the length of fiber traversed from the total length traversed. The program must perform error checking to assure that this length is as accurate as possible. Since the compiler is only accurate to 6 digits in floating point numbers, small error is introduced into the values during the long calculation. As a result of this computational error, it is important to check the values generated by the program in certain situations to make sure that they are close to the expected values. The scintillator for this test case will have a length of 125 cm, a width of 62.5 cm, and a height of 6.25 cm as in the previous case.

Table 2. Data Points Collected from testing of the muon path module.

Muon Initial Position (x,y,z) (⅛cm)	**Muon Direction (x,y,z) (⅛cm)**	**Wave Shifting Fiber Position (y,z) (⅛cm)**	**Expected Path Length (⅛cm)**	**Outputted Path Length (⅛cm)**
(500,200,50)	(0,0,-1)	(200,48)	48	48
(500,199.5,50)	(0,0,-1)	(200,48)	48.26795	48.2679
(500,202,50)	(0,-1,-1)	(200,48)	68.71067	68.7107
(500,100,50)	(0,0,-1)	(200,48) (100,48)	48	48
(500,99.5,50)	(0,0,-1)	(200,48) (100,48)	48.26795	48.2679
(500,200,50)	(0,0,-1)	(200,0)	49	49
(500,30,50)	(0,-1,-1)	None	42.42641	42.4264
(30,200,50)	(-1,0,-1)	None	42.42641	42.4264
(500,480,50)	(0,1,-1)	(500,30)	27.28427	27.2843
(500,200,50)	(0,0,-1)	(200,50)	49	49

All the generated values are the same as the expected values to six significant digits. Several cases were checked here, such as a simple path through the center of a fiber as shown in the first row in table 2. The expected path length is generated by taking the total path length until the muon exits the scintillator and then subtracting the distance through the fiber it has traveled from that total. The program is also tested with the muon passing through a chord of a fiber in row 2 of table 2. This tests the error checking functions in the program since the photon's step length will bring it to the inside of the fiber instead of right to the surface. The program was then tested with a diagonal path through a fiber in row 3 in table 2. Next measurements were done with multiple fibers to check the program's ability to handle multiple fibers in rows 4 and 5 in table 2. The rest of the measurements check the x and y boundary checking functions and the program's handling of the fiber-scintillator boundary overlap. All of these processes are working as expected. The next major module is the Photon Reflection Module. This module tracks the photon as it bounces around inside the scintillator. This module is designed to run until the photon hits a wall of the scintillator or contacts the wave shifting fiber. As a result the data produced by this module is less complex then the data produced by the previous module. The test scintillator has a length of 125 cm, a width of 62.5 cm, and a height of 6.25 cm.

Table 3: Data points sampled from the testing of the photon reflection module.

Photon Initial Position (x,y,z) (⅛cm)	**Photon Direction (x,y,z) (⅛cm)**	**Wave Shifting Fiber Position (y,z) (⅛cm)**	**Expected Path Length (⅛cm)**	**Outputted Path Length (⅛cm)**
(500,200,50)	(0,0,-.1)	None	50	50
(500,40 ,30)	(0,-.1,-.05)	None	44.72136	44.7214
(100,200,50)	(-.4,-.1,-.1)	None	106.0660	106.066
(100,200,50)	(0,0,-1)	(200,0)	49	49
(100,200,50)	(0,-.1,-.1)	(200,0) (175,25)	34.35534	34.3553
(100,200,25)	(0,.1,0)	(300,25)	99	99
(265,355,31)	(-.1,.1,-.1)	(369,17)	23.0241	23.024

In the first row of table 3 we check to see if the program stops when the photon hits a wall and gives an accurate path length. Then in row 2 we test a photon with a direction component in more then one direction. In row 4 we test to see if the program stops when the photon hits a wave-shifting fiber and gives an accurate path length. It is becoming evident in this module that rounding error is occurring. Some of the values are being rounded to three decimal places instead of four. This is because the module manipulates floating point numbers with a maximum of six significant digits. The reason for this is larger floating point numbers are much harder for the compiler to manipulate. The final major module is the Fiber Path Module. This module takes a photon that has been emitted inside the wave shifting fiber and allows is to bounce around until it reaches the photomultiplier tube. The photon is "killed" if it escapes the fiber or if it heads in the opposite direction of the photomultiplier tube where the black paint on the wrong end of the fiber absorbs it. The test fiber has a diameter of $1/4^{th}$ cm and an index of refraction of 1.60 with cladding having an index of 1.49.

Table 4: Data points collected from the testing of the fiber path module.

Photon Initial Position (x,y,z) (⅛cm)	**Photon Direction (x,y,z) (⅛cm)**	**Wave Shifting Fiber Position (y,z) (⅛cm)**	**Expected Path Length (⅛cm)**	**Outputted Path Length (⅛cm)**
(100,201,20)	(1,.5,-.5)	(200,20)	Dead Photon	Dead Photon
(100,201,20)	(-1,-.5,-.5)	(200,20)	Dead Photon	Dead Photon
(100,201,20)	(-1,-.2,-.5)	(200,20)	113.578	113.578
(100,200,20)	(0,0,1)	(200,20)	Dead Photon	Dead Photon
(100,201.05,20)	(-1,-1,-1)	(200,20)	102.74	102.74
(100,201,20)	(-3,-1,.3)	(200,20)	105.883	105.883
(100,199,20)	(-3,.3,-.1)	(200,20)	100.554	100.554

The fiber path module is performing properly under all cases. In row 1 of table 4 the photon is thrown with an angle smaller then the critical angle of the fiber. As a result we would expect the photon to escape and the program return a dead photon. In row 3 of table 4 the photon successfully internally reflects off the walls of the fiber. We expect the program to return a path length for the total path the photon travels from its initial x value to x equals zero. If a photon dies the module is generating that fact, whereas if it doesn't the module is providing an accurate path length to determine how far the photon travels to get to the end of the fiber. These four modules are the bulk of the program and their output is both accurate and consistent with expectations.

5. Sources of Error

One source of error involves the nature of the scintillation material itself. It is possible for a photon to be emitted then reabsorbed again inside a scintillator. This reabsorption is rare and will only have a small effect on the accuracy of my program. Reabsorption can occur due to impurities in the scintillation material or due to the overlap of the absorption and emission spectra [6]. Though seemingly small, reabsorption can be a major concern over a large path length inside the scintillator. Reabsorption due to impurities is simply out of the scope of this program since it is dependent on a large number of factors and can be greatly reduced by purification techniques. The compiler itself is a source of error in this computation. The compiler is only accurate to 6 significant figures in floating point digits. Though this doesn't seem like much when the program is doing thousands of computations it can add up. This error is unavoidable, however it can be kept in check by documenting the output of each module and making

sure the answers it is generating are not too far off the answers that are expected. Another source of error generated by the compiler is the round off error inherent in floating point variables. These are the variables that store important numbers in the program, such as the coefficients of the polynomial function that is used to determine photon wavelength. A double version of the floating point variable can hold about 15 significant digits [16]. Though this is not a problem in most cases, it may make a difference in higher order polynomial fits, and the user must take it into account when supplying a curve fit for the scintillator or wave shifting fiber. In addition, two notable sources of error occur in the module that determines the path length of the muon. The first of these concerns the accurate detection of the path length. An approximation is necessary here to correctly veto the amount of wave shifting fiber the muon has traversed. This is because the muon is moved using Cartesian coordinates while the wave shifting fiber is represented using cylindrical coordinates. As a result it becomes very tedious to correctly veto the muon's path perfectly. In addition to this, the compiler only has floating point accuracy to six significant figures. A numerical method will be used to give good accuracy for all six significant figures that are known to be accurate. Though this causes some error in the final output of the module, it is not significant enough to cause problems. A more troubling source of error is due to the fact that the muon can't be checked at every point. Attempting to do this many checks would greatly decrease program speed to where it wouldn't be worth running. If a muon starts outside of a scintillating fiber and passes through a chord of it before a check is run, the program will treat it as if it hadn't contacted a fiber at all. This can be a significant source of error, however due to the fact that the muons have much greater z velocity, the muon will

hardly ever contact more then one wave shifting fiber. In addition, since the muon only travels $1/80^{th}$ of a centimeter before a check is run, the amount of path length incorrectly added to its total path will be negligible. In addition to this, it is very rare that a muon will hit a fiber in exactly the right place to cause this problem at all. The muon would have to hit an area of about $2.5 * 10^{-5}$ square centimeters out of a possible area of tens to hundreds of square centimeters. As a result of this, I feel that this approximation is valid and will not cause problems with the accuracy of my program. One of the primary sources of error in this experiment is the accuracy of the curve fits. It is nearly impossible for a perfect fit to a spectrum for the scintillator or the fiber to be generated by any method that this program is capable of handling. As a result a lot of error in the wavelength and hence in the absorption probabilities can be introduced by a curve. This is necessary, however, since though accuracy is lost the user is able to enter a curve fit of their own if they choose to. Another source of error can be caused by the bending of the wave-shifting fiber between the end of the scintillator and the photomultiplier tube. The program does not take this bending into account, however the bending has to be reasonably sharp to cause a noticeable different. This simulation can be adopted to a design with fiber bending by calculating the loss due to bending and modifying the program output based on this calculation.

6. Conclusions

After the program is linked it is important to test its final output and compare it to experiment. Data from an experiment that had been done by the William and Mary MECO team shows 38 photoelectrons being detected per muon on average. The program, using this same configuration, predicts 62 photoelectrons being detected per muon on average. A small difference is to be expected between these results. Inaccuracies such as corrections for bending of the fibers as they leave the scintillator that effect efficiency can occur that my program cannot take into account. These differences are hard to measure or control in the actual experiment since each fiber is bending differently and there is no mechanism to hold the fibers firmly into place beyond the scintillator. In addition, the program doesn't take all of the physics into account. Some aspects such as the muon energy spectrum aren't considered due to time constraints. The program is still effective for comparing scintillator designs, however. This is due to the fact that all scintillator designs are simulated under the same set of premises and it becomes possible to run the program using two different designs and then compare the results. The difference between the designs should be representative of the difference that would be seen in an actual calculation. This is because the physics that the program excludes is common to all scintillator designs and doesn't favor one design over another. Though not exactly in agreement with experiment my program is effective for what it was designed for, mainly, comparing different scintillator designs in simulation. Results pertaining to actual designs can then be surmised from the simulation results.

Appendix A

The Experiment to Determine the Scattering Pattern for the TiO_2 Paint

The purpose of this experiment is to determine how a photon will scatter from the titanium dioxide paint that is coating the scintillator. This is important so a photon's motion can be accurately predicted when it is reflecting around inside the scintillator. The experiment was modified from a Pasco spectrophotometer and uses the parts from that setup. This experiment makes use of a mercury lamp and the 408 nanometer violet line that is characteristic of the mercury spectrum. The 408 nanometer line is very close to the 410 nanometer peak in the emission spectrum of the scintillator so I thought it would be a good approximation of the wavelengths that would actually be impacting the paint. The light from the mercury lamp first passed through a collimating slit to focus the beam onto a small area. The beam then passed through a collimating lens to further focus it and immediately after passed through a diffraction grating. The diffraction grating was turned at an angle for the purpose of spreading out the different spectral lines in the beam, making it possible to isolate the 408 nanometer violet line. After this the violet line was passed through a wavelength selector that blocked all the other lines except the 408 nanometer violet line. Since the violet line was off center it was reflected off a mirror back toward the detector pivot axis. A piece of painted scintillator was then placed in the path of this reflection and a detector mounted on an angle measuring wheel was used to scan the area around the scintillator to detect the light reflected by the paint through a wide range of angles. The diagram of this setup is given in Fig. 20.

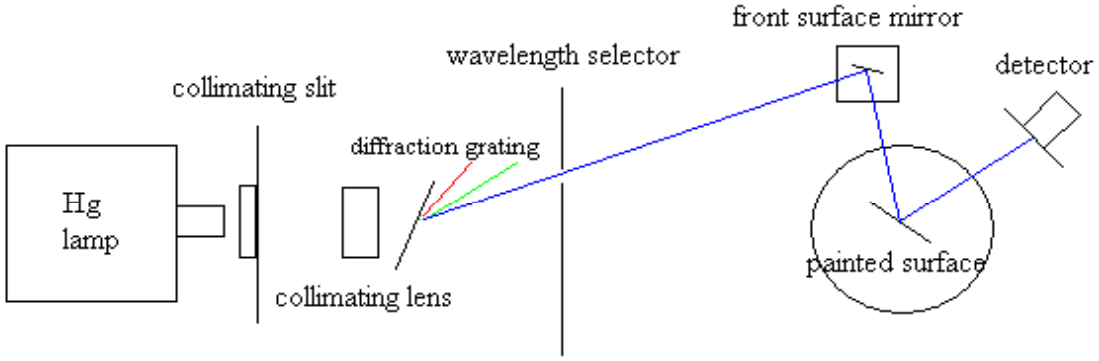

Figure 20: The basic experimental setup for the experiment to measure the reflection pattern of the titanium oxide paint.

We made several different types of measurements to try and determine the form of the scattering from the paint. Initially we had the light traveling through the scintillator material to the paint inside the scintillator and back out again to the detector. This produced a very sharp Gaussian curve. We decided this method was flawed however since the index of refraction of the scintillator is 1.6 and the index of refraction of air is about 1. Any photon reflecting off the paint at an angle greater then the critical angle would be deflected away from the detector and remain inside the scintillator. This would cause a sharp Gaussian peak. To correct this we instead used the paint on the back of the

scintillator to generate the reflection curve. This produced a less pronounced Gaussian-like pattern with a lot of scattering at other angles as shown in fig. 21.

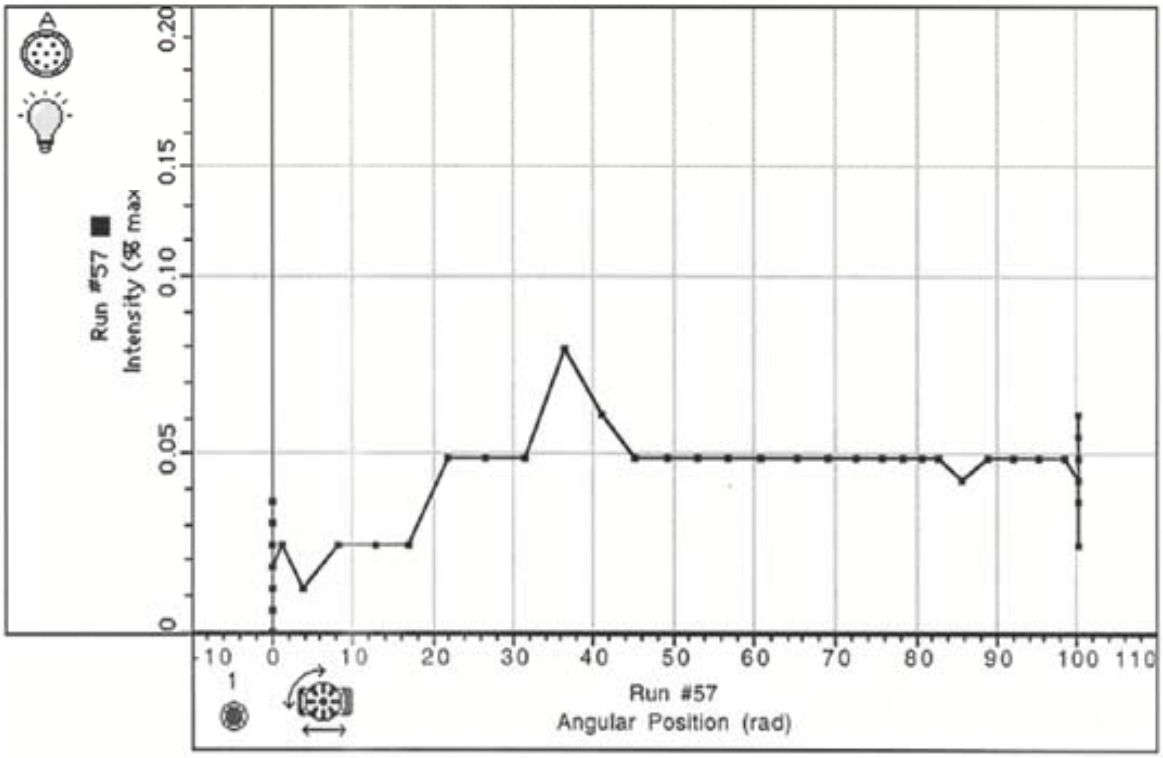

Figure 21: A plot of the intensity versus the angular position for reflections off the paint.

Though the curve shown can be modeled piecewise, it is a lot simpler to assume the paint reflects totally randomly. This simplifies the calculations necessary to predict how the photon reflects immensely. In addition, the peak of the Gaussian isn't significantly large to expect much effect on the results by making this approximation. In addition to measuring the reflection off the paint, it was also necessary to measure the amount of background light being detected. This was done by allowing the light to shine through

the slit but covering up the mirror so none of it could reflect toward the paint. The background was then graphed and a mean value was calculated.

Appendix B

The Experiment to Determine the Length Dependence of the Attenuation of the Fiber

The purpose of this experiment is to determine how the attenuation of light in the wave-shifting fiber depends on the fiber length the light has traveled through. This is important because it allows a more accurate determination of the behavior of the photon inside the wave shifting fiber. The program moves the photon in increments of a certain size, a curve is necessary to determine the probability of the photon being absorbed after each of these increments. This experiment also uses the Pasco spectrophotometer and the parts from that setup. This experiment makes use of the 408 nanometer line of the mercury spectrum as well. It is important to know what wavelength of light was used for this measurement because the attenuation changes based on the wavelength used. This is not a problem since the change is given by a known curve and it will be compensated for prior to the application of the attenuation versus length curve. The setup for this experiment is similar to the setup for the previous experiment. The light is first generated by the mercury lamp and passes through a collimating slit. The light then passes through a collimating lens and the diffraction grating shortly after. The diffraction grating in this case is not rotated at any angle and the light hits it perpendicular to the plane of the grating. The spectrum created by the diffraction grating is then passed through a

focusing lens and focused to very thin lines by the time they reach the detector. The diagram of this setup is given in Fig. 22.

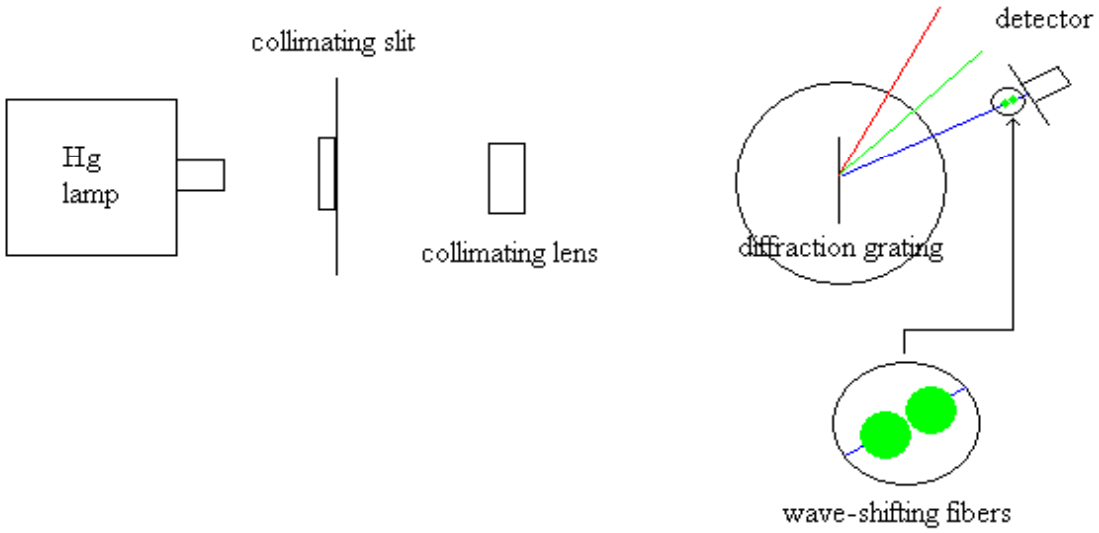

Figure 22: The experimental setup used to determine the attenuation of 408 nanometer light through a certain thickness of wave-shifting fiber.

The detector is set to the thinnest slit and a measurement is taken with no fibers to get a good idea of how much light is being seen by the detector. It is important that the slit be narrow because it can detect green photons given off by the fiber. I would prefer the detector only see the blue light that passes through the fibers which is focused on the slit so making the slit narrower decreases the number of green photons that make it to the detector. After the amount of light without a fiber is recorded a single 2 millimeter wave shifting fiber is placed between the beam and the detector. It is important the beam be narrow so it passes through as much of the fiber as possible. The beam will be focused

on the middle of the fiber so optimally it will all pass through 2 millimeters of fiber before reaching the detector. After the fiber has been positioned correctly measurements will be taken on the amount of light that is passing through the fiber. This result will be recorded and then another fiber will be placed in contact with the first fiber so the beam must pass through both of them as near to the center of the fibers as possible. This result will be recorded and these three data points will be used to calculate the attenuation curve as shown in Fig. 23.

