The Alpha Particles program
version 2.0

Program file: AlphaParticles2_Main.py

Author: Kyrollos Iskandar

RMIT University

Author's program of study: Master of Medical Physics
Course: Programming Fundamentals for Scientists

Date: 25 Oct. 2020

Contents

1. Introduction

1.1. Objective of the program

The Alpha Particles 2.0 program tries to answer the following questions:

- "How far do alpha particles of a given initial kinetic energy travel in a given medium in the case when they all have the same initial kinetic energy?"
- "What properties must an alpha particle beam have for being used to deliver a particular absorbed dose of radiation to human tissue?" This question is related to radiotherapy.

The program also quizzes the user about the attenuation of an alpha particle beam through a given material.

1.2. Specifications of the program

The program must model the collisions of alpha partis with atomic electrons in a given medium for calculating how far into the medium the beam of alpha particles penetrates [1]. "*The measures of this penetration distance are the [mean and maximum ranges]. The mean range is the distance before which 50% of all of the particles stop and after which the other 50% of the particles stop. The maximum range is the distance travelled by the particle that penetrates the furthest.*"

"*The program must show the user the results of the calculations as well as data about the random numbers it generated*" [1] in the case where the user runs the program in beam analysis mode. In this mode, a user-defined number of simulation instances are run and the final mean and maximum ranges of the simulation are the averages of all instances.

The program must also allow the user to learn a bit about what properties an alpha particle beam must have if it is to deliver a particular dose to human tissue in the context of radiotherapy as well as about how much particular materials can attenuate an alpha particle beam. These two things are the radiotherapy game and attenuation quiz modes of the program.

More detail about the beam analysis, radiotherapy game and attenuation quiz modes are in Section 1.4.

1.3. Relation of the Alpha Particles 2.0 program to my program of study

"*Ionising radiation is a big part of Medical Physics. Medical Physics is the application of Physics to Medicine. The effects of ionising radiation on human tissue is important in radiotherapy and diagnostic imaging as well as the design of special shielding for keeping the occupants of a room with a radiation generator and of adjacent rooms safe from receiving too much radiation*" [1].

One of the forms of ionising radiation are alpha particles. Alpha particles are classified as heavy, charged, sub-atomic particles and are helium nuclei [2]. "*When they enter a medium, they interact with the electrons in the medium through the Coulomb force because both electrons and alpha particles have an electric charge. It is very rare, around* 10^{-15} *times as likely, that the alpha*

particles will interact with the atomic nuclei because atomic electrons occupy around 10^{15} *times more volume of an atom than does an atomic nucleus. The collisions that alpha particles have with atomic electrons and nuclei can be treated as billiard-ball-style collisions, meaning that the alpha particles and electrons behave as solid billiard balls. For every collision an alpha particle has with an electron or nucleus, it loses kinetic energy. Eventually, the alpha particle has lost so much energy that it is travelling slow enough to capture two surrounding electrons and become a helium atom. At this point, the alpha particle has left the beam it was in*" [1].

The alpha particles are around 7300 times heavier than atomic electrons, meaning that an alpha particle loses only a tiny fraction of its kinetic energy per collision with an electron [2, 3]. "*It takes thousands of collisions with stationary atomic electrons for an alpha particle to lose all of its kinetic energy. The kinetic energies of the atomic electrons are so small compared to the energies of alpha particles in a beam that the electrons can be considered to be stationary*" [1].

1.4. Features of the program

The program has six key features. The first three features are the fundamental ones. The second three features are modes in which the program can be run.

1.4.1. Feature 1: Inputs

The program takes the follow inputs from the user, some of which may not be asked for depending on which of the three modes of the program the user runs it in:

- The initial kinetic energy of the alpha particles.
- The initial number of alpha particles in the beam.
- The probability density function, also known as a random distribution, to use for generating the random numbers for updating the alpha particles' momentum vectors at each simulation step.
- Whether or not to seed the random number generators used in the program.
- Which seed to use for seeding the aforementioned random number generators. Note that the user must provide an integer.
- The atomic number, atomic weight and mass density of the medium through which the alpha particles are to travel through. Note that these inputs are asked for only in the beam analysis mode.
- The number of instances to run for the simulation. Note that this input is asked for only in the beam analysis mode.
- The height and width of the alpha particle beam. Note that this input is asked for only in the radiotherapy game mode.
- One or more user-predicted thicknesses of materials/media that the alpha particle must travel through. Note that these inputs are asked for only in the attenuation quiz mode.