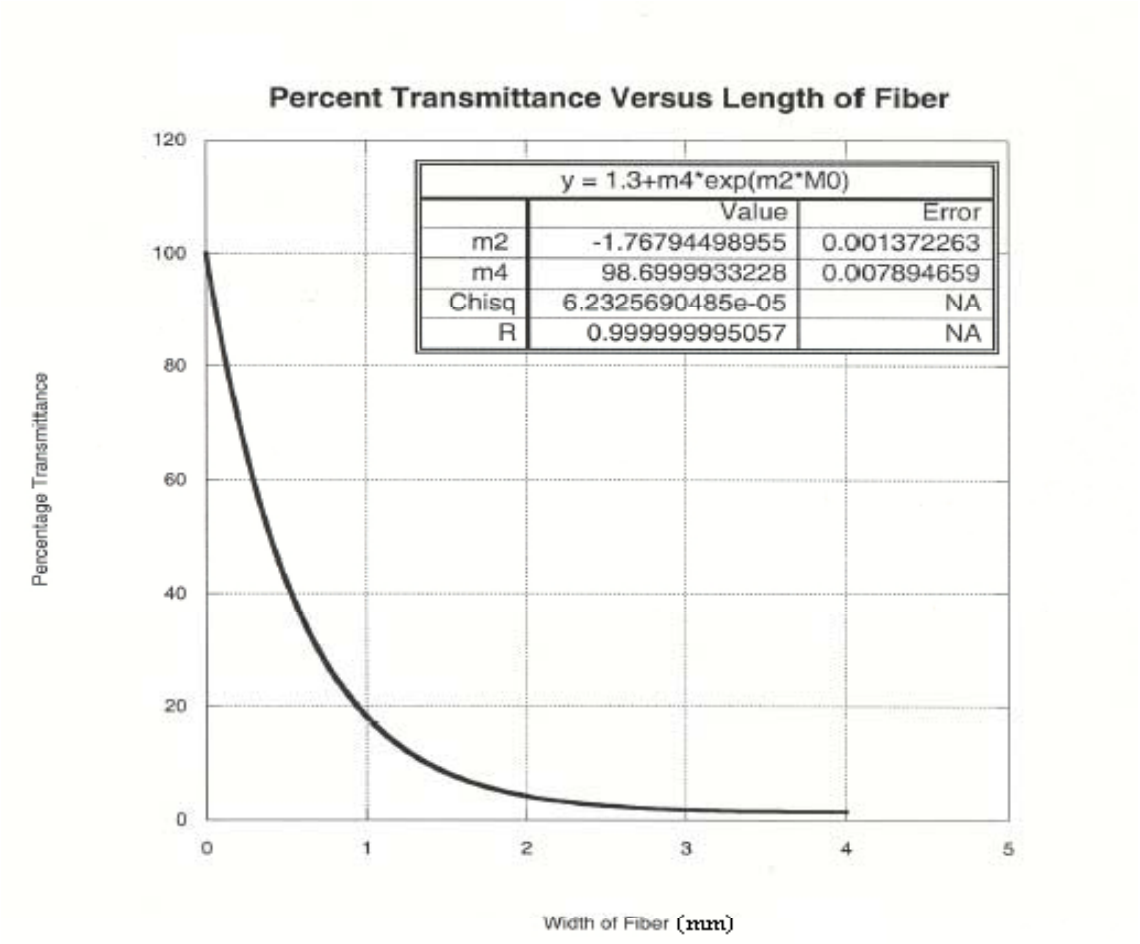

Figure 23: Percentage of light versus the width of the wave shifting fiber.

The attenuation starts large and then drops off as the amount of light that reaches the detector drops off. This fit can easily be used in addition to the absorption spectrum for a particular wavelength to calculate the probability that a photon is absorbed when passing through a small length of the fiber.

Bibliography

[1] NOAA, Cosmic Rays.

http://www.ngdc.noaa.gov/stp/SOLAR/COSMIC_RAYS/cosmic.html.

[2] W. R. Leo, *Techniques for Nuclear and Particle Physics*. (Springer-Verlag, Switzerland, 1994).

[3] J. R. Kane *MECO R&D Proposal to Design and Optimize an Active Cosmic Ray Scintillator Shield*.

[4] The MINOS Collaboration, *The MINOS Detectors Technical Design Report Version 1.0*. NuM I-L-337, October 1998.

[5] D. A. Simon, GUIDEIT V1.1 Users Manuel, 1993, unpublished.

[6] J.B. Birks, *The Theory and Practice of Scintillation Counting*. (The Macmillan Company, NY, 1964).

[7] D. E. Knuth, *The Art of Computer Programming*. (Addison-Wesley, MA, 1997).

[7a] Synergy Software. http://www.synergy.com.

[8] E. O. Lawerence, *High-Energy Particle Data*, UCRL – 2426, Vol. 2. (Berkeley, CA, 1966).

[9] E. P. Lavin, *Monographs on Applied Optics*. (American Elsevier Publishing Company, Inc., NY, 1971).

[10] Bicron, Premium Plastic Scintillators. http://www.detectors.saint-gobain.com/Media/Documents/pdsbc400416.pdf.

[11] J. Lekner, *Theory of Reflection*. (Martinus Nijhoff Publishers, New Zealand, 1987).

[12] J.C. Stover, *Optical Scattering*. (McGraw Hill, Inc., New York, 1990).

[13] Bicron, Scintillating Optical Fibers. http://www.detectors.saint-gobain.com/Media/Documents/pdsbc400416.pdf.

[14] Bicron, Scintillating Optical Fibers. http://www.detectors.saint-gobain.com/Media/Documents/fibers_brochure.pdf.

[15] Hamamatsu, Photomultiplier Tubes. http://usa.hamamatsu.com/hcpdf/parts R/R580.pdf.

[16] S. R. Davis, *C++ for Dummies*. (Hungry Minds, Inc., New York, 2000).

Appendix C
The Coding for the Scintillator Simulation Program

```
//This is the header file that defines the prototype for the userinput module.


#ifndef scinthea.h
#define scinthea.h


void UserInput(int&, int&, int&, int, float, float&, double&, double&, double&,
               double&, double&, double&, double&, double&, double&, double&,
               double&, float&, float&, float&, float&, float&, double&,
               double&, double&, double&, double&, double&, double&, double&,
               double&, double&, double&, double&, double&, double&, double&,
               double&, double&, double&, double&, double&, double&, double&,
               float&, double, double, double, double, double, double, double,
               double, double, double, double, long&);

void MuonImpact(int, int, int, double, float, int&, int&);

void MuonPath(int, int, int, double, float, double, float, float, float&,
              long&, float, int);

void PhotonCreation(double, double, float, int&, float, float, double,
                    double, int&, double, double, double, double, double,
                    double, double, double, double, double, double,
                    int, float, float, int);

int nPhotonReflection(int, int, int, float, float, double, double, double&,
                      int&, int&, int);

int nWallReflect(double, int, int);

int AngleOfReflection(double, double, float, int);

int nInsideFiber(double, double, float, int, float, double&, double, double,
                 double, double, double, double, double, double, double, double,
                 double, int);

void FiberEmit(double, double, double, double, double, double, double, double,
               double, double, double, double, float, float, int&);

void FiberPath(double, double, float, float, double&, int&, int, float);

int nPhotoMultiplier(int, double, double, double, double, double, double,
                     double, double, double, double, double);

#endif
```

```
//This module works perfectly.


//This is the module that gets the user input.
//It outputs all the user inputted values.

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>



//Allow the user to input a function defining a wavelength emission curve after
//it is determined what the best function to represent that curve is.


void UserInput(int& nScintLength, int& nScintWidth, int& nScintHeight,
               int nNumberOfFibers, float fFiberMidpointPosition[10][2],
               float& fRadiusOfFiber, double& dXZero, double& dXOne,
               double& dXTwo, double& dXThree, double& dXFour, double& dXFive,
               double& dXSix, double& dXSeven, double& dXEight, double& dXNine,
               double& dXTen, float& fBoundaryOne, float& fBoundaryTwo,
               float& fEmitBoundaryOne, float& fEmitBoundaryTwo,
               float& fEfficiencyOfScint, double& dXAbsorbZero,
               double& dXAbsorbOne, double& dXAbsorbTwo, double& dXAbsorbThree,
               double& dXAbsorbFour, double& dXAbsorbFive, double& dXAbsorbSix,
               double& dXAbsorbSeven,
               double& dXAbsorbEight, double& dXAbsorbNine, double& dXAbsorbTen,
               double& dXEmitZero, double& dXEmitOne,
               double& dXEmitTwo, double& dXEmitThree, double& dXEmitFour,
               double& dXEmitFive, double& dXEmitSix, double& dXEmitSeven,
               double& dXEmitEight, double& dXEmitNine, double& dXEmitTen,
               float& fLengthOfOutsideFiber,double dXPMTZero, double dXPMTOne,
               double dXPMTTwo, double dXPMTThree, double dXPMTFour,
               double dXPMTFive, double dXPMTSix, double dXPMTSeven,
               double dXPMTEight, double dXPMTNine, double dXPMTTen,
               long& nUserMuonCount)
{
      int nSpecifyFibers = 0;
      int nCounter = 0;
      int nAccept = 0;
      int nGroupOne = 0;
      int nGroupTwo = 0;
      int nGroupThree = 0;
      int nGroupFour = 0;
      do
      {
      do
      {
      do
      {
            cout<<"\n\n\n"
                  <<"             1: Enter Scintillator Information.\n"
```

```
              <<"              2: Enter Fiber Information.\n"
              <<"              3: Enter Photomultiplier Tube
                                 Information.\n"
              <<"              4: Exit User Input.\n"
              <<"\n\n\n\n\n"<<"Scintillator Dimensions must be entered
                 first."<<"\n\n\n\n"
              <<"Enter your choice: ";
       cin >> nGroupOne;
       clrscr();
}
while((nGroupOne != 1) & (nGroupOne != 2) & (nGroupOne != 3) &
       (nGroupOne != 4));
}
while((nGroupOne != 1) & ((nScintLength == 0) | (nScintWidth == 0) |
       (nScintHeight == 0)));

//This is all the input dealing with the scintillator
if(nGroupOne == 1)
{
       do
       {
       do
       {
              cout<<"\n\n\n"
                     <<"              1: Enter Scintillator Length.\n"
                     <<"              2: Enter Scintillator Width.\n"
                     <<"              3: Enter Scintillator Height.\n"
                     <<"              4: Enter Scintillator Efficiency.\n"
                     <<"              5: Enter Scintillator Emission
                                        Spectrum.\n"
                     <<"              6: Exit Scintillator Information.\n"
                     <<"\n\n\n\n\n\n\n\n\n"
                     <<"Enter your choice: ";
              cin >> nGroupTwo;
              clrscr();
       }
       while((nGroupTwo != 1) & (nGroupTwo != 2) & (nGroupTwo != 3) &
              (nGroupTwo != 4) & (nGroupTwo != 5) & (nGroupTwo != 6));
       if(nGroupTwo == 1)
       {
              do
              {
                     cout<<"Enter the value for the length of the
                            scintillator in cm(1-500).\n"
                            <<"Default is "<<(nScintLength/8)<<".\n";
                     cin >> nScintLength;
                     nScintLength = (nScintLength * 8);
                     clrscr();
              }
              while((nScintLength <1) | (nScintLength > 4000));
       }
       if(nGroupTwo == 2)
       {
              do
              {
                     cout<<"Enter the value for the width of the scintillator
                            in cm(1-500).\n"
```

```
                        <<"Default is "<<(nScintWidth/8)<<".\n";
                  cin >> nScintWidth;
                  nScintWidth = (nScintWidth * 8);
                  clrscr();
            }
            while((nScintWidth <1) | (nScintWidth > 4000));
      }
      if(nGroupTwo == 3)
      {
            do
            {
                  cout<<"Enter the value for the height of the
                        scintillator in cm(1-10).\n"
                      <<"Default is "<<(nScintHeight/8)<<".\n";
                  cin >> nScintHeight;
                  nScintHeight = (nScintHeight * 8);
                  clrscr();
            }
            while((nScintHeight <1) | (nScintHeight > 80));
      }
      if(nGroupTwo == 4)
      {
            do
            {
                  cout<<"Input the efficiency of the scintillator in
                        percent of anthracene.\n"
                      <<"Default is "<<(fEfficiencyOfScint*100)
                      <<"% of anthracene. (Enter value as a decimal, i.e.
.68)\n";
                  cin >> fEfficiencyOfScint;
                  clrscr();
            }
            while((fEfficiencyOfScint<0) | (fEfficiencyOfScint > 3));
      }
      if(nGroupTwo == 5)
      {
            do
            {
                  cout<<"Do you want to enter a normalized equation for
                        the emission\n"
                        <<"spectrum of the scintillator? 1 = Yes  0 =
                          No\n"
                        <<"Normalize the spectrum so the maximum value is
                          one.\n"
                        <<"Default value is \n"<<dXZero<<" +\n"
                        <<dXOne<<"x +\n"<<dXTwo<<"x^2 +\n"<<dXThree<<"x^3
                          +\n"
                        <<dXFour<<"x^4 +\n"<<dXFive<<"x^5
                          +\n"<<dXSix<<"x^6 +\n"
                        <<dXSeven<<"x^7 +\n"<<dXEight<<"x^8 +\n"
                        <<dXNine<<"x^9 +\n"<<dXTen<<"x^10.\n\n";
                  cin >>nAccept;
                  clrscr();
            }
            while((nAccept != 1) & (nAccept != 0));
            if(nAccept == 1)
            {
```

```
dXZero = 0;
dXOne = 0;
dXTwo = 0;
dXThree = 0;
dXFour = 0;
dXFive = 0;
dXSix = 0;
dXSeven = 0;
dXEight = 0;
dXNine = 0;
dXTen = 0;
int nPolynomialSelector = 0;
do
{
      cout<<"Enter the order of the polynomial. (0-
            10)\n";
      cin >> nPolynomialSelector;
      clrscr();
}
while((nPolynomialSelector < 0) | (nPolynomialSelector >
      10));
do
{
      switch(nPolynomialSelector)
      {
            case 0:  cout<<"Enter the coefficient for
                         the zeroth polynomial.\n";
                              cin >> dXZero;
                              clrscr();
                              break;
            case 1:  cout<<"Enter the coefficient for
                         the first polynomial.\n";
                              cin >> dXOne;
                              clrscr();
                              break;
            case 2:  cout<<"Enter the coefficient for
                         the second polynomial.\n";
                              cin >> dXTwo;
                              clrscr();
                              break;
            case 3:  cout<<"Enter the coefficient for
                         the third polynomial.\n";
                              cin >> dXThree;
                              clrscr();
                              break;
            case 4:  cout<<"Enter the coefficient for
                         the fourth polynomial.\n";
                              cin >> dXFour;
                              clrscr();
                              break;
            case 5:  cout<<"Enter the coefficient for
                         the fifth polynomial.\n";
                              cin >> dXFive;
                              clrscr();
                              break;
            case 6:  cout<<"Enter the coefficient for
                         the sixth polynomial.\n";
```

```
                                        cin >> dXSix;
                                        clrscr();
                                        break;
                                case 7:  cout<<"Enter the coefficient for
                                        the seventh polynomial.\n";
                                        cin >> dXSeven;
                                        clrscr();
                                        break;
                                case 8:  cout<<"Enter the coefficient for
                                        the eight polynomial.\n";
                                        cin >> dXEight;
                                        clrscr();
                                        break;
                                case 9:  cout<<"Enter the coefficient for
                                        the ninth polynomial.\n";
                                        cin >> dXNine;
                                        clrscr();
                                        break;
                                case 10: cout<<"Enter the coefficient for
                                        the tenth polynomial.\n";
                                        cin >> dXTen;
                                        clrscr();
                                        break;
                        }
                        nPolynomialSelector--;
                }
                while(nPolynomialSelector != -1);
                cout<<"Enter the left boundary for the polynomial.\n
                        Default value is "
                         <<fBoundaryOne<<".\n";
                cin >> fBoundaryOne;
                        clrscr();
                cout<<"Enter the right boundary for the polynomial.\n
                        Default value is "
                         <<fBoundaryTwo<<".\n";
                cin >> fBoundaryTwo;
                clrscr();
        }
    }
    }
    while(nGroupTwo != 6);
}

//This is all the input dealing with the fibers.
if(nGroupOne == 2)
{
    do
    {
    do
    {
        cout<<"\n\n\n"
                <<"             1: Enter Fiber Diameter.\n"
                <<"             2: Enter Fiber Positions.\n"
                <<"             3: Enter Fiber Excess.\n"
                <<"             4: Enter Fiber Absorption Spectrum.\n"
                <<"             5: Enter Fiber Emission Spectrum.\n"
                <<"             6: Exit Fiber Information.\n"
```

```
             <<"\n\n\n\n\n\n\n\n\n"
             <<"Enter your choice: ";
      cin >> nGroupThree;
      clrscr();
}
while((nGroupThree != 1) & (nGroupThree != 2) & (nGroupThree != 3) &
      (nGroupThree != 4) & (nGroupThree != 5) & (nGroupThree != 6));
if(nGroupThree == 1)
{
      do
      {
            cout<<"Enter the diameter of the wave-shifting fibers in
                mm(1-10).\n"
                <<"Default Value is "<<(fRadiusOfFiber/.4)<<".\n";
            cin >> fRadiusOfFiber;
            fRadiusOfFiber = (.4 * fRadiusOfFiber);
            clrscr();
      }
      while((fRadiusOfFiber < 0) | (fRadiusOfFiber > 4));
}
if(nGroupThree == 2)
{
      //This is the loop allowing the user to input the fiber data.
      do
      {
            cout<<"Type 0 to specify the x, y, and z positions for
                each of your fibers\n"
                <<"or type -1 for default.\n\n"<<"Default value is
                evenly spaced fibers "
                <<"with the z position equal to the radius \n"<<"of
                the fiber.\n\n";
            cin >> nSpecifyFibers;
            clrscr();
      }
      while((nSpecifyFibers !=-1) & (nSpecifyFibers !=0));
      if(nSpecifyFibers == 0)
      {
            nCounter = 0;
            do
            {
                  nCounter++;
                  cout<<"Specify Location of fiber
                        "<<nCounter<<".\n";
                  cout<<"\n\n";
                  cout<<"        --------- x=length          -------
                        ---    z=height \n";
                  cout<<"        |       |                   |
                        |  ^          \n";
                  cout<<"        |       |                   -------
                        ---  |  z = 0   \n";
                  cout<<"        |       |                   ^ --->
                        ^             \n";
                  cout<<"        |       |                   |
                        |             \n";
                  cout<<"        |       |   ^             y=0
                        y=width         \n";
                  cout<<"        |       |   |
```

```
                        \n";
                  cout<<"        |       |   |          Cross
                        section at x = 0.     \n";
                  cout<<"        --------- x = 0
                        \n";
                  cout<<"        ^ -->   ^
                        \n";
                  cout<<"        |       |
                        \n";
                  cout<<"       y=0    y=width
                        \n";
                  cout<<"\n";
                  cout<<"           ---
                        \n";
                  cout<<"           | | <---Photomultiplier Tube
                        \n\n";
                  cout<<"Specify y Coordinate in cm (0-
                        "<<(nScintWidth/8)<<").\n";
                  cin >> fFiberMidpointPosition[(nCounter-1)][0];
                  cout<<"Specify z Coordinate in cm (0-
                        "<<(nScintHeight/8)<<").\n";
                  cin >> fFiberMidpointPosition[(nCounter-1)][1];
                  clrscr();
            }
            while(nCounter != nNumberOfFibers);
      }
      else
      {
            do
            {
                  fFiberMidpointPosition[nCounter][0] = ((1. /
                  (nNumberOfFibers + 1.))* (nScintWidth) *
                  (nCounter+1));
                  fFiberMidpointPosition[nCounter][1] =
                  (nScintHeight - fRadiusOfFiber);
                  nCounter++;
            }
            while(nCounter != nNumberOfFibers);
      }
}
if(nGroupThree == 3)
{
      do
      {
            cout<<"Enter the length of the fiber, in centimeters
                  between the scintillator\n"
                <<"and the photomultiplier tube.  Default Value
                  is "<<(fLengthOfOutsideFiber/8)<<".\n";
            cin >> fLengthOfOutsideFiber;
            fLengthOfOutsideFiber = (8*fLengthOfOutsideFiber);
            clrscr();
      }
      while(fLengthOfOutsideFiber<0);
}
if(nGroupThree == 4)
{
      do
```

```
{
      cout<<"Do you want to enter a normalized equation for
             the absorption\n"
          <<"spectrum of the fiber? 1 = Yes  0 = No\n"
          <<"Normalize the spectrum so the maximum value is
            one.\n"
          <<"Default value is \n"<<dXAbsorbZero<<"
            +\n"<<dXAbsorbOne<<"x +\n"
          <<dXAbsorbTwo<<"x^2 +\n"<<dXAbsorbThree<<"x^3 +\n"
          <<dXAbsorbFour<<"x^4 +\n"<<dXAbsorbFive<<"x^5 +\n"
          <<dXAbsorbSix<<"x^6 +\n"<<dXAbsorbSeven<<"x^7 +\n"
          <<dXAbsorbEight<<"x^8 +\n"<<dXAbsorbNine<<"x^9 +\n"
          <<dXAbsorbTen<<"x^10.\n\n";
      cin >>nAccept;
      clrscr();
}
while((nAccept != 1) & (nAccept != 0));
if(nAccept == 1)
{
      dXAbsorbZero = 0;
      dXAbsorbOne = 0;
      dXAbsorbTwo = 0;
      dXAbsorbThree = 0;
      dXAbsorbFour = 0;
      dXAbsorbFive = 0;
      dXAbsorbSix = 0;
      dXAbsorbSeven = 0;
      dXAbsorbEight = 0;
      dXAbsorbNine = 0;
      dXAbsorbTen = 0;
      int nPolynomialSelector = 0;
      do
      {
            cout<<"Enter the order of the polynomial. (0-
                  10)\n";
            cin >> nPolynomialSelector;
            clrscr();
      }
      while((nPolynomialSelector < 0) | (nPolynomialSelector >
             10));
      do
      {
            switch(nPolynomialSelector)
            {
                  case 0:  cout<<"Enter the coefficient for
                               the zeroth polynomial.\n";
                                    cin >> dXAbsorbZero;
                                    clrscr();
                                    break;
                  case 1:  cout<<"Enter the coefficient for
                               the first polynomial.\n";
                                    cin >> dXAbsorbOne;
                                    clrscr();
                                    break;
                  case 2:  cout<<"Enter the coefficient for
                               the second polynomial.\n";
                                    cin >> dXAbsorbTwo;
```

```
                                        clrscr();
                                        break;
                                case 3:  cout<<"Enter the coefficient for
                                        the third polynomial.\n";
                                        cin >> dXAbsorbThree;
                                        clrscr();
                                        break;
                                case 4:  cout<<"Enter the coefficient for
                                        the fourth polynomial.\n";
                                        cin >> dXAbsorbFour;
                                        clrscr();
                                        break;
                                case 5:  cout<<"Enter the coefficient for
                                        the fifth polynomial.\n";
                                        cin >> dXAbsorbFive;
                                        clrscr();
                                        break;
                                case 6:  cout<<"Enter the coefficient for
                                        the sixth polynomial.\n";
                                        cin >> dXAbsorbSix;
                                        clrscr();
                                        break;
                                case 7:  cout<<"Enter the coefficient for
                                        the seventh polynomial.\n";
                                        cin >> dXAbsorbSeven;
                                        clrscr();
                                        break;
                                case 8:  cout<<"Enter the coefficient for
                                        the eight polynomial.\n";
                                        cin >> dXAbsorbEight;
                                        clrscr();
                                        break;
                                case 9:  cout<<"Enter the coefficient for
                                        the ninth polynomial.\n";
                                        cin >> dXAbsorbNine;
                                        clrscr();
                                        break;
                                case 10: cout<<"Enter the coefficient for
                                        the tenth polynomial.\n";
                                        cin >> dXAbsorbTen;
                                        clrscr();
                                        break;
                        }
                        nPolynomialSelector--;
                }
                while(nPolynomialSelector != -1);
        }
}
if(nGroupThree == 5)
{
        do
        {
                cout<<"Do you want to enter a normalized equation for
                        the emission\n"
                    <<"spectrum of the fiber? 1 = Yes  0 = No\n"
                    <<"Normalize the spectrum so the maximum value is
                      one.\n"
```

```
            <<"Default value is \n"<<dXEmitZero<<" +\n"
            <<dXEmitOne<<"x +\n"<<dXEmitTwo<<"x^2
              +\n"<<dXEmitThree<<"x^3 +\n"
            <<dXEmitFour<<"x^4 +\n"<<dXEmitFive<<"x^5 +\n"
            <<dXEmitSix<<"x^6 +\n"<<dXEmitSeven<<"x^7 +\n"
            <<dXEmitEight<<"x^8 +\n"<<dXEmitNine<<"x^9 +\n"
            <<dXEmitTen<<"x^10.\n\n";
      cin>>nAccept;
      clrscr();
}
while((nAccept != 1) & (nAccept != 0));
if(nAccept == 1)
{
      dXEmitZero = 0;
      dXEmitOne = 0;
      dXEmitTwo = 0;
      dXEmitThree = 0;
      dXEmitFour = 0;
      dXEmitFive = 0;
      dXEmitSix = 0;
      dXEmitSeven = 0;
      dXEmitEight = 0;
      dXEmitNine = 0;
      dXEmitTen = 0;
      int nPolynomialSelector = 0;
      do
      {
            cout<<"Enter the order of the polynomial. (0-
                  10)\n";
            cin >> nPolynomialSelector;
            clrscr();
      }
      while((nPolynomialSelector < 0) | (nPolynomialSelector >
            10));
      do
      {
            switch(nPolynomialSelector)
            {
                  case 0:  cout<<"Enter the coefficient for
                               the zeroth polynomial.\n";
                                    cin >> dXEmitZero;
                                    clrscr();
                                    break;
                  case 1:  cout<<"Enter the coefficient for
                               the first polynomial.\n";
                                    cin >> dXEmitOne;
                                    clrscr();
                                    break;
                  case 2:  cout<<"Enter the coefficient for
                               the second polynomial.\n";
                                    cin >> dXEmitTwo;
                                    clrscr();
                                    break;
                  case 3:  cout<<"Enter the coefficient for
                               the third polynomial.\n";
                                    cin >> dXEmitThree;
                                    clrscr();
```

```
                                    break;
                              case 4:  cout<<"Enter the coefficient for
                                   the fourth polynomial.\n";
                                    cin >> dXEmitFour;
                                    clrscr();
                                    break;
                              case 5:  cout<<"Enter the coefficient for
                                   the fifth polynomial.\n";
                                    cin >> dXEmitFive;
                                    clrscr();
                                    break;
                              case 6:  cout<<"Enter the coefficient for
                                   the sixth polynomial.\n";
                                    cin >> dXEmitSix;
                                    clrscr();
                                    break;
                              case 7:  cout<<"Enter the coefficient for
                                   the seventh polynomial.\n";
                                    cin >> dXEmitSeven;
                                    clrscr();
                                    break;
                              case 8:  cout<<"Enter the coefficient for
                                   the eight polynomial.\n";
                                    cin >> dXEmitEight;
                                    clrscr();
                                    break;
                              case 9:  cout<<"Enter the coefficient for
                                   the ninth polynomial.\n";
                                    cin >> dXEmitNine;
                                    clrscr();
                                    break;
                              case 10: cout<<"Enter the coefficient for
                                   the tenth polynomial.\n";
                                    cin >> dXEmitTen;
                                    clrscr();
                                    break;
                         }
                         nPolynomialSelector--;
                    }
                    while(nPolynomialSelector != -1);
                    cout<<"Enter the left boundary for the polynomial.
                         Default value is/n"
                          <<fEmitBoundaryOne<<".\n";
                    cin >> fEmitBoundaryOne;
                    clrscr();
                    cout<<"Enter the right boundary for the polynomial.
                         Default value is/n"
                          <<fEmitBoundaryTwo<<".\n";
                    cin >> fEmitBoundaryTwo;
                    clrscr();
               }
          }
          }
          while(nGroupThree != 6);
}
```

```
if(nGroupOne == 3)
{
      do
      {
      do
      {
            cout<<"\n\n\n"
                  <<"             1: Enter PMT Efficiency Curve.\n"
                  <<"             2: Exit Photomultiplier Tube
                    Information.\n"
                  <<"\n\n\n\n\n\n\n\n\n"
                  <<"Enter your choice: ";
            cin >> nGroupFour;
            clrscr();
      }

      while((nGroupFour != 1) & (nGroupFour != 2));
      if(nGroupFour == 1)
      {
            do
            {
                  cout<<"Do you want to enter a normalized equation for
                        the quantum\n"
                      <<"efficiency of the photomultiplier tube? 1 = Yes
                        0 = No\n"
                      <<"Normalize the spectrum so the maximum value is
                        one.\n"