The user has the option to provide an input file for all the three beam analysis, radiotherapy game and attenuation quiz modes in place of manually entering their inputs. However, the format of the input files is very strict.

There is a second input file for the attenuation quiz mode that the program, not the user, uses for randomly selecting a medium to ask about in the quiz. However, the user may add more media to the list in the file if they wished to, as long as they adhere to the format of the file.

1.4.2. Feature 2: Probabilistic computation

1.4.2.1. Random number generation

The vast majority of the random numbers that the program generates are used for updating alpha particles' momentum vectors during simulations. "*The matrix is reconstructed for every alpha particle per simulation step*" [1].

Other random numbers are used for randomised selections of options such as which sign to assign to random absolute numbers and which medium to ask about in the attenuation quiz.

1.4.2.2. Calculations for and after simulations

The calculations during simulations involve starting with a collection of alpha particles all with the same initial momentum vector and position along the x-axis (x-position). The x-axis is the beam's travel direction. There are no y- or z- Cartesian axes except implied/pseudo- y- and z- axes in the radiotherapy game mode when specifying the beam's height and width. The program generates a free path magnitude and a randomised 3x3 matrix for updating the momentum vector for each alpha particle per simulation step. The free path is randomly calculated from the mean free path of the particles in the given medium. The program keeps repeating these actions while removing the alpha particles that have momentum magnitudes below a preset minimum magnitude until there are no more particles in the beam. The program keeps track of all of the data of each alpha particle in a large numpy array. In each simulation step, the program extracts x-positions, and the momentum magnitudes if the radiotherapy game mode is being done, for later analysis.

Once the simulation has ended, the program calculates the mean and maximum ranges from the array of x-positions. The mean range is calculated only in the beam analysis mode. In the radiotherapy game mode, the radiation dose absorbed into human tissue is also calculated from the array of momentum magnitudes.

See Section 1.4.4 for more information above the beam analysis mode.

1.4.2.3. Other record-keeping for simulations

"*[T]he program [also] records every value that the off-diagonal elements of the randomised 3x3 matrix [for updating momentum vectors] take for every alpha particle per simulation step*" [1] and plots them as histograms. This allows the user to see how the random number generation for updating the alpha particles' momentum vectors affected the results of the simulation.

1.4.3. Feature 3: Outputs

The program exports the results of the beam analysis and attenuation quiz modes to a directory that was made when the program was started. The files exported in the beam analysis mode are a plot of the number of alpha particles in the beam as a function of distance through the user-specified medium, a data file containing a record of the user's inputs and the mean and maximum ranges of the overall simulation, and one histogram for each of the six off-diagonal elements of the randomised 3x3 matrix that was used to update the alpha particles' momentum vectors. For the attenuation quiz, only the plot is exported.

For all modes, a record of the execution times of particular methods and sections of code is exported as CSV files. These files can be easily formatted by using a macro that Kyrollos defined in the macro-enabled Microsoft Excel file named *ExcelMacros_AlphaParticles2.xlsm*. The macro can be activated using **Ctrl + Shift + E** provided that the macro-enabled file is open.

1.4.4. Feature 4: Statistical analysis of an alpha particle beam

In the beam analysis mode, multiple simulation instances can be run. After they have finished, the x-position array of each instance is used to calculate the mean and maximum ranges and the number of alpha particles in the beam as a function of distance in the instance. These three things are then taken from each instance and merged into dictionaries of mean and maximum ranges and the number of particles as a function of distance. The average and 3 times the population standard deviation of the mean and maximum range for the simulation as a whole are calculated, while the numbers of particles as a function of distance from each instance are plotted together in one figure. It was challenging to calculate the average and population standard deviation of the number of particles as a function of distance because each instance yielded datasets of different sizes and the distance of each particle was calculated using random number generation.

1.4.5. Feature 5: Radiotherapy game

The radiotherapy game mode approaches the use of an alpha particle beam from the perspective of calculating how much kinetic energy is loses to calculate the amount of radiation dose that the medium absorbs (Equation 1.4.5.1). The kinetic energy that the beam loses is equal to the energy that the medium gains as a result of the beam due to the conservation of energy. The user defines the height and width of the beam while the depth that the beam travels into the medium is determined from its simulated maximum range. These three spatial dimensions are used to calculate the volume V_{med}, and consequently the mass of the medium m_{med} (Equation 1.4.5.2), into which the beam deposited energy.

$$A_D = \frac{E_D}{m_{med}}$$

(Equation 1.4.5.1)

$$m_{med} = V_{med}\rho_{med}$$

(Equation 1.4.5.2)

where A_D, E_D, m_{med}, V_{med}, ρ_{med} are the absorbed dose in units of Gy = J/kg, the kinetic energy deposited into the medium in units of J, and the mass in units of kg, volume in units of m^3 and mass density in units of kg m^{-3} of the part of the medium that absorbed the energy, respectively.

The program then compares the calculated absorbed dose A_D to a preset goal absorbed dose to check if the user predicted a correct set of characteristics for the alpha particle beam. The user predicts the characteristics correctly when the calculated absorbed dose is within 5% of the preset goal absorbed dose. Otherwise, the program tells them that they either deposited too little or too much dose.

1.4.6. Feature 6: Attenuation quiz

The attenuation quiz mode approaches the use of alpha particle beam from the perspective of determining the thickness of one or more given materials that are required to attenuate, that is, reduce the intensity of, the alpha particle beam by a particular amount. The program asks the user to predict what thicknesses of one or more randomly selected materials are required to attenuate the beam so that the specified percentage of the beam is transmitted through it. It then runs a simulation to test their prediction, tells them whether they predicted correctly or a thickness that was too small or too large, and tells them a combination of correct thicknesses. The quiz allows the user to consider any number of materials.

2. Methodology

The flow of the program is shown by the flowchart in Figure 2.1.