                      <<"Default value is \n"<<dXPMTZero<<"
                        +\n"<<dXPMTOne<<"x +\n"
                      <<dXPMTTwo<<"x^2 +\n"<<dXPMTThree<<"x^3
                        +\n"<<dXPMTFour<<"x^4 +\n"
                      <<dXPMTFive<<"x^5 +\n"<<dXPMTSix<<"x^6 +\n"
                      <<dXPMTSeven<<"x^7 +\n"<<dXPMTEight<<"x^8 +\n"
                      <<dXPMTNine<<"x^9 +\n"<<dXPMTTen<<"x^10.\n\n";
                  cin >>nAccept;
                  clrscr();
            }
            while((nAccept != 1) & (nAccept != 0));
            if(nAccept == 1)
            {
                  dXPMTZero = 0;
                  dXPMTOne = 0;
                  dXPMTTwo = 0;
                  dXPMTThree = 0;
                  dXPMTFour = 0;
                  dXPMTFive = 0;
                  dXPMTSix = 0;
                  dXPMTSeven = 0;
                  dXPMTEight = 0;
                  dXPMTNine = 0;
                  dXPMTTen = 0;
                  int nPolynomialSelector = 0;
                  do
```

```
{
      cout<<"Enter the order of the polynomial. (0-
            10)\n";
      cin >> nPolynomialSelector;
      clrscr();
}
while((nPolynomialSelector < 0) | (nPolynomialSelector >
      10));
do
{
      switch(nPolynomialSelector)
      {
            case 0:  cout<<"Enter the coefficient for
                         the zeroth polynomial.\n";
                               cin >> dXPMTZero;
                               clrscr();
                               break;
            case 1:  cout<<"Enter the coefficient for
                         the first polynomial.\n";
                               cin >> dXPMTOne;
                               clrscr();
                               break;
            case 2:  cout<<"Enter the coefficient for
                         the second polynomial.\n";
                               cin >> dXPMTTwo;
                               clrscr();
                               break;
            case 3:  cout<<"Enter the coefficient for
                         the third polynomial.\n";
                               cin >> dXPMTThree;
                               clrscr();
                               break;
            case 4:  cout<<"Enter the coefficient for
                         the fourth polynomial.\n";
                               cin >> dXPMTFour;
                               clrscr();
                               break;
            case 5:  cout<<"Enter the coefficient for
                         the fifth polynomial.\n";
                               cin >> dXPMTFive;
                               clrscr();
                               break;
            case 6:  cout<<"Enter the coefficient for
                         the sixth polynomial.\n";
                               cin >> dXPMTSix;
                               clrscr();
                               break;
            case 7:  cout<<"Enter the coefficient for
                         the seventh polynomial.\n";
                               cin >> dXPMTSeven;
                               clrscr();
                               break;
            case 8:  cout<<"Enter the coefficient for
                         the eight polynomial.\n";
                               cin >> dXPMTEight;
                               clrscr();
                               break;
```

```
                                    case 9:  cout<<"Enter the coefficient for
                                                the ninth polynomial.\n";
                                                cin >> dXPMTNine;
                                                clrscr();
                                                break;
                                    case 10: cout<<"Enter the coefficient for
                                                the tenth polynomial.\n";
                                                cin >> dXPMTTen;
                                                clrscr();
                                                break;
                              }
                              nPolynomialSelector--;
                        }
                        while(nPolynomialSelector != -1);
                  }
            }
            }
            while(nGroupFour != 2);
   }
      }
      while(nGroupOne != 4);
      do
      {
            cout<<"Enter the number of total muons that you want incident on the
                "<<"scintillator.\n(1-10000)\n";
            cin >> nUserMuonCount;

            clrscr();
      }
   while((nUserMuonCount < 0)&(nUserMuonCount > 10000));
}


/*void main()
{
int nScintLength = 0;
int nScintWidth = 0;
int nScintHeight = 0;
int nNumberOfFibers = 1;
float fFiberMidpointPositionMatrix[10][2] = {{0,0},{0,0},{0,0},{0,0},{0,0},
                                             {0,0},{0,0},{0,0},{0,0},{0,0}};
float fRadiusOfFibers = 1.;
double dXZero = 0.;
double dXOne = 0.;
double dXTwo = 0.;
double dXThree = 0.;
double dXFour = 0.;
double dXFive = 0.;
double dXSix = 0.;
double dXSeven = 0.;
double dXEight = 0.;
double dXNine = 0.;
double dXTen = 0.;
float fBoundaryOne = 0.;
float fBoundaryTwo = 0.;
float fEmitBoundaryOne = 0.;
float fEmitBoundaryTwo = 0.;
```

```
float fEfficiencyOfScint = 0.;
double dXAbsorbZero = 0.;
double dXAbsorbOne = 0.;
double dXAbsorbTwo = 0.;
double dXAbsorbThree = 0.;
double dXAbsorbFour = 0.;
double dXAbsorbFive = 0.;
double dXAbsorbSix = 0.;
double dXAbsorbSeven = 0.;
double dXAbsorbEight = 0.;
double dXAbsorbNine = 0.;
double dXAbsorbTen = 0.;
double dXEmitZero = 0.;
double dXEmitOne = 0.;
double dXEmitTwo = 0.;
double dXEmitThree = 0.;
double dXEmitFour = 0.;
double dXEmitFive = 0.;
double dXEmitSix = 0.;
double dXEmitSeven = 0.;
double dXEmitEight = 0.;
double dXEmitNine = 0.;
double dXEmitTen = 0.;
float fLengthOfOutsideFiber = 0.;
double dXPMTZero = 0.;
double dXPMTOne = 0.;
double dXPMTTwo = 0.;
double dXPMTThree = 0.;
double dXPMTFour = 0.;
double dXPMTFive = 0.;
double dXPMTSix = 0.;
double dXPMTSeven = 0.;
double dXPMTEight = 0.;
double dXPMTNine = 0.;
double dXPMTTen = 0.;
long int nUserMuonCount = 0;
UserInput(nScintLength, nScintWidth, nScintHeight, nNumberOfFibers,
          fFiberMidpointPositionMatrix, fRadiusOfFibers, dXZero, dXOne, dXTwo,
          dXThree, dXFour, dXFive, dXSix, dXSeven, dXEight, dXNine, dXTen,
          fBoundaryOne, fBoundaryTwo, fEmitBoundaryOne, fEmitBoundaryTwo,
          fEfficiencyOfScint, dXAbsorbZero, dXAbsorbOne, dXAbsorbTwo,
          dXAbsorbThree, dXAbsorbFour, dXAbsorbFive, dXAbsorbSix, dXAbsorbSeven,
          dXAbsorbEight, dXAbsorbNine, dXAbsorbTen, dXEmitZero, dXEmitOne,
          dXEmitTwo, dXEmitThree, dXEmitFour, dXEmitFive, dXEmitSix,
          dXEmitSeven, dXEmitEight, dXEmitNine, dXEmitTen,
          fLengthOfOutsideFiber, dXPMTZero, dXPMTOne, dXPMTTwo, dXPMTThree,
          dXPMTFour, dXPMTFive, dXPMTSix, dXPMTSeven, dXPMTEight, dXPMTNine,
          dXPMTTen, nUserMuonCount);
cout<<nScintLength<<"\n"<<nScintWidth<<"\n"<<nScintHeight<<"\n"<<nNumberOfFibers
       <<"\n"<<fFiberMidpointPositionMatrix[0][0]<<"\n"
       <<fFiberMidpointPositionMatrix[0][1]<<"\n"<<fRadiusOfFibers<<"\n"<<dXZero
       <<"\n"<<dXOne<<"\n"<<dXTwo<<"\n"<<dXThree<<"\n"<<dXFour<<"\n"<<dXFive
       <<"\n"
       <<dXSix<<"\n"<<dXSeven<<"\n"<<dXEight<<"\n"<<dXNine<<"\n"<<dXTen<<"\n"
       <<fBoundaryOne<<"\n"<<fBoundaryTwo<<"\n"<<fEmitBoundaryOne<<"\n"
       <<fEmitBoundaryTwo<<"\n"<<fEfficiencyOfScint<<"\n"<<dXAbsorbZero<<"\n"<<
       dXAbsorbOne<<"\n"<<dXAbsorbTwo<<"\n"<<dXAbsorbThree<<"\n"<<dXAbsorbFour
```

```
        <<"\n"<<dXAbsorbFive<<"\n"<<dXAbsorbSix<<"\n"<<dXAbsorbSeven<<"\n"<<
        dXAbsorbEight<<"\n"<<dXAbsorbNine<<"\n"<<dXAbsorbTen<<"\n"<<dXEmitZero
        <<"\n"<<dXEmitOne<<"\n"<<dXEmitTwo<<"\n"<<dXEmitThree<<"\n"<<dXEmitFour
        <<"\n"
        <<dXEmitFive
        <<"\n"<<dXEmitSix<<"\n"<<dXEmitSeven<<"\n"<<dXEmitEight<<"\n"<<
        dXEmitNine<<
        "\n"<<dXEmitTen<<"\n"<<fLengthOfOutsideFiber<<"\n"<<dXPMTZero
        <<"\n"
        <<dXPMTOne<<"\n"<<dXPMTTwo<<"\n"<<dXPMTThree<<"\n"<<dXPMTFour<<"\n"<<
        dXPMTFive<<"\n"<<dXPMTSix<<"\n"<<dXPMTSeven<<"\n"<<dXPMTEight<<"\n"<<
        dXPMTNine<<"\n"<<dXPMTTen<<"\n"<<nUserMuonCount<<"\n";
}*/
```

```
//This module is totally complete and should run without errors.

//This is the module that determines where the muon impacts the scintillator.
//It returns a muon position and a muon velocity as well as counter values.
//This is the first module of the simulation of the program.

#include <stdlib.h>
#undef abs
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>


void MuonImpact(int nScintLength, int nScintWidth, int nScintHeight,
                double fMuonPositionMatrix[3], float fMuonVelocityMatrix[3],
                int& nProgCount, int& nMuonBreak)
{
      nMuonBreak = 0;
      do
      {
            fMuonVelocityMatrix[0] = (.5+(-.01*random(101)));
            fMuonVelocityMatrix[1] = (.5+(-.01*random(101)));
      }
   while((fabs(fMuonVelocityMatrix[0]) < .0001) | (fabs(fMuonVelocityMatrix[1])
         < .0001));
      fMuonVelocityMatrix[2] = -1;
      fMuonPositionMatrix[0] = random(nScintLength);
      fMuonPositionMatrix[1] = random(nScintWidth);
      fMuonPositionMatrix[2] = nScintHeight;
      nMuonBreak = 1;
      nProgCount = (nProgCount + 1);
}

//Error checking is not needed because it is impossible for the muon to contact
//the scintillator at any point that is not on it's surface.
```

```
//All testing is complete, this module works perfectly.

//This is the module that determines the length of the path taken by the muon
//after impacting the scintillator for the purpose of ion generation.
//Any path through a wave shifting fiber will be ignored.  This will output the
//length of the path as well as the exit point of the muon.
//This is the second module of the simulation of the program.

#include <stdlib.h>
#undef abs
#include <stdio.h>
#include <iostream.h>
#include <math.h>

#include <conio.h>
#include <complex.h>

void MuonPath(int nScintLength, int nScintWidth, int nScintHeight,
              double fMuonPositionMatrix[3], float fMuonVelocityMatrix[3],
              double fMuonFinalPositionMatrix[3],
              float fFiberMidpointPosition[10][2],
              float fRadiusOfFiber, float& fMuonPathLength,
              long& nNumberOfPhotons, float fEfficiencyOfScint,
              int nNumberOfFibers)
{
      int nCounter2 = 0;
      int nBreak = 0;
      float fXAdjustment = 0.;
      float fYAdjustment = 0.;
      float fZAdjustment = 0.;
      float fCartesianAdjustment = 0;
      int nCorrection1 = 0;
      int nCorrection2 = 0;


      //This reduces the chance of skipping over a wave shifting fiber.
      fMuonVelocityMatrix[0]= (.1*fMuonVelocityMatrix[0]);
      fMuonVelocityMatrix[1]= (.1*fMuonVelocityMatrix[1]);
      fMuonVelocityMatrix[2]= (.1*fMuonVelocityMatrix[2]);

      fMuonFinalPositionMatrix[0]=fMuonPositionMatrix[0];
      fMuonFinalPositionMatrix[1]=fMuonPositionMatrix[1];
      fMuonFinalPositionMatrix[2]=fMuonPositionMatrix[2];
      float fMuonVoidMatrix[3] = {0.,0.,0.};
      do
      {
            if(((fMuonFinalPositionMatrix[0] > 0) &
              (fMuonFinalPositionMatrix[0] < nScintLength)) &
              ((fMuonFinalPositionMatrix[1] > 0) &
              (fMuonFinalPositionMatrix[1] < nScintWidth)) &
              ((fMuonFinalPositionMatrix[2] > 0) &
              (fMuonFinalPositionMatrix[2] <= nScintHeight)))
            {
                  fMuonFinalPositionMatrix[0] = (fMuonFinalPositionMatrix[0] +
                                                 fMuonVelocityMatrix[0]);
                  fMuonFinalPositionMatrix[1] = (fMuonFinalPositionMatrix[1] +
```

```
                                    fMuonVelocityMatrix[1]);
        fMuonFinalPositionMatrix[2] = (fMuonFinalPositionMatrix[2] +
                                    fMuonVelocityMatrix[2]);
}
else
{
        nBreak = 1;
        if(fMuonFinalPositionMatrix[2] < 0)
        {
                float fZAdjustment = 0.;
                fZAdjustment = fabs((0-fMuonFinalPositionMatrix[2])/
                                fMuonVelocityMatrix[2]);
                fCartesianAdjustment = (1-fZAdjustment);
                fMuonFinalPositionMatrix[0]=(fMuonFinalPositionMatrix[0]
                        -(fZAdjustment*fMuonVelocityMatrix[0]));
                fMuonFinalPositionMatrix[1]=(fMuonFinalPositionMatrix[1]
                        -(fZAdjustment*fMuonVelocityMatrix[1]));
                fMuonFinalPositionMatrix[2]=(fMuonFinalPositionMatrix[2]
                        -(fZAdjustment*fMuonVelocityMatrix[2]));
        }
        if(fMuonFinalPositionMatrix[0] < 0)
        {
                if(fMuonVelocityMatrix[0] < -.0001)
                {
                        fXAdjustment = fabs((fMuonFinalPositionMatrix[0])/
                                        fMuonVelocityMatrix[0]);
                }
                else
                {
                        fXAdjustment = .0001;
                }
        }
        if(fMuonFinalPositionMatrix[0] > nScintLength)
        {
                if(fMuonVelocityMatrix[0] > .0001)
                {
                        fXAdjustment = fabs((fMuonFinalPositionMatrix[0] –
                                nScintLength)/fMuonVelocityMatrix[0]);
                }
                else
                {
                        fXAdjustment = .0001;
                }
        }
        if(fMuonFinalPositionMatrix[1] < 0)
        {
                if(fMuonVelocityMatrix[0] < -.0001)
                {
                        fYAdjustment = fabs((fMuonFinalPositionMatrix[1])/
                                        fMuonVelocityMatrix[1]);
                }
                else
                {
                        fYAdjustment = .0001;
                }
        }
        if(fMuonFinalPositionMatrix[1] > nScintWidth)
```

```
	{
		if(fMuonVelocityMatrix[0] > .0001)
		{
			fYAdjustment = fabs((fMuonFinalPositionMatrix[1] –
					nScintWidth)/fMuonVelocityMatrix[1]);
		}
		else
		{
			fYAdjustment = .0001;
		}
	}
	if(fXAdjustment >= fYAdjustment)
	{
		fMuonFinalPositionMatrix[0] =
		(fMuonFinalPositionMatrix[0] - (fMuonVelocityMatrix[0]
		* fXAdjustment));
		fMuonFinalPositionMatrix[1] =
		(fMuonFinalPositionMatrix[1] -(fMuonVelocityMatrix[1]
		* fXAdjustment));
		fMuonFinalPositionMatrix[2] =
		(fMuonFinalPositionMatrix[2] -(fMuonVelocityMatrix[2]
		* fXAdjustment));
		if(fXAdjustment != 0)
		{
			fCartesianAdjustment = (1-fXAdjustment);
		}
	}
	if(fXAdjustment < fYAdjustment)
	{
		fMuonFinalPositionMatrix[0] =
		(fMuonFinalPositionMatrix[0] -(fMuonVelocityMatrix[0] *
		fYAdjustment));
		fMuonFinalPositionMatrix[1] =
		(fMuonFinalPositionMatrix[1] -fMuonVelocityMatrix[1] *
		fYAdjustment));
		fMuonFinalPositionMatrix[2] =
		(fMuonFinalPositionMatrix[2] - (fMuonVelocityMatrix[2] *
		fYAdjustment));
		fCartesianAdjustment = (1-fYAdjustment);
	}
}
if(((fMuonFinalPositionMatrix[0] > 0) &
  (fMuonFinalPositionMatrix[0] < nScintLength)) &
  ((fMuonFinalPositionMatrix[1] > 0) &
  (fMuonFinalPositionMatrix[1] < nScintWidth)) &
  ((fMuonFinalPositionMatrix[2] > 0) &
  (fMuonFinalPositionMatrix[2] <= nScintHeight)))
{
}
else
{
	nBreak = 1;
	if(fMuonFinalPositionMatrix[2]<0)
	{
		fZAdjustment = fabs((0-fMuonFinalPositionMatrix[2])/
				fMuonVelocityMatrix[2]);
		fCartesianAdjustment = (1-fZAdjustment);
```

```
      fMuonFinalPositionMatrix[0]=(fMuonFinalPositionMatrix[0]
      -(fZAdjustment*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1]=(fMuonFinalPositionMatrix[1]
      -(fZAdjustment*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2]=(fMuonFinalPositionMatrix[2]
      -(fZAdjustment*fMuonVelocityMatrix[2]));
}
if(fMuonFinalPositionMatrix[0] < 0)
{
      if(fMuonVelocityMatrix[0] < -.0001)
      {
            fXAdjustment = fabs((fMuonFinalPositionMatrix[0])/
                              fMuonVelocityMatrix[0]);
      }
      else
      {
            fXAdjustment = .0001;
      }
}
if(fMuonFinalPositionMatrix[0] > nScintLength)
{
      if(fMuonVelocityMatrix[0] > .0001)
      {
            fXAdjustment = fabs((fMuonFinalPositionMatrix[0] –
                        nScintLength)/MuonVelocityMatrix[0]);
      }
      else
      {
            fXAdjustment = .0001;
      }
}
if(fMuonFinalPositionMatrix[1] < 0)
{
      if(fMuonVelocityMatrix[1] < -.0001)
      {
            fYAdjustment = fabs((fMuonFinalPositionMatrix[1])/
                              fMuonVelocityMatrix[1]);
      }
      else
      {
            fYAdjustment = .0001;
      }
}
if(fMuonFinalPositionMatrix[1] > nScintWidth)
{
      if(fMuonVelocityMatrix[0] > .0001)
      {
            fYAdjustment = fabs((fMuonFinalPositionMatrix[1] –
                        nScintWidth)/fMuonVelocityMatrix[1]);
      }
      else
      {
            fYAdjustment = .0001;
      }
}
if(fXAdjustment >= fYAdjustment)
{
```

```
            fMuonFinalPositionMatrix[0] =
            (fMuonFinalPositionMatrix[0] -(fMuonVelocityMatrix[0] *
            fXAdjustment));
            fMuonFinalPositionMatrix[1] =
            (fMuonFinalPositionMatrix[1] -(fMuonVelocityMatrix[1] *
            fXAdjustment));
            fMuonFinalPositionMatrix[2] =
           (fMuonFinalPositionMatrix[2] -(fMuonVelocityMatrix[2] *
            fXAdjustment));
            if(fXAdjustment != 0)
            {
                  fCartesianAdjustment = (1-fXAdjustment);
            }
      }
      if(fXAdjustment < fYAdjustment)
      {
            fMuonFinalPositionMatrix[0] =
           (fMuonFinalPositionMatrix[0] -(fMuonVelocityMatrix[0] *
            fYAdjustment));
            fMuonFinalPositionMatrix[1] =
            (fMuonFinalPositionMatrix[1] –
            (fMuonVelocityMatrix[1] * fYAdjustment));
            fMuonFinalPositionMatrix[2] =
            (fMuonFinalPositionMatrix[2] –(fMuonVelocityMatrix[2] *
            fYAdjustment));
            fCartesianAdjustment = (1-fYAdjustment);
      }
}




do
{
      if(nBreak != 1)
      {
      //Beginning of if statement block.
      if(sqrtl(((fMuonFinalPositionMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]) *
            (fMuonFinalPositionMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]))+
            ((fMuonFinalPositionMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])*
            (fMuonFinalPositionMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])))
            <fRadiusOfFiber)
  {
      if(sqrtl(((fMuonFinalPositionMatrix[1]-
            fMuonVelocityMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]) *
            (fMuonFinalPositionMatrix[1]- fMuonVelocityMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]))+
            ((fMuonFinalPositionMatrix[2]- fMuonVelocityMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])*
            (fMuonFinalPositionMatrix[2]- fMuonVelocityMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])))
```

```
<=fRadiusOfFiber)
{
      fMuonVoidMatrix[0] = (fMuonVoidMatrix[0] +
                            fabs(fMuonVelocityMatrix[0]));
      fMuonVoidMatrix[1] = (fMuonVoidMatrix[1] +
                            fabs(fMuonVelocityMatrix[1]));
      fMuonVoidMatrix[2] = (fMuonVoidMatrix[2] +
                            fabs(fMuonVelocityMatrix[2]));
}
else
{
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
fMuonVelocityMatrix[0]);
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
fMuonVelocityMatrix[1]);
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
fMuonVelocityMatrix[2]);
int nBreak2 = 0;
do
{
//Numerical approximation is necessary.
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      nBreak2 = 1;
}
else
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.1*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.1*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.1*fMuonVelocityMatrix[2]));
}
}
while(nBreak2 != 1);
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
(.1*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
(.1*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] =
```

```
(fMuonFinalPositionMatrix[2] -
(.1*fMuonVelocityMatrix[2]));
int nBreak3 = 0;
do
{
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      nBreak3 = 1;
}
else
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.01*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.01*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.01*fMuonVelocityMatrix[2]));
}
}
while(nBreak3 != 1);
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
(.01*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
(.01*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
(.01*fMuonVelocityMatrix[2]));
int nBreak4 = 0;
do
{
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      nBreak4 = 1;
}
else
{
```

```
                fMuonFinalPositionMatrix[0] =
                (fMuonFinalPositionMatrix[0] +

                (.001*fMuonVelocityMatrix[0]));
                fMuonFinalPositionMatrix[1] =
                (fMuonFinalPositionMatrix[1] +
                (.001*fMuonVelocityMatrix[1]));
                fMuonFinalPositionMatrix[2] =
                (fMuonFinalPositionMatrix[2] +
                (.001*fMuonVelocityMatrix[2]));
            }
            }
            while(nBreak4 != 1);

            fMuonFinalPositionMatrix[0] =
            (fMuonFinalPositionMatrix[0] -
            (.001*fMuonVelocityMatrix[0]));
            fMuonFinalPositionMatrix[1] =
            (fMuonFinalPositionMatrix[1] -
            (.001*fMuonVelocityMatrix[1]));
            fMuonFinalPositionMatrix[2] =
            (fMuonFinalPositionMatrix[2] -
            (.001*fMuonVelocityMatrix[2]));
            int nBreak5 = 0;
            do
            {
            if(sqrtl(((fMuonFinalPositionMatrix[1]-
                  fFiberMidpointPosition[nCounter2][0]) *
                  (fMuonFinalPositionMatrix[1]-
                  fFiberMidpointPosition[nCounter2][0]))+
                  ((fMuonFinalPositionMatrix[2]-
                  fFiberMidpointPosition[nCounter2][1])*
                  (fMuonFinalPositionMatrix[2]-
                  fFiberMidpointPosition[nCounter2][1])))
                  <fRadiusOfFiber)
            {
                  nBreak5 = 1;
            }
            else
            {
                  fMuonFinalPositionMatrix[0] =
                  (fMuonFinalPositionMatrix[0] +
                  (.0001*fMuonVelocityMatrix[0]));
                  fMuonFinalPositionMatrix[1] =
                  (fMuonFinalPositionMatrix[1] +
                  (.0001*fMuonVelocityMatrix[1]));
                  fMuonFinalPositionMatrix[2] =
                  (fMuonFinalPositionMatrix[2] +
                  (.0001*fMuonVelocityMatrix[2]));
            }
            }
            while(nBreak5 != 1);
            }
            //We have reached the limit of compiler accuracy.
      }
      else
      {
```

```
if((sqrtl(((fMuonFinalPositionMatrix[1]-
fMuonVelocityMatrix[1]-
fFiberMidpointPosition[nCounter2][0]) *
(fMuonFinalPositionMatrix[1]- fMuonVelocityMatrix[1]-
fFiberMidpointPosition[nCounter2][0]))+
((fMuonFinalPositionMatrix[2]- fMuonVelocityMatrix[2]-
fFiberMidpointPosition[nCounter2][1])*
(fMuonFinalPositionMatrix[2]- fMuonVelocityMatrix[2]-
fFiberMidpointPosition[nCounter2][1])))
<fRadiusOfFiber)&(nCorrection1 != 1))
{
nCorrection1 = 1;
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
fMuonVelocityMatrix[0]);
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
fMuonVelocityMatrix[1]);
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
fMuonVelocityMatrix[2]);
int nBreak2 = 0;
do
{
//Numerical approximation is necessary.
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{

      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.1*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.1*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.1*fMuonVelocityMatrix[2]));
      fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                        fabs(.1*fMuonVelocityMatrix[0]));
      fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                        fabs(.1*fMuonVelocityMatrix[1]));
      fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                        fabs(.1*fMuonVelocityMatrix[2]));
}
else
{
      nBreak2 = 1;
}
}
```

```
while(nBreak2 != 1);
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
(.1*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
(.1*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
(.1*fMuonVelocityMatrix[2]));
fMuonVoidMatrix[0]= (fMuonVoidMatrix[0] –
                    fabs(.1*fMuonVelocityMatrix[0]));
fMuonVoidMatrix[1]= (fMuonVoidMatrix[1] –
                    fabs(.1*fMuonVelocityMatrix[1]));
fMuonVoidMatrix[2]= (fMuonVoidMatrix[2] -
                    fabs(.1*fMuonVelocityMatrix[2]));
int nBreak3 = 0;
do
{
      if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.01*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.01*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.01*fMuonVelocityMatrix[2]));
      fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                        fabs(.01*fMuonVelocityMatrix[0]));
      fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                        fabs(.01*fMuonVelocityMatrix[1]));
      fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                        fabs(.01*fMuonVelocityMatrix[2]));
}
else
{
      nBreak3 = 1;
}
}
while(nBreak3 != 1);
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
(.01*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
(.01*fMuonVelocityMatrix[1]));
```

```
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
(.01*fMuonVelocityMatrix[2]));
fMuonVoidMatrix[0]= (fMuonVoidMatrix[0] –
                    fabs(.01*fMuonVelocityMatrix[0]));
fMuonVoidMatrix[1]= (fMuonVoidMatrix[1] –
                    fabs(.01*fMuonVelocityMatrix[1]));
fMuonVoidMatrix[2]= (fMuonVoidMatrix[2] -
                    fabs(.01*fMuonVelocityMatrix[2]));
int nBreak4 = 0;
do
{
      if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.001*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.001*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.001*fMuonVelocityMatrix[2]));
      fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                     fabs(.001*fMuonVelocityMatrix[0]));
      fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                     fabs(.001*fMuonVelocityMatrix[1]));
      fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                     fabs(.001*fMuonVelocityMatrix[2]));
}
else
{
      nBreak4 = 1;
}
}
while(nBreak4 != 1);
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
(.001*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
(.001*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
(.001*fMuonVelocityMatrix[2]));
fMuonVoidMatrix[0]= (fMuonVoidMatrix[0] –
                    fabs(.001*fMuonVelocityMatrix[0]));
fMuonVoidMatrix[1]= (fMuonVoidMatrix[1] –
                    fabs(.001*fMuonVelocityMatrix[1]));
```

```
      fMuonVoidMatrix[2]= (fMuonVoidMatrix[2] –
                           fabs(.001*fMuonVelocityMatrix[2]));
      int nBreak5 = 0;
      do
      {
            if(sqrtl(((fMuonFinalPositionMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]) *
            (fMuonFinalPositionMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]))+
            ((fMuonFinalPositionMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])*
            (fMuonFinalPositionMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])))