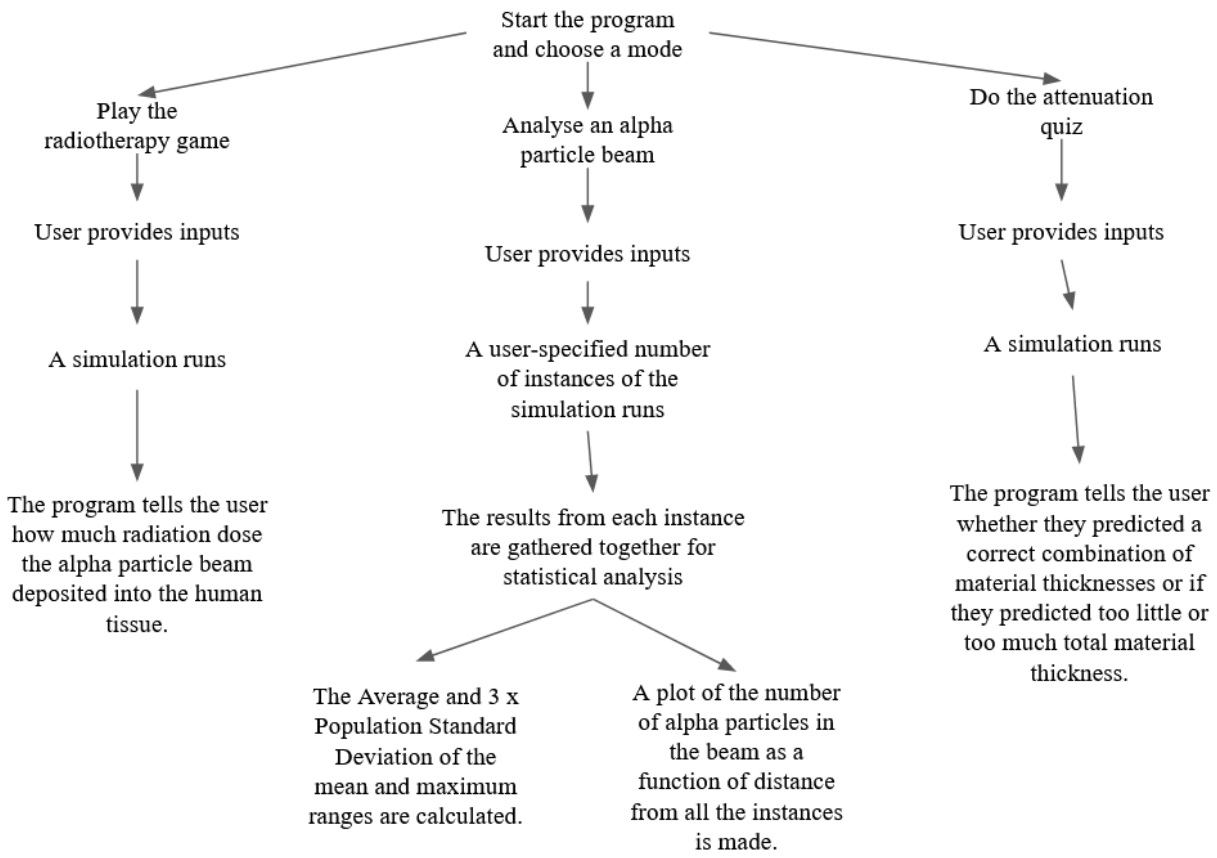

Figure 2.1: A high-level overview of the flow of the program in terms of its three modes.

2.1. Design of the program

"*The preliminary step in coding the program was to decide on how to structure the code*" [1]. Object-Oriented Programming (OOP) was chosen because there can be multiple methods with the same name but with either slightly varied or more different functionality depending on which class they belonged to [4]. This approach was easier than defining functions at the global level because functions defined at the global level must have unique named. This means that the script file(s) will probably have more code compared to the code in OOP.

"*Afterwards, the first thing that had to be decided upon for coding the simulation was the model for the collisions between the alpha particles and atomic electrons. Collisions of the alpha particles with atomic nuclei were ignored to simplify coding the simulation in its early stages*" [1]. Such collisions would happen only around 10^{-15} as often as collisions with atomic electrons, on average [2], so it was safe to ignore them [1].

"*When an alpha particle collides with an electron, not only does it lose some kinetic energy, but its direction of travel can also change. Thus, the program had to deal with momentum vectors, which both have a magnitude and a direction. A vector can be represented in Python as a column vector.*

And kinetic energy E_K is related to momentum $\boldsymbol{p}_\alpha$ by Equation [2.1.1] assuming that the alpha particles of rest mass m_α are travelling at non-relativistic speeds" [1].

$$E_K = \frac{p_\alpha^2}{2m_\alpha}$$

(Equation 2.1.1)

"*Equation [2.1.1] provided a way for the user's input of the initial kinetic energy $E_{K,initial}$ to be converted to a momentum vector for each of the alpha particles. The Cartesian coordinate system was chosen for specifying the vectors and matrices that modelled the collisions. The positive direction of the x-axis was chosen to be the initial direction of travel for all of the alpha particles. Thus, the positive square root of $\boldsymbol{p}_\alpha$ in Equation 2.1 was chosen (see Equation [2.1.2])*" [1].

$$\boldsymbol{p}_{\alpha,x} = \sqrt{2m_\alpha E_K}\,\hat{\boldsymbol{p}}_{\alpha,x}$$

(Equation 2.1.2)

"*where $\hat{\boldsymbol{p}}_{\alpha,x}$ is the unit vector, that is, the direction, of $\boldsymbol{p}_{\alpha,x}$.*
Thus, the initial momentum vector of all of the alpha particles was"

$$\begin{pmatrix} p_{\alpha xi} \\ p_{\alpha yi} \\ p_{\alpha zi} \end{pmatrix} = \begin{pmatrix} \sqrt{2m_\alpha E_{K,initial}} \\ 0 \\ 0 \end{pmatrix}$$

(Equation 2.1.3)

"*The program now needed a 3x3 matrix to transform the initial momentum vector to another momentum vector while adhering to the physics of the collision of an alpha particle with a stationary electron (Equation [2.1.4])*" [1].

$$\begin{pmatrix} p_{\alpha xf} \\ p_{\alpha yf} \\ p_{\alpha zf} \end{pmatrix} = \begin{pmatrix} C_{xx} & C_{yx} & C_{zx} \\ C_{xy} & C_{yy} & C_{zy} \\ C_{xz} & C_{yz} & C_{zz} \end{pmatrix} \begin{pmatrix} p_{\alpha xi} \\ p_{\alpha yi} \\ p_{\alpha zi} \end{pmatrix}$$

(Equation 2.1.4)

"*where $\begin{pmatrix} C_{xx} & C_{yx} & C_{zx} \\ C_{xy} & C_{yy} & C_{zy} \\ C_{xz} & C_{yz} & C_{zz} \end{pmatrix}$ is the randomised 3x3 matrix and $\begin{pmatrix} p_{\alpha xf} \\ p_{\alpha yf} \\ p_{\alpha zf} \end{pmatrix}$ is the alpha particle's momentum vector at the end of the simulation step.*"

"*For a given initial momentum vector, an alpha particle may collide with an electron either head-on or off-centre. Sometimes, the alpha particle may even miss the electron, that is, not collide with it at all*" [1]. The cases of whether or not the alpha particle would collide with an electron was incorporated into the free path of each particle via a "*Monte Carlo determination of distance … from*

an arbitrary point of departure to [the point of a collision]" [5] (Equation 2.1.5). It was assumed that the medium is "*both infinite and homogeneous*".

$$l = -l_{mean} \ln(1 - r)$$

(Equation 2.1.5)

where l is the free path of an alpha particle and l_{mean} is the mean, or average, free path of all alpha particles in the medium. Both the former and latter are variables for each alpha particle. However, the former varies randomly from collision to collision while the latter depends on the cross-section of the particle, which in turn depends on its kinetic energy. In the context of the program, a particle's cross-section is a measure of the probability that it will interact with an electron in the medium. r is a random number between and including 0 and 1. Equation 2.1.5, along with the PositionXArray numpy array described later in this Section, allowed the transition from describing the number of alpha particles remaining in the beam as a function of simulation step [1] to describing it as a function of distance that the beam travelled through the medium. Equation 2.1.5 is implemented in the update_AlphaParticlePosition() method of the AlphaParticles class.

A Table, in the form of a numpy array, was made for keeping track of the important variables for each alpha particle. The contents of the array, named "AlphaParticlesInfoList_ID_X_Momentum", is listed below. Each alpha particle had a value for each variable listed below:

- An identification (ID) number.
- The position of the particle along the x-axis (x-position).
- One column for each of the three components (x-, y-, z-) of the particle's momentum vector.
- The momentum magnitude of the particle.
- The energy-dependent cross-section of the particle.
- The mean free path of the particle in the medium.

For the attenuation quiz mode, the following variables were added as columns to this array for keeping track of which medium each alpha particle was in during the attenuation quiz simulation:

- The atomic number of the medium that the particle is in.
- The atomic weight of the medium that the particle is in.
- The mass density of the medium that the particle is in.
- An index for which medium the particle is in. This index was used to index a pandas DataFrame which had a record of all of the randomly selected media for the attenuation quiz simulation.
- A checkpoint subtotal thickness after which the particle will be considered as being in the next medium.

"*The randomised 3x3 matrix [for updating alpha particles' momentum vectors] had to have some constraints in order to accurately model the collisions. The first constraint was that, in any given simulation step, the final momentum of an alpha particle ($p_{\alpha,final}$) had to be less than its initial momentum ($p_{\alpha,initial}$) in terms of magnitude*" [1] (Equation 2.1.6).

$$p_{\alpha,final} < p_{\alpha,initial}$$

(Equation 2.1.6)

"*The simplest collision an alpha particle can have with a stationary atomic electron is a head-on collision because the travel direction of the alpha particle does not change. It only loses some kinetic energy. Therefore, the randomised 3x3 matrix had to have the following form for modelling such collisions*" [1] (Equation 2.1.7).

$$\begin{pmatrix} C_{xx} & C_{yx} & C_{zx} \\ C_{xy} & C_{yy} & C_{zy} \\ C_{xz} & C_{yz} & C_{zz} \end{pmatrix} = \begin{pmatrix} C_{xx} & 0 & 0 \\ 0 & C_{yy} & 0 \\ 0 & 0 & C_{zz} \end{pmatrix}$$