            <fRadiusOfFiber)
      {
            fMuonFinalPositionMatrix[0] =
            (fMuonFinalPositionMatrix[0] +
            (.0001*fMuonVelocityMatrix[0]));
            fMuonFinalPositionMatrix[1] =
            (fMuonFinalPositionMatrix[1] +
            (.0001*fMuonVelocityMatrix[1]));
            fMuonFinalPositionMatrix[2] =
            (fMuonFinalPositionMatrix[2] +
            (.0001*fMuonVelocityMatrix[2]));
            fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                        fabs(.0001*fMuonVelocityMatrix[0]));
            fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                        fabs(.0001*fMuonVelocityMatrix[1]));
            fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                        fabs(.0001*fMuonVelocityMatrix[2]));
      }

      else
      {
            nBreak5 = 1;
      }
      }
      while(nBreak5 != 1);
      }
}
//End of if statement block.
}
if(nBreak == 1)
{
//Beginning of if statement block.
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      (fMuonVelocityMatrix[1]*
```

```
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]- (fMuonVelocityMatrix[1] *
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]- (fMuonVelocityMatrix[2] *
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]- (fMuonVelocityMatrix[2] *
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      fMuonVoidMatrix[0] = (fMuonVoidMatrix[0] +
                           fabs((fMuonVelocityMatrix[0]*
                           (fCartesianAdjustment))));
      fMuonVoidMatrix[1] = (fMuonVoidMatrix[1] +
                           fabs((fMuonVelocityMatrix[1]*
                           (fCartesianAdjustment))));
      fMuonVoidMatrix[2] = (fMuonVoidMatrix[2] +
                           fabs((fMuonVelocityMatrix[2]*
                           (fCartesianAdjustment))));
}
else
{
nBreak = 0;
fMuonFinalPositionMatrix[0] = (fMuonFinalPositionMatrix[0] –
                              fabs((fMuonVelocityMatrix[0]*
                              (fCartesianAdjustment))));
fMuonFinalPositionMatrix[1] = (fMuonFinalPositionMatrix[1] –
                              fabs((fMuonVelocityMatrix[1]*
                              (fCartesianAdjustment))));
fMuonFinalPositionMatrix[2] = (fMuonFinalPositionMatrix[2] –
                              fabs((fMuonVelocityMatrix[2]*
                              (fCartesianAdjustment))));
int nBreak2 = 0;
do
{
//Numerical approximation is necessary.
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      nBreak2 = 1;
}
else
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.1*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
```

```
      (fMuonFinalPositionMatrix[1] +

      (.1*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.1*fMuonVelocityMatrix[2]));
}
}
while(nBreak2 != 1);
fMuonFinalPositionMatrix[0] = (fMuonFinalPositionMatrix[0] -
                              (.1*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] = (fMuonFinalPositionMatrix[1] -
                              (.1*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] = (fMuonFinalPositionMatrix[2] -
                              (.1*fMuonVelocityMatrix[2]));
int nBreak3 = 0;
do
{
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      nBreak3 = 1;
}
else
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.01*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.01*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.01*fMuonVelocityMatrix[2]));
}
}

while(nBreak3 != 1);
fMuonFinalPositionMatrix[0] = (fMuonFinalPositionMatrix[0] –
                              (.01*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] = (fMuonFinalPositionMatrix[1] –
                              (.01*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] = (fMuonFinalPositionMatrix[2] –
                              (.01*fMuonVelocityMatrix[2]));
int nBreak4 = 0;
do
{
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
```

```
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      nBreak4 = 1;
}
else
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.001*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.001*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.001*fMuonVelocityMatrix[2]));
}
}
while(nBreak4 != 1);
fMuonFinalPositionMatrix[0] = (fMuonFinalPositionMatrix[0] –
                              (.001*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] = (fMuonFinalPositionMatrix[1] -
                              (.001*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] = (fMuonFinalPositionMatrix[2] -
                              (.001*fMuonVelocityMatrix[2]));
int nBreak5 = 0;
do
{
if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      nBreak5 = 1;
}
else
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.0001*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.0001*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.0001*fMuonVelocityMatrix[2]));
}
}
```

```
while(nBreak5 != 1);
}
}
else
{
if((sqrtl(((fMuonFinalPositionMatrix[1]-
      (fMuonVelocityMatrix[1]*
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]- (fMuonVelocityMatrix[1] *
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]- (fMuonVelocityMatrix[2] *
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]- (fMuonVelocityMatrix[2] *
      (fCartesianAdjustment)) –
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)&(nCorrection2 != 1))
{
      nCorrection2 = 1;
      nBreak = 0;
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] -
      fabs((fMuonVelocityMatrix[0]*
      (fCartesianAdjustment))));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] -
      fabs((fMuonVelocityMatrix[1]*
      (fCartesianAdjustment))));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] -
      fabs((fMuonVelocityMatrix[2]*
      (fCartesianAdjustment))));
      int nBreak2 = 0;
      do
      {
      //Numerical approximation is necessary.
      if(sqrtl(((fMuonFinalPositionMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]) *
            (fMuonFinalPositionMatrix[1]-
            fFiberMidpointPosition[nCounter2][0]))+
            ((fMuonFinalPositionMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])*
            (fMuonFinalPositionMatrix[2]-
            fFiberMidpointPosition[nCounter2][1])))
            <fRadiusOfFiber)
      {
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.1*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.1*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.1*fMuonVelocityMatrix[2]));
```

```
fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                      fabs(.1*fMuonVelocityMatrix[0]));
fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                      fabs(.1*fMuonVelocityMatrix[1]));
fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                      fabs(.1*fMuonVelocityMatrix[2]));
}
else
{
      nBreak2 = 1;
}
}
while(nBreak2 != 1);
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
(.1*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
(.1*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
(.1*fMuonVelocityMatrix[2]));
fMuonVoidMatrix[0]= (fMuonFinalPositionMatrix[0] –
                     fabs(.1*fMuonVelocityMatrix[0]));
fMuonVoidMatrix[1]= (fMuonFinalPositionMatrix[1] -
                     fabs(.1*fMuonVelocityMatrix[1]));
fMuonVoidMatrix[2]= (fMuonFinalPositionMatrix[2] –
                     fabs(.1*fMuonVelocityMatrix[2]));
int nBreak3 = 0;
do
{
      if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.01*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.01*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.01*fMuonVelocityMatrix[2]));
      fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                            fabs(.01*fMuonVelocityMatrix[0]));
      fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                            fabs(.01*fMuonVelocityMatrix[1]));
      fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                            fabs(.01*fMuonVelocityMatrix[2]));
}
```

```
      else
{
      nBreak3 = 1;
}
}
while(nBreak3 != 1);
fMuonFinalPositionMatrix[0] =
(fMuonFinalPositionMatrix[0] -
(.01*fMuonVelocityMatrix[0]));
fMuonFinalPositionMatrix[1] =
(fMuonFinalPositionMatrix[1] -
(.01*fMuonVelocityMatrix[1]));
fMuonFinalPositionMatrix[2] =
(fMuonFinalPositionMatrix[2] -
(.01*fMuonVelocityMatrix[2]));
fMuonVoidMatrix[0]= (fMuonFinalPositionMatrix[0] –
                    fabs(.01*fMuonVelocityMatrix[0]));
fMuonVoidMatrix[1]= (fMuonFinalPositionMatrix[1] -
                    fabs(.01*fMuonVelocityMatrix[1]));
fMuonVoidMatrix[2]= (fMuonFinalPositionMatrix[2] -
                    fabs(.01*fMuonVelocityMatrix[2]));
int nBreak4 = 0;
do
{
      if(sqrtl(((fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]) *
      (fMuonFinalPositionMatrix[1]-
      fFiberMidpointPosition[nCounter2][0]))+
      ((fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])*
      (fMuonFinalPositionMatrix[2]-
      fFiberMidpointPosition[nCounter2][1])))
      <fRadiusOfFiber)
{
      fMuonFinalPositionMatrix[0] =
      (fMuonFinalPositionMatrix[0] +
      (.001*fMuonVelocityMatrix[0]));
      fMuonFinalPositionMatrix[1] =
      (fMuonFinalPositionMatrix[1] +
      (.001*fMuonVelocityMatrix[1]));
      fMuonFinalPositionMatrix[2] =
      (fMuonFinalPositionMatrix[2] +
      (.001*fMuonVelocityMatrix[2]));
      fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                    fabs(.001*fMuonVelocityMatrix[0]));
      fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                    fabs(.001*fMuonVelocityMatrix[1]));
      fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                    fabs(.001*fMuonVelocityMatrix[2]));
}
else
{
      nBreak4 = 1;
}
}
while(nBreak4 != 1);
fMuonFinalPositionMatrix[0] =
```

```
                  (fMuonFinalPositionMatrix[0] -
                  (.001*fMuonVelocityMatrix[0]));
                  fMuonFinalPositionMatrix[1] =
                  (fMuonFinalPositionMatrix[1] -
                  (.001*fMuonVelocityMatrix[1]));
                  fMuonFinalPositionMatrix[2] =
                  (fMuonFinalPositionMatrix[2] -
                  (.001*fMuonVelocityMatrix[2]));
                  fMuonVoidMatrix[0]= (fMuonFinalPositionMatrix[0] -
                                      fabs(.001*fMuonVelocityMatrix[0]));
                  fMuonVoidMatrix[1]= (fMuonFinalPositionMatrix[1] -
                                      fabs(.001*fMuonVelocityMatrix[1]));
                  fMuonVoidMatrix[2]= (fMuonFinalPositionMatrix[2] -
                                      fabs(.001*fMuonVelocityMatrix[2]));
                  int nBreak5 = 0;
                  do
                  {
                        if(sqrtl(((fMuonFinalPositionMatrix[1]-
                        fFiberMidpointPosition[nCounter2][0]) *
                        (fMuonFinalPositionMatrix[1]-
                        fFiberMidpointPosition[nCounter2][0]))+
                        ((fMuonFinalPositionMatrix[2]-
                        fFiberMidpointPosition[nCounter2][1])*
                        (fMuonFinalPositionMatrix[2]-
                        fFiberMidpointPosition[nCounter2][1])))
                        <fRadiusOfFiber)
                  {

                        fMuonFinalPositionMatrix[0] =
                        (fMuonFinalPositionMatrix[0] +
                        (.0001*fMuonVelocityMatrix[0]));
                        fMuonFinalPositionMatrix[1] =
                        (fMuonFinalPositionMatrix[1] +
                        (.0001*fMuonVelocityMatrix[1]));
                        fMuonFinalPositionMatrix[2] =
                        (fMuonFinalPositionMatrix[2] +
                        (.0001*fMuonVelocityMatrix[2]));
                        fMuonVoidMatrix[0] = (fMuonVoidMatrix[0]+
                                    fabs(.0001*fMuonVelocityMatrix[0]));
                        fMuonVoidMatrix[1] = (fMuonVoidMatrix[1]+
                                    fabs(.0001*fMuonVelocityMatrix[1]));
                        fMuonVoidMatrix[2] = (fMuonVoidMatrix[2]+
                                    fabs(.0001*fMuonVelocityMatrix[2]));
                  }
                  else
                  {
                        nBreak5 = 1;
                  }
                  }
                  while(nBreak5 != 1);
                  }
            }
            }
            //End of if statement block.
            nCounter2++;
      }
      while(nCounter2 != (nNumberOfFibers));
```

```
            nCounter2 = 0;
        }
        while(nBreak != 1);


        //Calculate the total nonvetoed length through which the muon passed.
        fMuonPathLength = sqrtl(((fabs(fMuonFinalPositionMatrix[0]-
                                 fMuonPositionMatrix[0])
                                 -fMuonVoidMatrix[0])*
                                (fabs(fMuonFinalPositionMatrix[0]-
                                fMuonPositionMatrix[0])-
                                fMuonVoidMatrix[0]))+
                                ((fabs(fMuonFinalPositionMatrix[1]-
                                fMuonPositionMatrix[1])
                                -fMuonVoidMatrix[1])*
                                (fabs(fMuonFinalPositionMatrix[1]-
                                fMuonPositionMatrix[1])-
                                fMuonVoidMatrix[1]))+
                                ((fabs(fMuonFinalPositionMatrix[2]-
                                fMuonPositionMatrix[2])
                                -fMuonVoidMatrix[2])*
                                (fabs(fMuonFinalPositionMatrix[2]-
                                fMuonPositionMatrix[2])-
                                fMuonVoidMatrix[2])));
        nNumberOfPhotons = (fMuonPathLength/8)*fEfficiencyOfScint*1837*1000)/68.);
}
```

```
//This module works perfectly.

//This module will generate a random photon velocity
//vector and a random photon intial location somewhere on the muon path.  The
//module will then assign the photon a wavelength based on a normalized emission
//spectrum.  The output will be the variables describing the photon as well as
//a count showing that a photon has been created.

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>



void PhotonCreation(double fMuonPositionMatrix[3],
                    double fMuonFinalPositionMatrix[3],
                    float fMuonVelocityMatrix[3], int& nPhotonCount,
                    float fRadiusOfFiber,
                    float fFiberMidpointPosition[10][2],
                    double fPhotonPositionMatrix[3],
                    double fPhotonVelocityMatrix[3], int& nPhotonWavelength,
                    double dXTen, double dXNine, double dXEight, double dXSeven,
                    double dXSix, double dXFive, double dXFour, double dXThree,
                    double dXTwo, double dXOne, double dXZero,
                    int nScintHeight, float fBoundaryOne, float fBoundaryTwo,
                    int nNumberOfFibers)
{
      int nCounter = 0;
      int nBreak = 0;
      int nBreak2 = 0;
      int x = 0;
      float y = 0.;

      //These loops create a photon that is inside the scintillator and void any
      //photons that are created inside the fibers.
       do
       {
            do
            {
            //do some error checking thing here for other fibers

            fPhotonPositionMatrix[2] = (nScintHeight –
                                       (.1*random(10*(fMuonPositionMatrix[2]-
                                       fMuonFinalPositionMatrix[2]))));
            fPhotonPositionMatrix[1] = ((fMuonPositionMatrix[1]+(
                                       (nScintHeight-fPhotonPositionMatrix[2])*
                                       fMuonVelocityMatrix[1])));
            fPhotonPositionMatrix[0] = ((fMuonPositionMatrix[0]+(
                                       (nScintHeight-fPhotonPositionMatrix[2])*
                                       fMuonVelocityMatrix[0])));
            }
            while((fPhotonPositionMatrix[1]<.0001) &
```

```
            (fPhotonPositionMatrix[2]<.0001));
nCounter = 0;
      do
      {
            if((((float)fPhotonPositionMatrix[1]-
                 fFiberMidpointPosition[nCounter][0])*
                 ((float)fPhotonPositionMatrix[1]-
                 fFiberMidpointPosition[nCounter][0]))< .0001)

            {

                  fPhotonPositionMatrix[1] =
                  (fFiberMidpointPosition[nCounter][0] + .0001);
            }
       //   cout<<fPhotonPositionMatrix[2]<<"
                  "<<fFiberMidpointPosition[nCounter][1]<<"\n";
       //   cout<<nCounter<<"\n";
       //   getch();
            if((((float)fPhotonPositionMatrix[2]-
                 fFiberMidpointPosition[nCounter][1])*
                 ((float)fPhotonPositionMatrix[2]-
                 fFiberMidpointPosition[nCounter][1]))< .0001)
            {
                  fPhotonPositionMatrix[2] =
                  (fFiberMidpointPosition[nCounter][1] + .0001);
            }
            if(sqrt((((float)fPhotonPositionMatrix[1]-
                  fFiberMidpointPosition[nCounter][0]) *
                  ((float)fPhotonPositionMatrix[1]-
                  fFiberMidpointPosition[nCounter][0]))+
                  (((float)fPhotonPositionMatrix[2]-
                  fFiberMidpointPosition[nCounter][1])*
                  ((float)fPhotonPositionMatrix[2]-
                  fFiberMidpointPosition[nCounter][1]))) <=
                  fRadiusOfFiber)
            {
                  nCounter = (nNumberOfFibers - 1);
            }
            else
            {
                  if(nCounter == (nNumberOfFibers-1))
                  {
              // cout<<fPhotonPositionMatrix[1]<<"
                        "<<fPhotonPositionMatrix[2]<<"\n";
              //  getch();
                  do
                  {
                        nBreak = 1;
                        fPhotonVelocityMatrix[0]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[1]= .001*(random(801)-400);

                        fPhotonVelocityMatrix[2]= .001*(random(801)-400);
                  }

while(((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0]) == 0) &
      ((fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1]) == 0) &
      ((fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]) == 0));
```

```
            }
            }
                  nCounter++;
            }
            while(nCounter != nNumberOfFibers);
      }
      while(nBreak != 1);
      nPhotonCount++;


      //Assign the photon a wavelength at this point.
      do
      {
            x = (random((fBoundaryTwo - fBoundaryOne + 1)) + fBoundaryOne);
            y = .001*random(1000);
            if(((dXZero)+(dXOne*x)+(dXTwo*x*x)+(dXThree*x*x*x)+
                  (dXFour*x*x*x*x)+(dXFive*x*x*x*x*x)+(dXSix*x*x*x*x*x*x)+
                  (dXSeven*x*x*x*x*x*x*x)+(dXEight*x*x*x*x*x*x*x*x)+
                  (dXNine*x*x*x*x*x*x*x*x*x)+(dXTen*x*x*x*x*x*x*x*x*x*x))>y)
            {
                  nBreak2 = 1;
            }
      }
      while(nBreak2 != 1);


      nPhotonWavelength = x;
}

/*void main()
{
      double fMuonPositionMatrix[3] = {20,10,8.};
      double fMuonFinalPositionMatrix[3] = {20,10,0.};
      float fMuonVelocityMatrix[3] = {0,0,-1.};
      int nPhotonCount = 0;
      float fRadiusOfFiber = 1;
      float fFiberMidpointPosition[10][2] =
{{10.,4.},{0.,0.},{0.,0.},{0.,0.},{0.,0.},{0.,0.},

      {0.,0.},{0.,0.},{0.,0.},{0.,0.}};
      double fPhotonPositionMatrix[3] = {0.,0.,0.};
      double fPhotonVelocityMatrix[3] = {0.,0.,0.};
      int nPhotonWavelength = 0;
      double dXTen = 0.;
      double dXNine = 0.;
      double dXEight = 0.;
      double dXSeven = 0.;
      double dXSix = 0.;
      double dXFive = 0.;
      double dXFour = 0.;
      double dXThree = 0.;
      double dXTwo = 0.;
      double dXOne = 0.;
      double dXZero = 1.;
      int nScintHeight = 8;
      float fBoundaryOne = 1.;
      float fBoundaryTwo = 1.;
```

```
	int nNumberOfFibers = 1;
	int nCounter2 = 0;
	do
	{
		PhotonCreation(fMuonPositionMatrix, fMuonFinalPositionMatrix,
				fMuonVelocityMatrix, nPhotonCount, fRadiusOfFiber,
				fFiberMidpointPosition, fPhotonPositionMatrix,
				fPhotonVelocityMatrix, nPhotonWavelength, dXTen,
				dXNine, dXEight, dXSeven, dXSix, dXFive, dXFour,
				dXThree, dXTwo, dXOne, dXZero, nScintHeight,
				fBoundaryOne, fBoundaryTwo, nNumberOfFibers);
		nCounter2++;
	}
	while(nCounter2 != 100);
}*/
```

```
//This module works perfectly.


//This module will follow the photon as it bounces around inside the
//scintillator.  The module will deal with reflections off the walls of the
//scintillator as well as determining whether the photon is absorbed or
//transmitted.  This module will calculate the total path length the the photon
//has traversed and will terminate when the photon hits contacts a wave shifting
//fiber.

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>



int nPhotonReflection(int nScintLength, int nScintWidth, int nScintHeight,
                      float fRadiusOfFiber,
                      float fFiberMidpointPosition[10][2],
                      double fPhotonPositionMatrix[3],
                      double fPhotonVelocityMatrix[3], double& dTotalPhotonPath,
                      int& nWallImpact, int& nFiberSelecter, int
                      nNumberOfFibers)
{
      float fXAdjustment = 0.;
      float fYAdjustment = 0.;
      float fZAdjustment = 0.;
      int nCounter = 0;
      int nEndOfModule = 0;
      int nContactFiber = 0;
      do
      {
            if(((fPhotonPositionMatrix[0] >= -.00001) &
              (fPhotonPositionMatrix[0] <= (nScintLength+.00001))) &
              ((fPhotonPositionMatrix[1] >= -.00001) &
              (fPhotonPositionMatrix[1] <= (nScintWidth+.00001))) &
              ((fPhotonPositionMatrix[2] >= -.00001) &
              (fPhotonPositionMatrix[2] <= (nScintHeight+.00001)))&(nEndOfModule
              != 1))
            {
                  fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
                                              fPhotonVelocityMatrix[0]);
                  fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
                                              fPhotonVelocityMatrix[1]);
                  fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                                              fPhotonVelocityMatrix[2]);
                  dTotalPhotonPath = (dTotalPhotonPath +
                                      sqrtl((fPhotonVelocityMatrix[0]*
                                      PhotonVelocityMatrix[0])+
                                      (fPhotonVelocityMatrix[1]
                                      *fPhotonVelocityMatrix[1])+
                                      (fPhotonVelocityMatrix[2]*
                                      fPhotonVelocityMatrix[2])));
```

```
do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]) *
            (fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]))+
            ((fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])*
            (fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])))
            <=fRadiusOfFiber)
      {
            nContactFiber = 1;
            nFiberSelecter = nCounter;
            fPhotonPositionMatrix[0] =
            (fPhotonPositionMatrix[0] -
            fPhotonVelocityMatrix[0]);
            fPhotonPositionMatrix[1] =
            (fPhotonPositionMatrix[1] -
            fPhotonVelocityMatrix[1]);
            fPhotonPositionMatrix[2] =
            (fPhotonPositionMatrix[2] -
            fPhotonVelocityMatrix[2]);
            dTotalPhotonPath = (dTotalPhotonPath –
            sqrtl((fPhotonVelocityMatrix[0]*
                  fPhotonVelocityMatrix[0])+
                  (fPhotonVelocityMatrix[1]
                  *fPhotonVelocityMatrix[1])+
                  (fPhotonVelocityMatrix[2]*
                  fPhotonVelocityMatrix[2])));;
            //Numerical Approximation.
            int nBreak2 = 0;

            do
            {
            if(sqrtl(((fPhotonPositionMatrix[1]-
                  fFiberMidpointPosition[nCounter][0]) *
                  (fPhotonPositionMatrix[1]-
                  fFiberMidpointPosition[nCounter][0]))+
                  ((fPhotonPositionMatrix[2]-
                  fFiberMidpointPosition[nCounter][1])*
                  (fPhotonPositionMatrix[2]-
                  fFiberMidpointPosition[nCounter][1])))
                  <fRadiusOfFiber)
            {

                  nBreak2 = 1;
            }
            else
            {
                  fPhotonPositionMatrix[0] =
                  (fPhotonPositionMatrix[0] +
                  (.1*fPhotonVelocityMatrix[0]));
                  fPhotonPositionMatrix[1] =
                  (fPhotonPositionMatrix[1] +
                  (.1*fPhotonVelocityMatrix[1]));
                  fPhotonPositionMatrix[2] =
```

```
      (fPhotonPositionMatrix[2] +
      (.1*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath +
      sqrtl((.1*fPhotonVelocityMatrix[0]*
      .1*fPhotonVelocityMatrix[0])+
      (.1*fPhotonVelocityMatrix[1]
      *.1*fPhotonVelocityMatrix[1])+
      (.1*fPhotonVelocityMatrix[2]*
      .1*fPhotonVelocityMatrix[2])));
}
}
while(nBreak2 != 1);

fPhotonPositionMatrix[0] =
(fPhotonPositionMatrix[0] -
(.1*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] =
(fPhotonPositionMatrix[1] -
(.1*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] =
(fPhotonPositionMatrix[2] -
(.1*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
sqrtl((.1*fPhotonVelocityMatrix[0]*
      .1*fPhotonVelocityMatrix[0])+
      (.1*fPhotonVelocityMatrix[1]
      *.1*fPhotonVelocityMatrix[1])+
      (.1*fPhotonVelocityMatrix[2]*
      .1*fPhotonVelocityMatrix[2])));
int nBreak3 = 0;
do
{
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])))
      <fRadiusOfFiber)
{
      nBreak3 = 1;
}
else
{
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] +
      (.01*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] +
      (.01*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] +
      (.01*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath +
      sqrtl((.01*fPhotonVelocityMatrix[0]*
```

```
            .01*fPhotonVelocityMatrix[0])+
            (.01*fPhotonVelocityMatrix[1]
            *.01*fPhotonVelocityMatrix[1])+
            (.01*fPhotonVelocityMatrix[2]*
            .01*fPhotonVelocityMatrix[2])));
}
}
while(nBreak3 != 1);

fPhotonPositionMatrix[0] =
(fPhotonPositionMatrix[0] -
(.01*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] =
(fPhotonPositionMatrix[1] -
(.01*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] =
(fPhotonPositionMatrix[2] -
(.01*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
sqrtl((.01*fPhotonVelocityMatrix[0]*
.01*fPhotonVelocityMatrix[0])+
(.01*fPhotonVelocityMatrix[1]
*.01*fPhotonVelocityMatrix[1])+
(.01*fPhotonVelocityMatrix[2]*
.01*fPhotonVelocityMatrix[2])));
int nBreak4 = 0;
do
{
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])))
      <fRadiusOfFiber)
{
      nBreak4 = 1;
}
else
{
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] +
      (.001*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] +
      (.001*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] +
      (.001*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath +
      sqrtl((.001*fPhotonVelocityMatrix[0]*
            .001*fPhotonVelocityMatrix[0])+
            (.001*fPhotonVelocityMatrix[1]
            *.001*fPhotonVelocityMatrix[1])+