(Equation 2.1.7)

Such a head-on collision obeys Equation 2.1.8 [2].

$$\Delta E_K = E_K \left(\frac{4m_e}{m_\alpha}\right)$$

(Equation 2.1.8)

where ΔE_K is the change in the alpha particle's kinetic energy and m_e is the rest mass of an electron. "*Equation [2.1.8] is an iterative equation because E_K is the alpha particle's kinetic energy just before the collision, not necessarily the kinetic energy it had back at the beginning of the simulation. Equation [2.1.8] had to be implemented into the randomised 3x3 matrix and thus had to be expressed in terms of momenta. We wanted an equation that expressed the physics of the collision in terms of an initial state and a final state. Thus, ΔE_K had to be replaced with Equation [2.1.9]*" [1].

$$\Delta E_K = E_{K,initial} - E_{K,final}$$

(Equation 2.1.9)

"Also, the kinetic energies had to be expressed in terms of momenta. Thus, Equation [2.1.1] had to be applied to Equations [2.1.8] and [2.1.9]. We start with Equation [2.1.10], substitute the kinetic energies for momenta (Equation [2.1.11]) and arrive to an equation for the momentum magnitude at the end of the simulation step ($p_{\alpha,final}$) (Equation [2.1.12])" [1].

$$E_{K,initial} - E_{K,final} = E_{K,initial}\left(\frac{4m_e}{m_\alpha}\right)$$

(Equation 2.1.10)

$$\frac{\left(p_{\alpha,initial}\right)^2}{2m_\alpha} - \frac{\left(p_{\alpha,final}\right)^2}{2m_\alpha} = \frac{\left(p_{\alpha,initial}\right)^2}{2m_\alpha}\left(\frac{4m_e}{m_\alpha}\right)$$

(Equation 2.1.11)

$$p_{\alpha,final} = p_{\alpha,initial}\sqrt{1-\frac{4m_e}{m_\alpha}}$$

(Equation 2.1.12)

"*Equation [2.1.12] satisfies Equation [2.1.6], and we see from it that*"

$$\begin{pmatrix} C_{xx} & 0 & 0 \\ 0 & C_{yy} & 0 \\ 0 & 0 & C_{zz} \end{pmatrix} = \sqrt{1-\frac{4m_e}{m_\alpha}}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(Equation 2.1.13)

"*The next step in creating the randomised 3x3 matrix was to make it also model off-centre collisions. Modelling off-centre collisions involves the off-diagonal elements of the 3x3 matrix. When any off-diagonal element is not 0, the final momentum vector's direction is different from the initial vector's direction*" [1]. Also, we should expect that the changes in direction be only slight because alpha particles are 7300 times heavier than electrons [3]. "*This meant that the off-diagonal elements had to be small. There are infinitely many new directions in which an alpha particle can travel after it has collided off-centre with an electron depending on how it approaches the electron. Remember that we are modelling the collisions as billiard-ball-style collisions. Random number generation is good for modelling such situations. Therefore, the values of all of the six off-diagonal elements of the matrix had to be randomised*" [1].

"*Equation [2.1.13] models the collision in which an alpha particle loses the most kinetic energy*" [1]. Therefore, "*when the off-diagonal elements are not 0*" and the matrix does not have any other constraints, the final momentum magnitudes of alpha particles can be greater than their initial momentum magnitudes. This is physically impossible in the context of an alpha particle beam travelling through a medium in which the atomic electrons are basically stationary compared to the movement of the alpha particles. Therefore, the off-diagonal elements of the randomised 3x3 matrix (left-hand side of Equation 2.1.7), which will be named "RandomMatrix", must have one or more constraints such that, for any particle, the momentum transferred from each of the three components of the initial momentum vector cannot allow that the final momentum magnitude be greater than the initial momentum magnitude. "*Thus, the sum of each column of [RandomMatrix] had to be no more than 1. This meant that each column represented how much of any particular initial momentum vector component was transferred to any of the three components of the final momentum vector. As a bucket of water cannot be emptied of more water than it initially has, no more momentum could be transferred away from any of the initial momentum vector components than what they had. It was permitted that the sum be no more than 1 rather than strictly equal to 1 because Equation [2.1.13] was a valid form of the matrix. This constraint meant that*" [1]

$$C_{xy} + C_{xz} \leq 1 - \sqrt{1-\frac{4m_e}{m_\alpha}}$$

(Equation 2.1.14(a))

$$C_{yx} + C_{yz} \leq 1 - \sqrt{1 - \frac{4m_e}{m_\alpha}}$$

(Equation 2.1.14(b))

$$C_{zx} + C_{zy} \leq 1 - \sqrt{1 - \frac{4m_e}{m_\alpha}}$$

(Equation 2.1.14(c))

"*Equation [2.1.14(a)] was implemented by first generating a random number between 0 and 1 by which to multiply* $1 - \sqrt{1 - \frac{4m_e}{m_\alpha}}$. *The result was assigned to either* C_{xy} *or* C_{xz} *depending on another random parameter named MeFirst. For example, it can be assigned to* C_{xy}" [1], in which case

$$C_{xy} = r_{xy}\left(1 - \sqrt{1 - \frac{4m_e}{m_\alpha}}\right)$$

(Equation 2.1.15(a))

where r_{xy} is a random number between 0 and 1. The low endpoint, 0, was always included in this range. However, the high endpoint, 1, was also included in this range if the user chose a probability density function, also known as a random distribution, that included 1 in its range. After a value has been chosen for the first off-diagonal element in the column of RandomMatrix that we are focusing on in this discussion, another random number r_{xz} would be used to assign a value to the other off-diagonal element in the column. In this case,

$$C_{xz} = r_{xz}\left(1 - \sqrt{1 - \frac{4m_e}{m_\alpha}} - C_{xy}\right)$$

(Equation 2.1.15(b))

The MeFirst variable that governs the order in which Equations 2.1.15(a) and 2.1.15(b) are done can take one of only two values, 0 or 1. If it is 0, Equations 2.1.15(a) and 2.1.15(b) as mentioned above occur. If its value is 1, Equations 2.1.16(a) and 2.1.16(b) occur.

$$C_{xz} = r_{xz}\left(1 - \sqrt{1 - \frac{4m_e}{m_\alpha}}\right)$$

(Equation 2.1.16(a))

$$C_{xy} = r_{xy}\left(1 - \sqrt{1 - \frac{4m_e}{m_\alpha}} - C_{xz}\right)$$

(Equation 2.1.16(b))

These calculations happen once per alpha particle per simulation step, and happen in the same way for the other two columns of RandomMatrix, separately. Once all of the off-diagonal elements have been assigned a value, they, along with the predefined constant diagonal elements C_{xx}, C_{yy} and C_{zz} are brought together into the form of RandomMatrix, that is, a 3x3 matrix. The program then applies it to an alpha particle's momentum column vector via a dot product as shown in Equation 2.1.4, thus updating it.

It was necessary to implement the MeFirst variable because, without it and in the case where the user chooses a uniform probability density function, the probability density function, or histogram, for assigning a random variable to the first off-diagonal element in a given column was indeed uniform (Figure 2.1.1) [1]. However, that for assigning a random variable to the second off-diagonal element in the column was not the same as that for the first element (Figure 2.1.2). The MeFirst variable itself is randomly assigned a value of 0 or 1, each with equal probability. With this variable implemented, the probability density function for assigning random values to the off-diagonal elements becomes somewhere in between Figures 2.1.1 and 2.1.2 (Figures 2.1.3 and 2.1.4). Both of the off-diagonal elements in each column had the same probability density function, although it was not uniform. It was a compromise.