            (.001*fPhotonVelocityMatrix[2]
```

```
            *.001*fPhotonVelocityMatrix[2])));
}
}
while(nBreak4 != 1);

fPhotonPositionMatrix[0] =
(fPhotonPositionMatrix[0] -
(.001*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] =
(fPhotonPositionMatrix[1] -
(.001*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] =
(fPhotonPositionMatrix[2] -
(.001*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
sqrtl((.001*fPhotonVelocityMatrix[0]*
      .001*fPhotonVelocityMatrix[0])
      +(.001*fPhotonVelocityMatrix[1]
      *.001*fPhotonVelocityMatrix[1])+
      (.001*fPhotonVelocityMatrix[2]
      *.001*fPhotonVelocityMatrix[2])));
int nBreak5 = 0;
do
{
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])))
      <fRadiusOfFiber)
{
      nBreak5 = 1;
}
else
{
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] +
      (.0001*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] +
      (.0001*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] +
      (.0001*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath +
      sqrtl((.0001*fPhotonVelocityMatrix[0]*
      .0001*fPhotonVelocityMatrix[0])+
      (.0001*fPhotonVelocityMatrix[1]
      *.0001*fPhotonVelocityMatrix[1])+
      (.0001*fPhotonVelocityMatrix[2]
      *.0001*fPhotonVelocityMatrix[2])));
}
}
while(nBreak5 != 1);
```

```
                        fPhotonPositionMatrix[0] =
                        (fPhotonPositionMatrix[0] -
                        (.0001*fPhotonVelocityMatrix[0]));
                        fPhotonPositionMatrix[1] =
                        (fPhotonPositionMatrix[1] -
                        (.0001*fPhotonVelocityMatrix[1]));
                        fPhotonPositionMatrix[2] =
                        (fPhotonPositionMatrix[2] -
                        (.0001*fPhotonVelocityMatrix[2]));
                        dTotalPhotonPath = (dTotalPhotonPath –
                        sqrtl((.0001*fPhotonVelocityMatrix[0]*
                        .0001*fPhotonVelocityMatrix[0])
                        +(.0001*fPhotonVelocityMatrix[1]
                        *.0001*fPhotonVelocityMatrix[1])+
                        (.0001*fPhotonVelocityMatrix[2]
                        *.0001*fPhotonVelocityMatrix[2])));
                        //End of numerical approximation.
                        nEndOfModule = 1;
                  }
                  nCounter++;
            }
            while(nCounter != nNumberOfFibers);
            nCounter = 0;
      }
      else
      {
            nEndOfModule = 1;
            if(fPhotonPositionMatrix[0] < -.000001)
            {
                  if(fPhotonVelocityMatrix[0] < -.0001)
                  {
                        fXAdjustment = fabs((fPhotonPositionMatrix[0])/
                                    fPhotonVelocityMatrix[0]);
                  }
                  else
                  {
                        fXAdjustment = .0001;
                  }
                  nWallImpact = 1;
            }
            if(fPhotonPositionMatrix[0] > (nScintLength+.000001))
            {
                  if(fPhotonVelocityMatrix[0] > .0001)
                  {
                        fXAdjustment = fabs((fPhotonPositionMatrix[0] –
                                    nScintLength)/
                                    fPhotonVelocityMatrix[0]);
                  }
                  else
                  {
                        fXAdjustment = .0001;
                  }
                  nWallImpact = 2;
            }
            if(fPhotonPositionMatrix[1] < -.000001)
            {
```

```
      if(fPhotonVelocityMatrix[1] < -.0001)
      {
            fYAdjustment = fabs((fPhotonPositionMatrix[1])/
                           fPhotonVelocityMatrix[1]);
      }
      else
      {
            fYAdjustment = .0001;
      }
      nWallImpact = 3;
}
if(fPhotonPositionMatrix[1] > (nScintWidth+.000001))
{
      if(fPhotonVelocityMatrix[1] > .0001)
      {
            fYAdjustment = fabs((fPhotonPositionMatrix[1] –
                           nScintWidth)/
                           fPhotonVelocityMatrix[1]);
      }
      else
      {
            fYAdjustment = .0001;
      }
      nWallImpact = 4;
}
if(fPhotonPositionMatrix[2] < -.000001)
{
      if(fPhotonVelocityMatrix[2] < -.0001)
      {
            fZAdjustment = fabs((fPhotonPositionMatrix[2])/
                           fPhotonVelocityMatrix[2]);
      }
      else
      {
            fZAdjustment = .0001;
      }
      nWallImpact = 5;
}
if(fPhotonPositionMatrix[2] > (nScintHeight+.000001))
{
      if(fPhotonVelocityMatrix[2] > .0001)
      {
            fZAdjustment = fabs((fPhotonPositionMatrix[2]-
                           nScintHeight)/
                           fPhotonVelocityMatrix[2]);
      }
      else
      {
fZAdjustment = .0001;
      }
      nWallImpact = 6;
}
if((fXAdjustment >= fYAdjustment)&(fXAdjustment >=
    fZAdjustment))
{
      fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] –
                                 (fPhotonVelocityMatrix[0] *
```

```
                                    fXAdjustment));
      fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                                    (fPhotonVelocityMatrix[1] *
                                    fXAdjustment));
      fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                                    (fPhotonVelocityMatrix[2] *
                                    fXAdjustment));
      dTotalPhotonPath = (dTotalPhotonPath -
                          sqrtl((fPhotonVelocityMatrix[0]*
                          fPhotonVelocityMatrix[0]*
                          fXAdjustment* fXAdjustment)
                          +(fPhotonVelocityMatrix[1]
                          *fPhotonVelocityMatrix[1]
                          *fXAdjustment*fXAdjustment)+
                          (fPhotonVelocityMatrix[2]
                          *fPhotonVelocityMatrix[2]
                          *fXAdjustment*fXAdjustment)));
}
if((fXAdjustment < fYAdjustment) & (fYAdjustment >=
   fZAdjustment))
{
      fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] –
                                    (fPhotonVelocityMatrix[0] *
                                    fYAdjustment));
      fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] –
                                    (fPhotonVelocityMatrix[1] *
                                    fYAdjustment));
      fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] –
                                    (fPhotonVelocityMatrix[2] *
                                    fYAdjustment));
      dTotalPhotonPath = (dTotalPhotonPath –
                          sqrtl((fPhotonVelocityMatrix[0]*
                                fPhotonVelocityMatrix[0]
                                *fYAdjustment*fYAdjustment)
                                +(fPhotonVelocityMatrix[1]
                                *fPhotonVelocityMatrix[1]
                                *fYAdjustment*fYAdjustment)+
                                (fPhotonVelocityMatrix[2]
                                *fPhotonVelocityMatrix[2]
                                *fYAdjustment*fYAdjustment)));
}
if((fXAdjustment < fZAdjustment) & (fYAdjustment <
   fZAdjustment))
{
      fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                                    (fPhotonVelocityMatrix[0] *
                                    fZAdjustment));
      fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                                    (fPhotonVelocityMatrix[1] *
                                    fZAdjustment));
      fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                                    (fPhotonVelocityMatrix[2] *
                                    fZAdjustment));
      dTotalPhotonPath = (dTotalPhotonPath -
                           sqrtl((fPhotonVelocityMatrix[0]*
                           fPhotonVelocityMatrix[0]*
                           fZAdjustment*fZAdjustment)
```

```
                        +(fPhotonVelocityMatrix[1]
                        *fPhotonVelocityMatrix[1]
                        *fZAdjustment*fZAdjustment)+
                        (fPhotonVelocityMatrix[2]
                        *fPhotonVelocityMatrix[2]
                        *fZAdjustment*fZAdjustment)));
}

//Beginning of numerical approximation.
int nBreak2 = 0;

do
{
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nCounter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nCounter][1])))
      <fRadiusOfFiber)
{
      nContactFiber = 1;
      nFiberSelecter = nCounter;
      fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                                  (fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                                  (fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                                  (fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath –
                        sqrtl((fPhotonVelocityMatrix[0]*
                        fPhotonVelocityMatrix[0])
                        +(fPhotonVelocityMatrix[1]
                        *fPhotonVelocityMatrix[1])+
                        (fPhotonVelocityMatrix[2]
                        *fPhotonVelocityMatrix[2])));
do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]) *
            (fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]))+
            ((fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])*
            (fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])))
            <fRadiusOfFiber)
      {
            nBreak2 = 1;
      }
      else
      {
            fPhotonPositionMatrix[0] =
            (fPhotonPositionMatrix[0] +
            (.1*fPhotonVelocityMatrix[0]));
```

```
            fPhotonPositionMatrix[1] =
            (fPhotonPositionMatrix[1] +
            (.1*fPhotonVelocityMatrix[1]));
            fPhotonPositionMatrix[2] =
            (fPhotonPositionMatrix[2] +
            (.1*fPhotonVelocityMatrix[2]));

            dTotalPhotonPath = (dTotalPhotonPath +
            sqrtl((.1*fPhotonVelocityMatrix[0]*
            .1*fPhotonVelocityMatrix[0])
            +(.1*fPhotonVelocityMatrix[1]
            *.1*fPhotonVelocityMatrix[1])+
            (.1*fPhotonVelocityMatrix[2]
            *.1*fPhotonVelocityMatrix[2])));
      }
}
while(nBreak2 != 1);

fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                           (.1*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                           (.1*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                           (.1*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
                    sqrtl((.1*fPhotonVelocityMatrix[0]*
                           .1*fPhotonVelocityMatrix[0])
                           +(.1*fPhotonVelocityMatrix[1]
                           *.1*fPhotonVelocityMatrix[1])+
                           (.1*fPhotonVelocityMatrix[2]
                           *.1*fPhotonVelocityMatrix[2])));
int nBreak3 = 0;

do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]) *
            (fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]))+
            ((fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])*
            (fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])))
            <fRadiusOfFiber)
      {
            nBreak3 = 1;
      }
      else
      {
            fPhotonPositionMatrix[0] =
            (fPhotonPositionMatrix[0] +
            (.01*fPhotonVelocityMatrix[0]));
            fPhotonPositionMatrix[1] =
            (fPhotonPositionMatrix[1] +
            (.01*fPhotonVelocityMatrix[1]));
            fPhotonPositionMatrix[2] =
            (fPhotonPositionMatrix[2] +
```

```
            (.01*fPhotonVelocityMatrix[2]));
            dTotalPhotonPath = (dTotalPhotonPath +
            sqrtl((.01*fPhotonVelocityMatrix[0]*
            .01*fPhotonVelocityMatrix[0])
            +(.01*fPhotonVelocityMatrix[1]
            *.01*fPhotonVelocityMatrix[1])+
            (.01*fPhotonVelocityMatrix[2]
            *.01*fPhotonVelocityMatrix[2])));
      }
}
while(nBreak3 != 1);

fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                           (.01*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                           (.01*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                           (.01*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
                   sqrtl((.01*fPhotonVelocityMatrix[0]*
                   .01*fPhotonVelocityMatrix[0])
                   +(.01*fPhotonVelocityMatrix[1]
                   *.01*fPhotonVelocityMatrix[1])+
                   (.01*fPhotonVelocityMatrix[2]
                   *.01*fPhotonVelocityMatrix[2])));
int nBreak4 = 0;

do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]) *
            (fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]))+
            ((fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])*
            (fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])))
            <fRadiusOfFiber)
      {
            nBreak4 = 1;
      }
      else
      {
            fPhotonPositionMatrix[0] =
            (fPhotonPositionMatrix[0] +
            (.001*fPhotonVelocityMatrix[0]));
            fPhotonPositionMatrix[1] =
            (fPhotonPositionMatrix[1] +
            (.001*fPhotonVelocityMatrix[1]));
            fPhotonPositionMatrix[2] =
            (fPhotonPositionMatrix[2] +
            (.001*fPhotonVelocityMatrix[2]));
            dTotalPhotonPath = (dTotalPhotonPath +
            sqrtl((.001*fPhotonVelocityMatrix[0]*
            .001*fPhotonVelocityMatrix[0])
```

```
            +(.001*fPhotonVelocityMatrix[1]
            *.001*fPhotonVelocityMatrix[1])+
            (.001*fPhotonVelocityMatrix[2]
            *.001*fPhotonVelocityMatrix[2])));
      }
}
while(nBreak4 != 1);

fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] –
                           (.001*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                           (.001*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                           (.001*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
                   sqrtl((.001*fPhotonVelocityMatrix[0]*
                   .001*fPhotonVelocityMatrix[0])
                   +(.001*fPhotonVelocityMatrix[1]
                   *.001*fPhotonVelocityMatrix[1])+
                   (.001*fPhotonVelocityMatrix[2]
                   *.001*fPhotonVelocityMatrix[2])));
int nBreak5 = 0;

do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]) *
            (fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nCounter][0]))+
            ((fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])*
            (fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nCounter][1])))
            <fRadiusOfFiber)
      {
            nBreak5 = 1;
      }
      else
      {
            fPhotonPositionMatrix[0] =
            (fPhotonPositionMatrix[0] +
            (.0001*fPhotonVelocityMatrix[0]));
            fPhotonPositionMatrix[1] =
            (fPhotonPositionMatrix[1] +
            (.0001*fPhotonVelocityMatrix[1]));
            fPhotonPositionMatrix[2] =
            (fPhotonPositionMatrix[2] +
            (.0001*fPhotonVelocityMatrix[2]));
            dTotalPhotonPath = (dTotalPhotonPath +
            sqrtl((.0001*fPhotonVelocityMatrix[0]*
            .0001*fPhotonVelocityMatrix[0])
            +(.0001*fPhotonVelocityMatrix[1]
            *.0001*fPhotonVelocityMatrix[1])+
            (.0001*fPhotonVelocityMatrix[2] \
            *.0001*fPhotonVelocityMatrix[2])));
      }
}
```

```
                  while(nBreak5 != 1);

                  fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                                              (.0001*fPhotonVelocityMatrix[0]));
                  fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                                              (.0001*fPhotonVelocityMatrix[1]));
                  fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                                              (.0001*fPhotonVelocityMatrix[2]));
                  dTotalPhotonPath = (dTotalPhotonPath –
                                       sqrtl((.0001*fPhotonVelocityMatrix[0]*
                                      .0001*fPhotonVelocityMatrix[0])
                                      +(.0001*fPhotonVelocityMatrix[1]
                                      *.0001*fPhotonVelocityMatrix[1])+
                                      (.0001*fPhotonVelocityMatrix[2]
                                      *.0001*fPhotonVelocityMatrix[2])));
                  //End of numerical approximation.
                  }
                  nCounter++;
                  }
                  while(nCounter != nNumberOfFibers);
                  nCounter = 0;
            }
      }
      while(nEndOfModule != 1);
      if(nContactFiber == 1)
      {
            return 1;
      }
      return 0;
}

/*void main()
{
      int nScintLength = 200;
      int nScintWidth = 32;
      int nScintHeight = 8;
      float fRadiusOfFiber = .8;
      float fFiberMidpointPosition[10][2]=
                                   {{8,7.2},{16,7.2},{24,7.2},{0,0},{0,0},{0,0},
                                    {0,0},{0,0},{0,0},{0,0}};

      double fPhotonPositionMatrix[3]= {20,8.80001,7.2};
      double fPhotonVelocityMatrix[3]= {.1,.001,-.02};
      double dTotalPhotonPath = 0;
      int nWallImpact = 0;
      int nFiberSelecter = 0;
      int nNumberOfFibers = 3;
      int nOutput = 0;
      nOutput = nPhotonReflection(nScintLength, nScintWidth, nScintHeight,
                                  fRadiusOfFiber, fFiberMidpointPosition,
                                  fPhotonPositionMatrix,
                                  fPhotonVelocityMatrix, dTotalPhotonPath,
                                  nWallImpact, nFiberSelecter, nNumberOfFibers);


} */
```

```
//This module works.

//This module is responsible for calculating the reflected velocity vector from
//a scintillator wall.  This will also calculate whether the photon was
//transmitted or reflected.


#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>

int nWallReflect(double fPhotonVelocityMatrix[3],
                                int nWallImpact, int nPhotonWavelength)
{
      float theta = 0;
      float fX = 0;
      float fY = 0;
      float fY3 = 0;
      int nAbsorb = 0;
      int nBreak = 0;
      float fScale = 0;
      fX = nPhotonWavelength;
      fY3 = (random(101));
      fY = (96.3048440998 - 60360087131.5*exp(-.0571259351031*fX));
      fScale = sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
                        (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
                        (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]));
      float fNormalization = 0;
      if(fY < fY3)
      {
            nAbsorb = 1;
      }
      if(nAbsorb == 0)
      {
            if(nWallImpact == 1)
            {
                  do
                  {
                        do
                        {
                              fPhotonVelocityMatrix[0]= .001*random(401);
                              fPhotonVelocityMatrix[1]= .001*(random(801)-400);
                              fPhotonVelocityMatrix[2]= .001*(random(801)-400);
                        }

      while(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
                  (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
                  (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))<=.001);
                  fNormalization =
                  (fScale/(sqrt((fPhotonVelocityMatrix[0]
                  *fPhotonVelocityMatrix[0])+
                  (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
                  (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
```

```
theta =(57.29577951*acos((fPhotonVelocityMatrix[2])/
       sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
       (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
       (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
       fPhotonVelocityMatrix[0] = (fNormalization *
       fPhotonVelocityMatrix[0]);
       fPhotonVelocityMatrix[1] = (fNormalization *
       fPhotonVelocityMatrix[1]);
       fPhotonVelocityMatrix[2] = (fNormalization *
       fPhotonVelocityMatrix[2]);
                  if((theta <= 160) & (theta >= 20))
                  {
                        nBreak = 1;
                  }
            }
            while(nBreak != 1);
       }
       if(nWallImpact == 2)
       {
            do
            {
                  do
                  {
                        fPhotonVelocityMatrix[0]= -.001*random(401);
                        fPhotonVelocityMatrix[1]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[2]= .001*(random(801)-400);
                  }

while(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
      (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
      (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))<=.001);
      fNormalization =
      (fScale/(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
      (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
      (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
      theta =(57.29577951*acos((fPhotonVelocityMatrix[2])/
             sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
             (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
             (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
             fPhotonVelocityMatrix[0] = (fNormalization *
             fPhotonVelocityMatrix[0]);
             fPhotonVelocityMatrix[1] = (fNormalization *
             fPhotonVelocityMatrix[1]);
             fPhotonVelocityMatrix[2] = (fNormalization *
             fPhotonVelocityMatrix[2]);
                  if((theta <= 160) & (theta >= 20))
                  {
                        nBreak = 1;
                  }
            }
            while(nBreak != 1);
       }
       if(nWallImpact == 3)
       {
            do
            {
                  do
```

```
                  {
                        fPhotonVelocityMatrix[0]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[1]= .001*(random(401));
                        fPhotonVelocityMatrix[2]= .001*(random(801)-400);
                  }

while(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
     (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
     (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))<=.001);
     fNormalization =
     (fScale/(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
     (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
     (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
     theta =(57.29577951*acos((fPhotonVelocityMatrix[2])/
            sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
            (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
            (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
            fPhotonVelocityMatrix[0] = (fNormalization *
            fPhotonVelocityMatrix[0]);
            fPhotonVelocityMatrix[1] = (fNormalization *
            fPhotonVelocityMatrix[1]);
            fPhotonVelocityMatrix[2] = (fNormalization *
            fPhotonVelocityMatrix[2]);
                  if((theta <= 160) & (theta >= 20))
                  {
                        nBreak = 1;
                  }

            }
            while(nBreak != 1);
       }
       if(nWallImpact == 4)
       {
            do
            {
                  do
                  {
                        fPhotonVelocityMatrix[0]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[1]= -.001*random(401);
                        fPhotonVelocityMatrix[2]= .001*(random(801)-400);
                  }

while(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
      (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
      (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))<=.001);
      fNormalization =
      (fScale/(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
      (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
      (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
      theta =(57.29577951*acos((fPhotonVelocityMatrix[2])/
             sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
             (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
             (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
             fPhotonVelocityMatrix[0] = (fNormalization *
             fPhotonVelocityMatrix[0]);
             fPhotonVelocityMatrix[1] = (fNormalization *
             fPhotonVelocityMatrix[1]);
```

```
             fPhotonVelocityMatrix[2] = (fNormalization *
             fPhotonVelocityMatrix[2]);
                  if((theta <= 160) & (theta >= 20))
                  {
                        nBreak = 1;
                  }
             }
             while(nBreak != 1);
        }
        if(nWallImpact == 5)
        {
             do
             {
                  do
                  {
                        fPhotonVelocityMatrix[0]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[1]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[2]= .001*random(401);
                  }

while(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
      (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
      (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))<=.001);
      fNormalization =
      (fScale/(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
      (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
      (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
      theta =(57.29577951*acos((fPhotonVelocityMatrix[2])/
             sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
             (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
             (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
             fPhotonVelocityMatrix[0] = (fNormalization *
             fPhotonVelocityMatrix[0]);
             fPhotonVelocityMatrix[1] = (fNormalization *
             fPhotonVelocityMatrix[1]);
             fPhotonVelocityMatrix[2] = (fNormalization *
             fPhotonVelocityMatrix[2]);
                  if((theta <= 160) & (theta >= 20))
                  {
                        nBreak = 1;
                  }
             }
             while(nBreak != 1);
        }
        if(nWallImpact == 6)
        {
             do
             {
                  do
                  {
                        fPhotonVelocityMatrix[0]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[1]= .001*(random(801)-400);
                        fPhotonVelocityMatrix[2]= -.001*random(401);
                  }

while(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
      (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
```

```
            (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))<=.001);
            fNormalization =
            (fScale/(sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
            (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
            (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
            theta =(57.29577951*acos((fPhotonVelocityMatrix[2])/
                   sqrt((fPhotonVelocityMatrix[0]*fPhotonVelocityMatrix[0])+
                   (fPhotonVelocityMatrix[1]*fPhotonVelocityMatrix[1])+
                   (fPhotonVelocityMatrix[2]*fPhotonVelocityMatrix[2]))));
                   fPhotonVelocityMatrix[0] = (fNormalization *
                   fPhotonVelocityMatrix[0]);
                   fPhotonVelocityMatrix[1] = (fNormalization *
                   fPhotonVelocityMatrix[1]);
                   fPhotonVelocityMatrix[2] = (fNormalization *
                   fPhotonVelocityMatrix[2]);
                        if((theta <= 160) & (theta >= 20))
                        {
                              nBreak = 1;
                        }
                  }
                  while(nBreak != 1);
             }
      }
      return nAbsorb;
}
```

```
//This module is working perfectly.

//This module will calculate the angle at which the particle impacts the
//waveshifting fiber and the angle at which it reflects, if it reflects.  This
//module will determine the Cartesian coordinates of the reflected angle and
//output them as the new velocity vector of the photon.  If the photon enters
//the fiber this module will output a value to show that and do nothing else.
//1 if the photon enters the fiber, 0 if it does not.



#include <stdlib.h>

#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>


int AngleOfReflection(double fPhotonPositionMatrix[3],
                      double fPhotonVelocityMatrix[3],
                      float fFiberMidpointPosition[10][2],
                      int nFiberSelecter)
{
      double fXOne = 0.;
      double fYOne = 0.;
      double fXTwo = 0.;
      double fYTwo = 0.;
      double fX = 0.;
      double fY = 0.;
      float theta = 0;
      fX = (fPhotonPositionMatrix[1]-
           fFiberMidpointPosition[nFiberSelecter][0]);
      fY = (fPhotonPositionMatrix[2]-
           fFiberMidpointPosition[nFiberSelecter][1]);
      fXOne = ((fPhotonPositionMatrix[1] –
              fFiberMidpointPosition[nFiberSelecter][0])
              - fPhotonVelocityMatrix[1]);
      fYOne = ((fPhotonPositionMatrix[2]-
              fFiberMidpointPosition[nFiberSelecter][1])
              - fPhotonVelocityMatrix[2]);
      float fPhotonInitialVector[3] = {(-1*fPhotonVelocityMatrix[0]),(-
                                       1*fPhotonVelocityMatrix[1]),
                                       (-1*fPhotonVelocityMatrix[2])};
      float fRadiusVector[3] = {0, fPhotonPositionMatrix[1] -
                                fFiberMidpointPosition[nFiberSelecter][0],
                                fPhotonPositionMatrix[2] -
                                fFiberMidpointPosition[nFiberSelecter][1]};
      theta =(57.29577951*acos(((fRadiusVector[1]*fPhotonInitialVector[1])+
                        (fRadiusVector[2]*fPhotonInitialVector[2]))/
                        (sqrtl((fRadiusVector[1]*fRadiusVector[1])+
                        (fRadiusVector[2]*fRadiusVector[2]))*
                        sqrtl((fPhotonInitialVector[0]*fPhotonInitialVector[0])+
                        (fPhotonInitialVector[1]*fPhotonInitialVector[1])+
                        (fPhotonInitialVector[2]*fPhotonInitialVector[2])))));
```

```
if(theta>=70.57)
{
      if(abs((((fY*fY)*(((fX*fXOne)+(fY*fYOne))*
        ((fX*fXOne)+(fY*fYOne))))-
        ((((fX*fX)+(fY*fY)))*(-(fYOne*fYOne*fX*fX)+
        (fYOne*fYOne*fY*fY)+(2*fXOne*fX*fYOne*fY))))) < .000001)
      {
            fYTwo = (fY*((fXOne*fX)+(fYOne*fY))/((fX*fX)+(fY*fY)));
            if(fabs(fX) < .00001)
            {
                  fXTwo = sqrt((fXOne*fXOne)+(fYOne*fYOne)-(fYTwo*fYTwo));
                  if((fYOne == fYTwo) & (fXOne == fXTwo))
                  {
                        fXTwo = (-1*sqrt((fXOne*fXOne)+(fYOne*fYOne)-
                                (fYTwo*fYTwo)));
                  }
            }
            else
            {
                  fXTwo = (((fXOne*fX)+(fYOne*fY)-(fYTwo*fY))/fX);
            }
      }
      else
      {
            fYTwo = (((fY*((fXOne*fX)+(fYOne*fY)))-
                    sqrt(((fY*fY)*(((fX*fXOne)+(fY*fYOne))*
                    ((fX*fXOne)+(fY*fYOne))))-
                    (((fX*fX)+(fY*fY))*(-1*(fYOne*fYOne*fX*fX)+

(fYOne*fYOne*fY*fY)+(2*fXOne*fX*fYOne*fY)))))/((fX*fX)+(fY*fY)));
            fXTwo = (((fXOne*fX)+(fYOne*fY)-(fYTwo*fY))/fX);
            if((fYOne == fYTwo) & (fXOne == fXTwo))
            {
                  fYTwo = (((fY*((fXOne*fX)+(fYOne*fY)))+
                          sqrt(((fY*fY)*(((fX*fXOne)+(fY*fYOne))*
                         ((fX*fXOne)+(fY*fYOne))))-
                         (((fX*fX)+(fY*fY))*(-1*(fYOne*fYOne*fX*fX)+
                         (fYOne*fYOne*fY*fY)+(2*fXOne*fX*fYOne*fY)))))
                         /((fX*fX)+(fY*fY)));
                  fXTwo = (((fXOne*fX)+(fYOne*fY)-(fYTwo*fY))/fX);
            }
      }
      fPhotonVelocityMatrix[1] = (fXTwo-fX);
      fPhotonVelocityMatrix[2] = (fYTwo-fY);
      fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
                                  (.001 * fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
                                  (.001 * fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                                  (.001 * fPhotonVelocityMatrix[2]));
      return 0;
}
else
{
      fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
                                  (.0001 * fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
```

```
                                        (.0001 * fPhotonVelocityMatrix[1]));
            fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                                        (.0001 * fPhotonVelocityMatrix[2]));
            return 1;
      }
}
/*
void main()
{
int nOrder = 0;
double fPhotonPositionMatrix[3] = {91.398, 8.27143, 7.95256};
double fPhotonVelocityMatrix[3] = {-.126027, -.136315, -.372936};
float fFiberMidpointPosition[10][2]= {{8,7.2},{16,7.2},{24,7.2},
                                      {0,0},{0,0},{0,0},
                                      {0,0},{0,0},{0,0},{0,0}};
int nFiberSelecter = 0;
nOrder = AngleOfReflection(fPhotonPositionMatrix,
                           fPhotonVelocityMatrix,
                           fFiberMidpointPosition,
                           nFiberSelecter);
cout<<fPhotonPositionMatrix[0]<<" "<<fPhotonPositionMatrix[1]<<"
    "<<fPhotonPositionMatrix[2]<<"\n";
cout<<fPhotonVelocityMatrix[0]<<" "<<fPhotonVelocityMatrix[1]<<"
    "<<fPhotonVelocityMatrix[2]<<"\n";
cout<<(sqrt(((fPhotonPositionMatrix[1]-
       fFiberMidpointPosition[0][0]) *
       (fPhotonPositionMatrix[1]- fFiberMidpointPosition[0][0]))+
       ((fPhotonPositionMatrix[2]- fFiberMidpointPosition[0][1])*
       (fPhotonPositionMatrix[2]- fFiberMidpointPosition[0][1]))))<<"\n";
cout<<nOrder<<"\n";
getch();


} */
```

```
//This module works perfectly.


//This module is responsible for calculating if a photon is absorbed into a
//wave-shifting fiber.  If not absorbed the photon will continue to bounce
//inside the scintillator.  If absorbed it will be reemitted with a different
//wavelength.

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>

int nInsideFiber(double fPhotonPositionMatrix[3],
                 double fPhotonVelocityMatrix[3],
                 float fRadiusOfFiber, int nPhotonWavelength,
                 float fFiberMidpointPosition[10][2],
                 double& dTotalPhotonPath, double dXAbsorbZero, double
                 dXAbsorbOne,
                 double dXAbsorbTwo, double dXAbsorbThree, double dXAbsorbFour,
                 double dXAbsorbFive, double dXAbsorbSix, double dXAbsorbSeven,
                 double dXAbsorbEight, double dXAbsorbNine, double dXAbsorbTen,
                 int nFiberSelecter)
{
      int nBreak = 0;
      int nAbsorb = 0;
      float fX = 0;
      float fX2 = 0;
      long double ldProbability = 0;
      do
      {
                  if(sqrtl(((fPhotonPositionMatrix[1]-
                        fFiberMidpointPosition[nFiberSelecter][0]) *
                        (fPhotonPositionMatrix[1]-
                        fFiberMidpointPosition[nFiberSelecter][0]))+
                        ((fPhotonPositionMatrix[2]-
                        fFiberMidpointPosition[nFiberSelecter][1])*
                        (fPhotonPositionMatrix[2]-
                        fFiberMidpointPosition[nFiberSelecter][1])))
                        <=fRadiusOfFiber)
                  {
                        fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
                                                   (.1*fPhotonVelocityMatrix[0]));
                        fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
                                                   (.1*fPhotonVelocityMatrix[1]));
                        fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                                                   (.1*fPhotonVelocityMatrix[2]));
                        dTotalPhotonPath = (dTotalPhotonPath +
                                            sqrtl((fPhotonVelocityMatrix[0]*
                                           .1*.1*fPhotonVelocityMatrix[0])
                                           +(fPhotonVelocityMatrix[1]
                                           *.1*.1*fPhotonVelocityMatrix[1])+
                                           (.1*.1*fPhotonVelocityMatrix[2]
                                           *fPhotonVelocityMatrix[2])));
```

```
            float fPhotonPath = sqrtl((fPhotonVelocityMatrix[0]*
                                .1*.1*fPhotonVelocityMatrix[0])
                                +(fPhotonVelocityMatrix[1]
                                *.1*.1*fPhotonVelocityMatrix[1])+
                                (.1*.1*fPhotonVelocityMatrix[2]*
                                fPhotonVelocityMatrix[2]));
            fX = nPhotonWavelength;
            if(sqrtl(((fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nFiberSelecter][0]) *
            (fPhotonPositionMatrix[1]-
            fFiberMidpointPosition[nFiberSelecter][0]))+
            ((fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nFiberSelecter][1])*
            (fPhotonPositionMatrix[2]-
            fFiberMidpointPosition[nFiberSelecter][1])))
            <fRadiusOfFiber)
            {
                  fX2 = (.01 * random(10001));
                  ldProbability = ((100-
                                    1.30838897329+(98.6916110267*
                                    exp(-1.76944342127*fPhotonPath))))*
                                    ((dXAbsorbZero + (dXAbsorbOne *
                                    fX)+(dXAbsorbTwo*fX*fX)+
                                    (dXAbsorbThree*fX*fX*fX)+
                                    (dXAbsorbFour*fX*fX*fX*fX)+
                                    (dXAbsorbFive*fX*fX*fX*fX*fX)+
                                    (dXAbsorbSix*fX*fX*fX*fX*fX*fX)+
                                   (dXAbsorbSeven*fX*fX*fX*fX*fX*fX*fX)+
                                 (dXAbsorbEight*fX*fX*fX*fX*fX*fX*fX*fX)+
                               (dXAbsorbNine*fX*fX*fX*fX*fX*fX*fX*fX*fX)+
                             (dXAbsorbTen*fX*fX*fX*fX*fX*fX*fX*fX*fX*fX))/
                                      (dXAbsorbZero + (dXAbsorbOne *
                                      408)+(dXAbsorbTwo*408*408)

                                      (dXAbsorbThree*408*408*408)+
                                      (dXAbsorbFour*408*408*408*408)+
                                     (dXAbsorbFive*408*408*408*408*408)+
                                   (dXAbsorbSix*408*408*408*408*408*408)+
                                 (dXAbsorbSeven*408*408*408*408*408*408*408)+
                               (dXAbsorbEight*408*408*408*408*408*408*408*408)+
                             (dXAbsorbNine*408*408*408*408*408*408*408*408*408)+
                  dXAbsorbTen*408*408*408*408*408*408*408*408*408*408))));

                  if (fX2 <= ldProbability)
                  {
                        nBreak = 1;
                        nAbsorb = 1;
                  }
            }
      }
      else
      {
            nBreak = 1;
            fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                                       (.1*fPhotonVelocityMatrix[0]));
            fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                                       (.1*fPhotonVelocityMatrix[1]));
```

```
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                           (.1*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
                   sqrtl((fPhotonVelocityMatrix[0]*
                   .1*.1*fPhotonVelocityMatrix[0])+
                   (fPhotonVelocityMatrix[1]*
                   .1*.1*fPhotonVelocityMatrix[1])+
                   (.1*.1*fPhotonVelocityMatrix[2]
                   *fPhotonVelocityMatrix[2])));
int nBreak2 = 0;
do
{
//Numerical approximation is necessary.
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelecter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelecter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])))
      <fRadiusOfFiber)
{

      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] +
      (.01*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] +
      (.01*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] +
      (.01*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath +
      sqrtl((.01*.01*fPhotonVelocityMatrix[0]*
      fPhotonVelocityMatrix[0])
      +(fPhotonVelocityMatrix[1]
      *.01*.01*fPhotonVelocityMatrix[1])+
      (.01*.01*fPhotonVelocityMatrix[2]
      *fPhotonVelocityMatrix[2])));
}
else
{
      nBreak2 = 1;
}
}
while(nBreak2 != 1);
fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                          (.01*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                          (.01*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                          (.01*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
                sqrtl((.01*.01*fPhotonVelocityMatrix[0]*
                fPhotonVelocityMatrix[0])
                +(fPhotonVelocityMatrix[1]
```

```
                *.01*.01*fPhotonVelocityMatrix[1])+
               (.01*.01*fPhotonVelocityMatrix[2]
               *fPhotonVelocityMatrix[2])));
int nBreak3 = 0;
do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelecter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelecter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])))
      <fRadiusOfFiber)
{
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] +
      (.001*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] +
      (.001*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] +
      (.001*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath +
           sqrtl((.001*.001*fPhotonVelocityMatrix[0]*
             fPhotonVelocityMatrix[0])
             +(fPhotonVelocityMatrix[1]
             *.001*.001*fPhotonVelocityMatrix[1])+
             (.001*.001*fPhotonVelocityMatrix[2]
             *fPhotonVelocityMatrix[2])));
}
else
{
      nBreak3 = 1;
}
}
while(nBreak3 != 1);
fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                  (.001*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                  (.001*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                  (.001*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
              sqrtl((.001*.001*fPhotonVelocityMatrix[0]*

              fPhotonVelocityMatrix[0])
              +(fPhotonVelocityMatrix[1]
              *.001*.001*fPhotonVelocityMatrix[1])+
              (.001*.001*fPhotonVelocityMatrix[2]
              *fPhotonVelocityMatrix[2])));
int nBreak4 = 0;
do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
```

```
      fFiberMidpointPosition[nFiberSelecter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelecter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])))
      <fRadiusOfFiber)
{
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] +
      (.0001*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] +
      (.0001*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] +
      (.0001*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath +
            sqrtl((.0001*.0001*fPhotonVelocityMatrix[0]*
             fPhotonVelocityMatrix[0])+
             (fPhotonVelocityMatrix[1]
             *.0001*.0001*fPhotonVelocityMatrix[1])+
             (.0001*.0001*fPhotonVelocityMatrix[2]
             *fPhotonVelocityMatrix[2])));
}
else
{
      nBreak4 = 1;
}
}
while(nBreak4 != 1);
fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] -
                  (.0001*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] -
                  (.0001*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] -
                  (.0001*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath –
            sqrtl((.0001*.0001*fPhotonVelocityMatrix[0]*
             fPhotonVelocityMatrix[0])
             +(fPhotonVelocityMatrix[1]
             *.0001*.0001*fPhotonVelocityMatrix[1])+
             (.0001*.0001*fPhotonVelocityMatrix[2]
             *fPhotonVelocityMatrix[2])));
int nBreak5 = 0;
do
{
      if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelecter][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelecter][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelecter][1])))
      <fRadiusOfFiber)
```

```
                        {
                              fPhotonPositionMatrix[0] =
                              (fPhotonPositionMatrix[0] +
                              (.00001*fPhotonVelocityMatrix[0]));
                              fPhotonPositionMatrix[1] =
                              (fPhotonPositionMatrix[1] +

                              (.00001*fPhotonVelocityMatrix[1]));
                              fPhotonPositionMatrix[2] =
                              (fPhotonPositionMatrix[2] +
                              (.00001*fPhotonVelocityMatrix[2]));
                              dTotalPhotonPath = (dTotalPhotonPath +
                              sqrtl((.00001*.00001*fPhotonVelocityMatrix[0]*
                              fPhotonVelocityMatrix[0])
                              +(fPhotonVelocityMatrix[1]
                              *.00001*.00001*fPhotonVelocityMatrix[1])+
                              (.00001*.00001*fPhotonVelocityMatrix[2]
                              *fPhotonVelocityMatrix[2])));
                        }
                        else
                        {
                              nBreak5 = 1;
                        }
                        }
                        while(nBreak5 != 1);
                  }
      }
      while(nBreak != 1);
      if(nAbsorb == 1)
      {
            return 1;
      }
      else
      {
            return 0;
      }
}

/*void main()
{
randomize();
int nNumberOfFibers = 1;
double fPhotonPositionMatrix[3] = {384.063,.372122,.918184};
double fPhotonVelocityMatrix[3] = {.232467,-.430748,-.0478609};
float fRadiusOfFiber = 1;
int nPhotonWavelength = 300;
float fFiberMidpointPosition[10][2]={{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},
                                    {0,0},{0,0},{0,0}};
double dTotalPhotonPath = 0;
double dXAbsorbZero = 0;
double dXAbsorbOne = 0;
double dXAbsorbTwo = 0;
double dXAbsorbThree = 0;
double dXAbsorbFour = 0;
double dXAbsorbFive = 0;
double dXAbsorbSix = 0;
double dXAbsorbSeven = 0;
```

```
double dXAbsorbEight = 0;
double dXAbsorbNine = 0;
double dXAbsorbTen = 1;
int nAbsorb2 = nInsideFiber(fPhotonPositionMatrix,fPhotonVelocityMatrix,
                                fRadiusOfFiber,nPhotonWavelength,
                                fFiberMidpointPosition,
                                dTotalPhotonPath, dXAbsorbZero, dXAbsorbOne,
                                dXAbsorbTwo, dXAbsorbThree, dXAbsorbFour,
                                dXAbsorbFive, dXAbsorbSix, dXAbsorbSeven,
                                dXAbsorbEight, dXAbsorbNine, dXAbsorbTen,
                                nNumberOfFibers);
cout<<fPhotonPositionMatrix[0]<<" "<<fPhotonPositionMatrix[1]
       <<" "<<fPhotonPositionMatrix[2]<<"\n";
cout<<fPhotonVelocityMatrix[0]<<" "<<fPhotonVelocityMatrix[1]
       <<" "<<fPhotonVelocityMatrix[2]<<"\n";
getch();

}*/
```

```
//This module is working perfectly.

//This module calculates the wavelength of the emitted photon in the fiber.

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>

void FiberEmit(double fPhotonVelocityMatrix[3], double dXEmitZero,
               double dXEmitOne, double dXEmitTwo, double dXEmitThree,
               double dXEmitFour, double dXEmitFive, double dXEmitSix,
               double dXEmitSeven, double dXEmitEight, double dXEmitNine,
               double dXEmitTen, float fEmitBoundaryOne,
               float fEmitBoundaryTwo, int& nPhotonWavelength)
{
      int nBreak2 = 0;
      int x = 0;
      float y = 0.;


      //Assign the photon a wavelength at this point.
      do
      {
            x = random((fEmitBoundaryTwo –
                fEmitBoundaryOne+1))+fEmitBoundaryOne;
            y = .001*random(1000);
            if(((dXEmitZero)+(dXEmitOne*x)+(dXEmitTwo*x*x)+(dXEmitThree*x*x*x)+
                 dXEmitFour*x*x*x*x)+(dXEmitFive*x*x*x*x*x)
                 +(dXEmitSix*x*x*x*x*x*x)+
                 (dXEmitSeven*x*x*x*x*x*x*x)+(dXEmitEight*x*x*x*x*x*x*x*x)+
                 (dXEmitNine*x*x*x*x*x*x*x*x*x)
                 +(dXEmitTen*x*x*x*x*x*x*x*x*x*x))>y)
            {
                  nBreak2 = 1;
            }
      }
      while(nBreak2 != 1);
      nPhotonWavelength = x;
}

/*void main()
{
      float fPhotonVelocityMatrix[3] = {0,0,0};
      double dXEmitZero = .1;
      double dXEmitOne = 0;
      double dXEmitTwo = 0;
      double dXEmitThree = 0;
      double dXEmitFour = 0;
      double dXEmitFive = 0;
      double dXEmitSix = 0;
      double dXEmitSeven = 0;
      double dXEmitEight = 0;
      double dXEmitNine = 0;
```

```
      double dXEmitTen = 0;
      float fEmitBoundaryOne = 1;
      float fEmitBoundaryTwo = 9;
      int nPhotonWavelength = 0;
      FiberEmit(fPhotonVelocityMatrix, dXEmitZero,
                              dXEmitOne, dXEmitTwo, dXEmitThree,
                              dXEmitFour, dXEmitFive, dXEmitSix,
                              dXEmitSeven, dXEmitEight, dXEmitNine,
                              dXEmitTen, fEmitBoundaryOne,
                              fEmitBoundaryTwo, nPhotonWavelength);
      cout<<nPhotonWavelength<<"\n";
      getch();
}*/
```

```
//This works perfectly.

//The purpose of this module is to follow the photon through the wave shifting
//fiber.  This module assumes that one end of the fiber is coated with a black
//paint.  X is the scintillator width at this end.  The module will bring the
//photon to the wall and calculate the angle.  If the angle is enough for total
//internal reflection, because of geometry the photon makes it to the end of the
//fiber.

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>


void FiberPath(double fPhotonVelocityMatrix[3], double fPhotonPositionMatrix[3],
                              float fRadiusOfFiber,
                              float fFiberMidpointPosition[10][2],
                              double& dTotalPhotonPath, int& nDeadPhoton,
                              int nFiberSelector,
                              float fLengthOfOutsideFiber)
{
      float fX = 0;
      int nBreak = 0;
      float fXAdjustment = 0;
      int nWallContact = 0;
      float theta = 0;
      double dAttenuationInFiber = 0.;
      do
      {
            fPhotonVelocityMatrix[0]= .0001*(random(801)-400);
            fPhotonVelocityMatrix[1]= .0001*(random(801)-400);
            fPhotonVelocityMatrix[2]= .0001*(random(801)-400);
      }
      while((fPhotonVelocityMatrix[0] == 0) &
                  (fPhotonVelocityMatrix[1] == 0) &
                  (fPhotonVelocityMatrix[2] == 0));
      fPhotonVelocityMatrix[0] = .1*fPhotonVelocityMatrix[0];
      fPhotonVelocityMatrix[1] = .1*fPhotonVelocityMatrix[1];
      fPhotonVelocityMatrix[2] = .1*fPhotonVelocityMatrix[2];

      if(fPhotonVelocityMatrix[0] > 0)
      {
            nDeadPhoton = 1;
      }
      else
      {
            do
            {
            nWallContact = 0;
            fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
                                        fPhotonVelocityMatrix[0]);
            fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
                                        fPhotonVelocityMatrix[1]);
```

```
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                           fPhotonVelocityMatrix[2]);
dTotalPhotonPath = (dTotalPhotonPath +
                    sqrtl((fPhotonVelocityMatrix[0]*
                    fPhotonVelocityMatrix[0])
                    +(fPhotonVelocityMatrix[1]*
                    fPhotonVelocityMatrix[1])+
                    (fPhotonVelocityMatrix[2]
                    *fPhotonVelocityMatrix[2])));
              int nBreak2 = 0;
              //Numerical approximation is necessary.
              if(sqrtl(((fPhotonPositionMatrix[1]-
                    fFiberMidpointPosition[nFiberSelector][0]) *
                    (fPhotonPositionMatrix[1]-
                    fFiberMidpointPosition[nFiberSelector][0]))+
                    ((fPhotonPositionMatrix[2]-
                    fFiberMidpointPosition[nFiberSelector][1])*
                    (fPhotonPositionMatrix[2]-
                    fFiberMidpointPosition[nFiberSelector][1])))
                    >=fRadiusOfFiber)
              {
              do
              {
              if(sqrtl(((fPhotonPositionMatrix[1]-
                    fFiberMidpointPosition[nFiberSelector][0]) *
                    (fPhotonPositionMatrix[1]-
                    fFiberMidpointPosition[nFiberSelector][0]))+
                    ((fPhotonPositionMatrix[2]-
                    fFiberMidpointPosition[nFiberSelector][1])*
                    (fPhotonPositionMatrix[2]-
                    fFiberMidpointPosition[nFiberSelector][1])))
                    <fRadiusOfFiber)
              {
                    nBreak2 = 1;
              }
              else
              {
                    nWallContact = 1;
                    fPhotonPositionMatrix[0] =
                    (fPhotonPositionMatrix[0] -
                    (.1*fPhotonVelocityMatrix[0]));
                    fPhotonPositionMatrix[1] =
                    (fPhotonPositionMatrix[1] -
                    (.1*fPhotonVelocityMatrix[1]));
                    fPhotonPositionMatrix[2] =
                    (fPhotonPositionMatrix[2] -
                    (.1*fPhotonVelocityMatrix[2]));
                    dTotalPhotonPath = (dTotalPhotonPath –
                    sqrtl((.1*.1*fPhotonVelocityMatrix[0]*
                    PhotonVelocityMatrix[0])+(fPhotonVelocityMatrix[1]
                    *.1*.1*fPhotonVelocityMatrix[1])+
                    (.1*.1*fPhotonVelocityMatrix[2]
                    *fPhotonVelocityMatrix[2])));
              }
              }
              while(nBreak2 != 1);
              fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
```

```
                          (.1*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
                          (.1*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                          (.1*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath +
sqrtl((.1*.1*fPhotonVelocityMatrix[0]*
fPhotonVelocityMatrix[0])+(fPhotonVelocityMatrix[1]
*.1*.1*fPhotonVelocityMatrix[1])+
(.1*.1*fPhotonVelocityMatrix[2]
*fPhotonVelocityMatrix[2])));
int nBreak3 = 0;
do
{
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelector][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelector][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelector][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelector][1])))
      <fRadiusOfFiber)
{
      nBreak3 = 1;
}
else
{
      nWallContact = 1;
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] -
      (.01*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] -
      (.01*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] -
      (.01*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath –
      sqrtl((.01*.01*fPhotonVelocityMatrix[0]*
      PhotonVelocityMatrix[0])+(fPhotonVelocityMatrix[1]
      *.01*.01*fPhotonVelocityMatrix[1])+
      (.01*.01*fPhotonVelocityMatrix[2]
      *fPhotonVelocityMatrix[2])));
}
}
while(nBreak3 != 1);
fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
                          (.01*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
                          (.01*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                          (.01*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath +
sqrtl((.01*.01*fPhotonVelocityMatrix[0]*
fPhotonVelocityMatrix[0])+(fPhotonVelocityMatrix[1]
*.01*.01*fPhotonVelocityMatrix[1])+
```

```
(.01*.01*fPhotonVelocityMatrix[2]
*fPhotonVelocityMatrix[2])));
int nBreak4 = 0;
do
{
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelector][0]) *
      (fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelector][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelector][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelector][1])))
      <fRadiusOfFiber)
{
      nBreak4 = 1;
}
else
{
      nWallContact = 1;
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] -
      (.001*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] -
      (.001*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] -
      (.001*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath –
      sqrtl((.001*.001*fPhotonVelocityMatrix[0]*
      fPhotonVelocityMatrix[0])
      +(fPhotonVelocityMatrix[1]
      *.001*.001*fPhotonVelocityMatrix[1])+
      (.001*.001*fPhotonVelocityMatrix[2]
      *fPhotonVelocityMatrix[2])));
}
}
while(nBreak4 != 1);
fPhotonPositionMatrix[0] = (fPhotonPositionMatrix[0] +
                  (.001*fPhotonVelocityMatrix[0]));
fPhotonPositionMatrix[1] = (fPhotonPositionMatrix[1] +
                  (.001*fPhotonVelocityMatrix[1]));
fPhotonPositionMatrix[2] = (fPhotonPositionMatrix[2] +
                  (.001*fPhotonVelocityMatrix[2]));
dTotalPhotonPath = (dTotalPhotonPath +
sqrtl((.001*.001*fPhotonVelocityMatrix[0]*
fPhotonVelocityMatrix[0])+(fPhotonVelocityMatrix[1]
*.001*.001*fPhotonVelocityMatrix[1])+
(.001*.001*fPhotonVelocityMatrix[2]
*fPhotonVelocityMatrix[2])));
int nBreak5 = 0;
do
{
if(sqrtl(((fPhotonPositionMatrix[1]-
      fFiberMidpointPosition[nFiberSelector][0]) *
      (fPhotonPositionMatrix[1]-
```

```
      fFiberMidpointPosition[nFiberSelector][0]))+
      ((fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelector][1])*
      (fPhotonPositionMatrix[2]-
      fFiberMidpointPosition[nFiberSelector][1])))
      <fRadiusOfFiber)
{
      nBreak5 = 1;
}
else
{
      nWallContact = 1;
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] -

      (.0001*fPhotonVelocityMatrix[0]));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] -
      (.0001*fPhotonVelocityMatrix[1]));
      fPhotonPositionMatrix[2] =
      (fPhotonPositionMatrix[2] -
      (.0001*fPhotonVelocityMatrix[2]));
      dTotalPhotonPath = (dTotalPhotonPath –
      sqrtl((.0001*.0001*fPhotonVelocityMatrix[0]*
      fPhotonVelocityMatrix[0])
      +(fPhotonVelocityMatrix[1]
      *.0001*.0001*fPhotonVelocityMatrix[1])+
      (.0001*.0001*fPhotonVelocityMatrix[2]
      *fPhotonVelocityMatrix[2])));
}
}
while(nBreak5 != 1);
}
if(fPhotonPositionMatrix[0] <= (-
   1*fLengthOfOutsideFiber))
{
      nBreak = 1;
      if(fPhotonVelocityMatrix[0] < -.0001)
      {
            fXAdjustment =
            fabs((fPhotonPositionMatrix[0]
            +fLengthOfOutsideFiber)/
            fPhotonVelocityMatrix[0]);
      }
      else
      {
            fXAdjustment = .0001;
      }
      fPhotonPositionMatrix[0] =
      (fPhotonPositionMatrix[0] -
      (fPhotonVelocityMatrix[0] *
      fXAdjustment));
      fPhotonPositionMatrix[1] =
      (fPhotonPositionMatrix[1] -
      (fPhotonVelocityMatrix[1] *
      fXAdjustment));
      fPhotonPositionMatrix[2] =
```

```
      (fPhotonPositionMatrix[2] -
      (fPhotonVelocityMatrix[2] *
      fXAdjustment));
      dTotalPhotonPath = (dTotalPhotonPath -
      sqrtl((fPhotonVelocityMatrix[0]*
      fPhotonVelocityMatrix[0]*
      fXAdjustment*fXAdjustment)
      +(fPhotonVelocityMatrix[1]
      *fPhotonVelocityMatrix[1]
      *fXAdjustment*fXAdjustment)+
      (fPhotonVelocityMatrix[2]
      *fPhotonVelocityMatrix[2]
      *fXAdjustment*fXAdjustment)));
}
//Deal with reflections from the fiber here the same way
//as they are
//dealt with for the outside of the fiber.
if(nWallContact == 1)
{
      float fPhotonInitialVector[3] = {(-
            1*fPhotonVelocityMatrix[0]),
            (-1*fPhotonVelocityMatrix[1]),
            (-1*fPhotonVelocityMatrix[2])};
      float fRadiusVector[3] = {0, (-
            1*(fPhotonPositionMatrix[1] -
            fFiberMidpointPosition[nFiberSelector][0])),
            (-1*(fPhotonPositionMatrix[2] -
            fFiberMidpointPosition[nFiberSelector][1]
            ))};
theta =(57.29577951*acos(((fRadiusVector[1]
       *fPhotonInitialVector[1])+
       (fRadiusVector[2]*fPhotonInitialVector[2]))/
       (sqrtl((fRadiusVector[0]*fRadiusVector[0])+
       (fRadiusVector[1]*fRadiusVector[1])+
       (fRadiusVector[2]*fRadiusVector[2]))*
       sqrtl((fPhotonInitialVector[0]
       *fPhotonInitialVector[0])+
       (fPhotonInitialVector[1]*
       fPhotonInitialVector[1])+
       (fPhotonInitialVector[2]
       *fPhotonInitialVector[2])))));
      //works so far.
      if(theta >=68.6)
      {
            if(fPhotonVelocityMatrix[0] < -.0001)
            {
                  dTotalPhotonPath = (dTotalPhotonPath +
sqrtl((
((((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
/fPhotonVelocityMatrix[0])*
fPhotonVelocityMatrix[0])*
(((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
/fPhotonVelocityMatrix[0])*

fPhotonVelocityMatrix[0]))+
((((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
/fPhotonVelocityMatrix[0])*
```

```
                        fPhotonVelocityMatrix[1])*
                        (((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                        /fPhotonVelocityMatrix[0])*
                        fPhotonVelocityMatrix[1]))+
                        ((((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                        /fPhotonVelocityMatrix[0])*
                        fPhotonVelocityMatrix[2])*
                        (((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                        /fPhotonVelocityMatrix[0])*
                        fPhotonVelocityMatrix[2])))));
                                    }
                                    else
                                    {
                        fPhotonVelocityMatrix[0] = -.0001;
                        dTotalPhotonPath = (dTotalPhotonPath + sqrtl((
                       ((((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                       /fPhotonVelocityMatrix[0])*
                       fPhotonVelocityMatrix[0])*
                       (((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                       /fPhotonVelocityMatrix[0])*
                       fPhotonVelocityMatrix[0]))+
                       ((((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                       /fPhotonVelocityMatrix[0])*
                       fPhotonVelocityMatrix[1])*
                       (((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                       /fPhotonVelocityMatrix[0])*
                       fPhotonVelocityMatrix[1]))+
                       ((((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                       /fPhotonVelocityMatrix[0])*
                       fPhotonVelocityMatrix[2])*
                       (((fPhotonPositionMatrix[0]+fLengthOfOutsideFiber)
                       /fPhotonVelocityMatrix[0])*
                       fPhotonVelocityMatrix[2])))));
                                    }
                                    nBreak = 1;
                                    //Deal with the efficiency curve of the PMT
                                    //after that.
                              }
                              else
                              {
                                    nBreak = 1;
                                    nDeadPhoton = 1;
                              }
                        }
                  }
                  while(nBreak != 1);
                  dAttenuationInFiber = exp((-dTotalPhotonPath/4000));
                  fX = (.0001*random(10001));
                  if(dAttenuationInFiber <= fX)
                  {
                        nBreak = 1;
                        nDeadPhoton = 1;
                  }
            }
}
/*
void main()
```

```
{
	randomize();
	int nCounter = 0;
	int nZeros = 0;
	int nOnes = 0;
	do
	{
	double fPhotonVelocityMatrix[3] = {-3,.3,-.1};
	double fPhotonPositionMatrix[3] = {100,200,20};
	float fRadiusOfFiber = 1.;
	float fFiberMidpointPosition[10][2] = {{200,20}};
	double dTotalPhotonPath = 0;
	int nFiberSelector = 0;
	int nDeadPhoton = 0;
	float fLengthOfOutsideFiber = 10000;
	FiberPath(fPhotonVelocityMatrix, fPhotonPositionMatrix,
	                    fRadiusOfFiber, fFiberMidpointPosition,
	                    dTotalPhotonPath, nDeadPhoton, nFiberSelector,
fLengthOfOutsideFiber);
	cout<<nDeadPhoton<<"\n";
	if(nDeadPhoton == 0)
	{
	      nZeros++;
	}
	else
	{
	nOnes++;
	}
	nCounter++;
	cout<<dTotalPhotonPath<<"\n";
	}
	while(nCounter != 100);
	cout<<nZeros<<" "<<nOnes<<"\n";
	getch();
}*/
```

```
//This module controls the photon's detection by the photomultiplier tube.
//The module uses the efficiency curve of the photomultiplier tube and the
//photon's wavelength to determine whether the photon is detected or not.

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>


int nPhotoMultiplier(int nPhotonWavelength, double dXPMTZero, double dXPMTOne,
                     double dXPMTTwo, double dXPMTThree, double dXPMTFour,
                     double dXPMTFive, double dXPMTSix, double dXPMTSeven,
                     double dXPMTEight, double dXPMTNine, double dXPMTTen)
{
      int fY = random(101);
      float fX = nPhotonWavelength;
      if(((dXPMTZero)+(dXPMTOne*fX)+(dXPMTTwo*fX*fX)+(dXPMTThree*fX*fX*fX)+
      (dXPMTFour*fX*fX*fX*fX)+(dXPMTFive*fX*fX*fX*fX*fX)+(dXPMTSix*fX*fX*fX*fX*
      fX*fX)+
      (dXPMTSeven*fX*fX*fX*fX*fX*fX*fX)+(dXPMTEight*fX*fX*fX*fX*fX*fX*fX*fX)+
      (dXPMTNine*fX*fX*fX*fX*fX*fX*fX*fX*fX)+(dXPMTTen*fX*fX*fX*fX*fX*fX*fX*fX*
      fX*fX))>=fY)
      {
            return 0;
      }
      else
      {
            return 1;
      }
}
```

```
//This is the main program that is responsible for making the simulation of the
//scintillation process.

//Libraries.
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <complex.h>

//Header Files
#include "scinthea.h"



//Modules.
#include "userinpu.cpp"
#include "muonimpa.cpp"
#include "muonpath.cpp"
#include "photoncr.cpp"
#include "photonre.cpp"
#include "advadvwi.cpp"
#include "reflecti.cpp"
#include "fiberpen.cpp"
#include "fiberemi.cpp"
#include "advfpath.cpp"
#include "photomul.cpp"

void main()
{
      randomize();
      int nUserRestart = 0;
      do
      {
      int nDataSelecter = 0;
      int nInitCounter = 0;
      int nCounter3 = 0;
      double dTotalScintPath = 0.;
      int nPEBins[300];

      long nBounceBins[80];
      do
      {
            nBounceBins[nInitCounter] = 0;
            nInitCounter++;
      }
      while(nInitCounter != 80);
      nInitCounter = 0;
      long nScintillatorBins[300];
      do
      {
            nPEBins[nInitCounter] = 0;
            nScintillatorBins[nInitCounter] = 0;
            nInitCounter++;
      }
      while(nInitCounter != 300);
```

```
int nPECounter = 0;
int nPhotonRecievedCount2 = 0;
long nTotalBounces = 0;
int nFiberAbsorb = 0;
int nReflect = 0;
int nDeadPhoton = 0;
int nProgCount = 0;
int nMuonBreak = 0;
int nEndOfSimulation = 0;
int nPhotonCount = 0;
int nPhotonWavelength = 0;
long nNumberOfPhotons = 0;
float fEfficiencyOfScint = .68;
float fLengthOfOutsideFiber = 0.;
float fMuonPathLength = 0.;
int nContactFiber = 0;
double dTotalPhotonPath = 0.;
int nWallImpact = 0;
int nRecievedPhoton = 0;
int nPhotonRecievedCount = 0;
long fSumOfAllPhotons = 0;

//Set this equal to the constant after fiber penetration.
int nFiberSelecter = 0;

long nUserMuonCount = 0;
int nNumberOfFibers = 0;
float fRadiusOfFiber = 1.;

//Scintillator Emission.
double dXZero = (4181935.25922/100);
double dXOne = (38881.4332870/100);
double dXTwo = (102.662676924/100);
double dXThree = (.0609780818744/100);
double dXFour = (-.000577925965284/100);
double dXFive = (2.6550828884*.0000001/100);
double dXSix = (8.75972077777*.0000000001/100);
double dXSeven = (1.39516783915*.000000000001/100);
double dXEight = (-5.876098468*.0000000000000001/100);;
double dXNine = (4.17832612535*.00000000000000000001/100);
double dXTen = 0;
float fBoundaryOne= 382;
float fBoundaryTwo= 488;

//Fiber Absorbtion.
double dXAbsorbZero = (-30282.477957);
double dXAbsorbOne = (447.747345866);
double dXAbsorbTwo = (-2.74193745473);
double dXAbsorbThree = (.00889853330148);
double dXAbsorbFour = (-.0000161360732946);
double dXAbsorbFive = (.0000000154972144093);
double dXAbsorbSix = (-.00000000000615713961691);
double dXAbsorbSeven = 0;
double dXAbsorbEight = 0;
double dXAbsorbNine = 0;
double dXAbsorbTen = 0;
```

```
//Fiber Emission.
double dXEmitZero = (-18393.1185986);
double dXEmitOne = (101.723244355);
double dXEmitTwo = (-.00740757824906);
double dXEmitThree = (-.00109287186075);
double dXEmitFour = (.00000300959365738);
double dXEmitFive = (-.000000003292882630009);
double dXEmitSix = (.0000000000133096500507);
double dXEmitSeven = 0;
double dXEmitEight = 0;
double dXEmitNine = 0;
double dXEmitTen = 0;
float fEmitBoundaryOne = 471;
float fEmitBoundaryTwo = 595;

//PMT Efficiency
double dXPMTZero = (2659.58506425);
double dXPMTOne = (42.9797147972);
double dXPMTTwo = (.270744442978);
double dXPMTThree = (-.0008650906364);
double dXPMTFour = (.00000150149626509);
double dXPMTFive = (-.000000000135589152326);
double dXPMTSix = (.0000000000000500713578771);
double dXPMTSeven = 0;
double dXPMTEight = 0;
double dXPMTNine = 0;
double dXPMTTen = 0;


//Length = x
//Width = y
//Height = z
int nScintLength = 0;
int nScintWidth = 0;
int nScintHeight = 0;


//[0] = x
//[1] = y
//[2] = z
double fMuonPositionMatrix[3] = {0,0,0};
double fMuonFinalPositionMatrix[3] = {0,0,0};
double fPhotonPositionMatrix[3] = {0,0,0};


//[0] = x
//[1] = y
//[2] = z
float fMuonVelocityMatrix[3] = {0,0,0};
double fPhotonVelocityMatrix[3] = {0,0,0};


do
{
      cout<<"Enter the number of wave-shifting fibers in your
            scintillator(1-10).\n";
```

```
            cin >> nNumberOfFibers;
            clrscr();
      }
      while((nNumberOfFibers < 1)|(nNumberOfFibers > 10));

      //[0] = y
      //[1] = z
      float fFiberMidpointPosition[10][2] = {{0,0},{0,0},{0,0},{0,0},{0,0},
                                             {0,0},{0,0},{0,0},{0,0},{0,0}};


      //User Inputs values at this point.  This will only be run once.
      UserInput(nScintLength, nScintWidth, nScintHeight, nNumberOfFibers,
                  fFiberMidpointPosition, fRadiusOfFiber, dXZero, dXOne, dXTwo,
                  dXThree, dXFour, dXFive, dXSix, dXSeven, dXEight, dXNine,
                  dXTen, fBoundaryOne, fBoundaryTwo, fEmitBoundaryOne,
                  fEmitBoundaryTwo,
                  fEfficiencyOfScint, dXAbsorbZero, dXAbsorbOne, dXAbsorbTwo,
                  dXAbsorbThree, dXAbsorbFour, dXAbsorbFive, dXAbsorbSix,
                  dXAbsorbSeven,
                  dXAbsorbEight, dXAbsorbNine, dXAbsorbTen, dXEmitZero,
                  dXEmitOne,
                  dXEmitTwo, dXEmitThree, dXEmitFour, dXEmitFive, dXEmitSix,
                  dXEmitSeven, dXEmitEight, dXEmitNine, dXEmitTen,
                  fLengthOfOutsideFiber, dXPMTZero, dXPMTOne, dXPMTTwo,
                  dXPMTThree,
                  dXPMTFour, dXPMTFive, dXPMTSix, dXPMTSeven, dXPMTEight,
                  dXPMTNine,
                  dXPMTTen, nUserMuonCount);

      //This is the beginning of the actual program.  This will be looped until
      //maximum loops are achieved or the user breaks the loop.
      do
      {
            MuonImpact(nScintLength, nScintWidth, nScintHeight,
                        fMuonPositionMatrix, fMuonVelocityMatrix, nProgCount,
                        nMuonBreak);
                  if(nProgCount == nUserMuonCount)
                  {
                        nEndOfSimulation = 1;
                  }
                  MuonPath(nScintLength, nScintWidth, nScintHeight,
                           fMuonPositionMatrix, fMuonVelocityMatrix,
                           fMuonFinalPositionMatrix, fFiberMidpointPosition,
                           fRadiusOfFiber, fMuonPathLength, nNumberOfPhotons,
                           fEfficiencyOfScint, nNumberOfFibers);
                  fSumOfAllPhotons = (fSumOfAllPhotons +
(float)nNumberOfPhotons);
                  do
                  {
                        dTotalPhotonPath = 0;
                    //  cout<<"0"<<"\n";
                        PhotonCreation(fMuonPositionMatrix,
                                       fMuonFinalPositionMatrix,
                                       fMuonVelocityMatrix, nPhotonCount,
                                       fRadiusOfFiber,
                                       fFiberMidpointPosition,
```

```
                        fPhotonPositionMatrix,
                        fPhotonVelocityMatrix, nPhotonWavelength,
                        dXTen, dXNine, dXEight, dXSeven, dXSix,
                        dXFive, dXFour, dXThree,
                        dXTwo, dXOne, dXZero, nScintHeight,
                        fBoundaryOne, fBoundaryTwo,
                        nNumberOfFibers);
                do
                {
                      //cout<<"1"<<"\n";
                      nContactFiber =
                      nPhotonReflection(nScintLength, nScintWidth,
                                        nScintHeight,
                                        fRadiusOfFiber,
                                        fFiberMidpointPosition,
                                        fPhotonPositionMatrix,
                                        fPhotonVelocityMatrix,
                                        dTotalPhotonPath, nWallImpact,
                                        nFiberSelecter,
                                        nNumberOfFibers);
                      if(nContactFiber == 0)
                      {
                            nDeadPhoton =
                        nWallReflect(fPhotonVelocityMatrix, nWallImpact,
                                     nPhotonWavelength);
                            if(nDeadPhoton == 0)
                            {
                                  nTotalBounces = (nTotalBounces + 1);
                            }
                      }
                      if(nContactFiber == 1)
                      {
                            //cout<<"3"<<"\n";
                            if(dTotalPhotonPath <= 8990)
                            {
                                  dTotalScintPath = dTotalPhotonPath;
                            }
                            nContactFiber = 0;
                            //Error occurs here.
                            nReflect =
                            AngleOfReflection(fPhotonPositionMatrix,
                                              fPhotonVelocityMatrix,
                                              fFiberMidpointPosition,
                                              nFiberSelecter);
                            //cout<<"4"<<"\n";
                            if(nReflect == 1)
                            {
                                  nFiberAbsorb =
                                  nInsideFiber(fPhotonPositionMatrix,
                                               fPhotonVelocityMatrix,
                                               fRadiusOfFiber,
                                               nPhotonWavelength,
                                               fFiberMidpointPosition,
                                               dTotalPhotonPath,
                                               dXAbsorbZero,
                                               dXAbsorbOne, dXAbsorbTwo,
                                               dXAbsorbThree,
```

```
                dXAbsorbFour,
                dXAbsorbFive,
                dXAbsorbSix,
                dXAbsorbSeven,
                dXAbsorbEight,
                dXAbsorbNine,
                dXAbsorbTen,
                nFiberSelecter);
if(((fPhotonPositionMatrix[0] >= -
    .00001) &
  (fPhotonPositionMatrix[0] <=
  (nScintLength+.00001))) &
  ((fPhotonPositionMatrix[1] >= -
  .00001) &
  (fPhotonPositionMatrix[1] <=
  (nScintWidth+.00001))) &
  ((fPhotonPositionMatrix[2] >= -
  .00001) &
  (fPhotonPositionMatrix[2] <=
  (nScintHeight+.00001))))
{
}
else
{
      nDeadPhoton = 1;
}
if(nFiberAbsorb == 1)
{
      //cout<<"5\n";
      FiberPath(fPhotonVelocityMatrix,
                fPhotonPositionMatrix,
                fRadiusOfFiber,
               fFiberMidpointPosition,
               dTotalPhotonPath,
               nDeadPhoton,
               nFiberSelecter,
               fLengthOfOutsideFiber);
      if(nDeadPhoton == 0)
      {
            //before this
  FiberEmit(fPhotonVelocityMatrix,
            dXEmitZero,
            dXEmitOne, dXEmitTwo,
            dXEmitThree,
            dXEmitFour, dXEmitFive,
            dXEmitSix,
            dXEmitSeven, dXEmitEight,
            dXEmitNine,
            dXEmitTen,
            fEmitBoundaryOne,
            fEmitBoundaryTwo,
            nPhotonWavelength);
            nDeadPhoton =
nPhotoMultiplier(nPhotonWavelength,
                 dXPMTZero,
                 dXPMTOne, dXPMTTwo,
                 dXPMTThree,
```

```
                                                dXPMTFour,
                                                 dXPMTFive, dXPMTSix,
                                                 dXPMTSeven,
                                                 dXPMTEight,
                                                 dXPMTNine,
                                                 dXPMTTen);
                                          }
                                          if(nDeadPhoton == 0)
                                          {
                                                nRecievedPhoton = 1;
                                                nPhotonRecievedCount =
                                               (nPhotonRecievedCount + 1);
                                                nPhotonRecievedCount2 =
                                              (nPhotonRecievedCount2 + 1);
                                          }
                                    }
                              }
                        }
                  }
                  while((nDeadPhoton != 1) & (nRecievedPhoton != 1));
                  if(dTotalScintPath != 0)
                  {
                        nScintillatorBins[((int)dTotalScintPath/30)]=
                      (nScintillatorBins[((int)dTotalScintPath/30)] + 1);
                  }
                  if(nTotalBounces <= 199)
                  {
                        nBounceBins[(nTotalBounces/2)] =
                        (nBounceBins[(nTotalBounces/2)] + 1);
                  }
                  nTotalBounces = 0;
                  if(nDeadPhoton == 1)
                  {
                        dTotalPhotonPath = 0;
                  }
            }
            while(nPhotonCount != nNumberOfPhotons);
            if(nProgCount == nUserMuonCount)
            {
                  nEndOfSimulation = 1;
            }
      if(nDeadPhoton == 0)
      {
            int nPlaceholder = 0;
            nPECounter++;
            nPlaceholder = nPhotonRecievedCount2;
            if(nPlaceholder <= 799)
            {
                  nPEBins[(nPlaceholder/2)] = (nPEBins[(nPlaceholder/2)] +
1);
            }
            nPhotonRecievedCount2 = 0;
      }
}
while(nEndOfSimulation != 1);
clrscr();
do
```

```
	{
		do
		{
		cout<<"The number of muons thrown is "<<nUserMuonCount<<".\n";
		cout<<"The average number of photons thrown is
		    "<<(fSumOfAllPhotons/nUserMuonCount)<<".\n";
		cout<<"The average number of photons detected is
		    "<<(nPhotonRecievedCount/nUserMuonCount)<<".\n";
		cout<<"\n\n";
		cout<<"              1. Create Photoelectron Data Points (Bin Size
		       = 3).\n"
		       <<"              2. Create Number of Bounces Data Points
		       (Bin Size = 1).\n"
		       <<"              3. Create Scintillator Path Data Points
		       (Bin Size = 30).\n"
		       <<"              4. Exit.\n\n"
		       <<"These will overwrite the previous data files, be sure to
		       transfer any\n"
		       <<"files you want to keep out of the program directory.\n\n";
		cout<<"Input Choice. ";
		cin >> nDataSelecter;
		clrscr();
		}
		while((nDataSelecter != 1) & (nDataSelecter != 2) & (nDataSelecter
!= 3)
		            & (nDataSelecter != 4));
		nCounter3 = 0;
		if(nDataSelecter == 1)
		{
		      FILE *stream;
		      stream = fopen("PEData.txt", "w");
		      do
		      {
		            if(nPEBins[nCounter3] != 0)
		            {
		                  fprintf(stream, "%d
		                          %d\n",(nCounter3*3),nPEBins[nCounter3]);
		            }
		            nCounter3++;
		      }
		      while(nCounter3 != 300);

		      fclose(stream);
		      clrscr();
		      cout<<"The data has been written to PEData.txt.";
		      getch();
		      clrscr();
		}
		if(nDataSelecter == 2)
		{
		      nCounter3 = 0;
		      FILE *stream;
		      stream = fopen("BounceData.txt", "w");
		      do
		      {
		            if(nBounceBins[nCounter3] != 0)
		            {
```

```
                        fprintf(stream, "%d
                                %d\n",(nCounter3),nBounceBins[nCounter3]);
                  }
                  nCounter3++;
            }
            while(nCounter3 != 80);
            fclose(stream);
            clrscr();
            cout<<"The data has been written to BounceData.txt.";
            getch();
            clrscr();
      }
      if(nDataSelecter == 3)
      {
            nCounter3 = 0;
            FILE *stream;
            stream = fopen("ScintData.txt", "w");
            do
            {
                  if(nScintillatorBins[nCounter3] != 0)
                  {
                        fprintf(stream, "%d
                                %d\n",(nCounter3*30),
                                nScintillatorBins[nCounter3]);
                  }
                  nCounter3++;
            }
            while(nCounter3 != 300);
            fclose(stream);
            clrscr();
            cout<<"The data has been written to ScintData.txt.";
            getch();
            clrscr();
      }
}
while(nDataSelecter != 4);
clrscr();
do

{
      cout<<"Do you want to run the program again? 1 = Yes 0 = No\n";
      cin >> nUserRestart;
      clrscr();
}
while((nUserRestart != 1) & (nUserRestart != 0));
}
while(nUserRestart == 1);
}
```