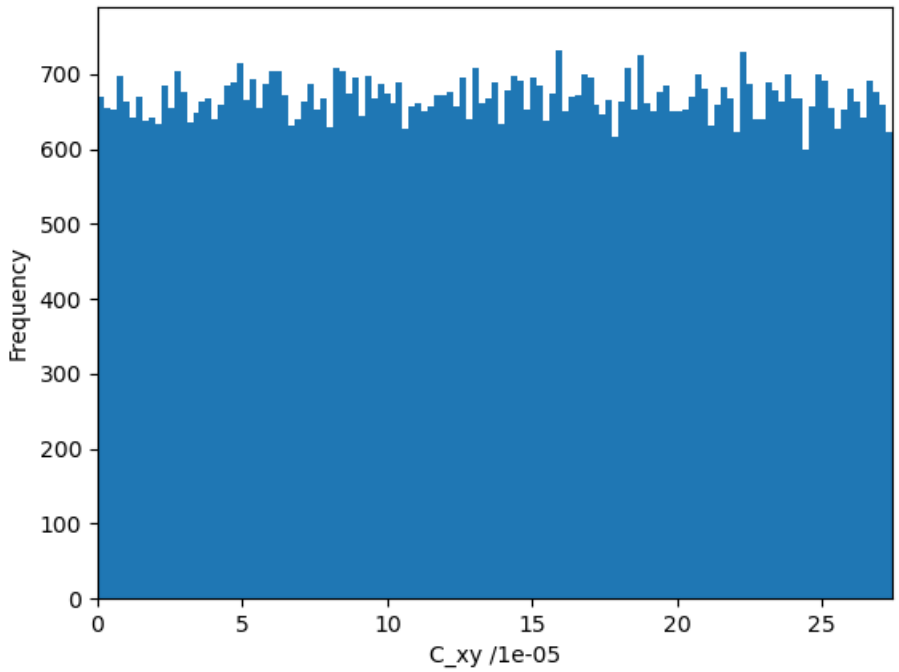

Figure 2.1.1: "*Histogram of the random values assigned to C_{xy} during a simulation in which random numbers were generated according to a discrete uniform probability density function of 10,000 numbers. In this case, the randomly generated number was multiplied with $1 - C_{xx}$*" [1].

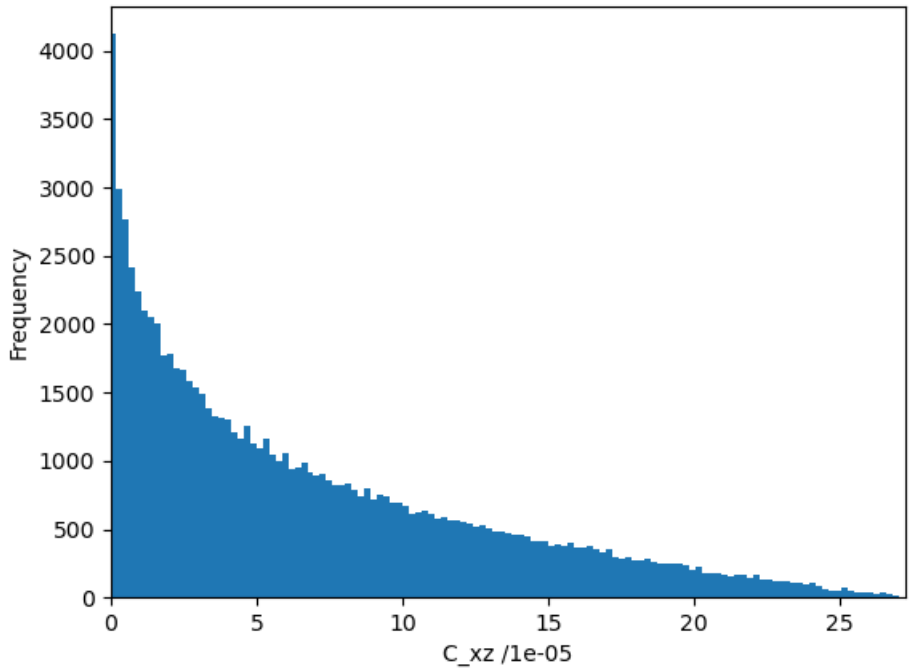

Figure 2.1.2: "*Histogram of the random values assigned to C_{xz} during the same simulation as the one from which Figure [2.1.1] was produced. In this case, the randomly generated number was multiplied with $1 - C_{xx} - C_{xy}$. Note the discrepancy in this histogram compared to the one of C_{xy}. They should have had the same approximate shape!*" [1]

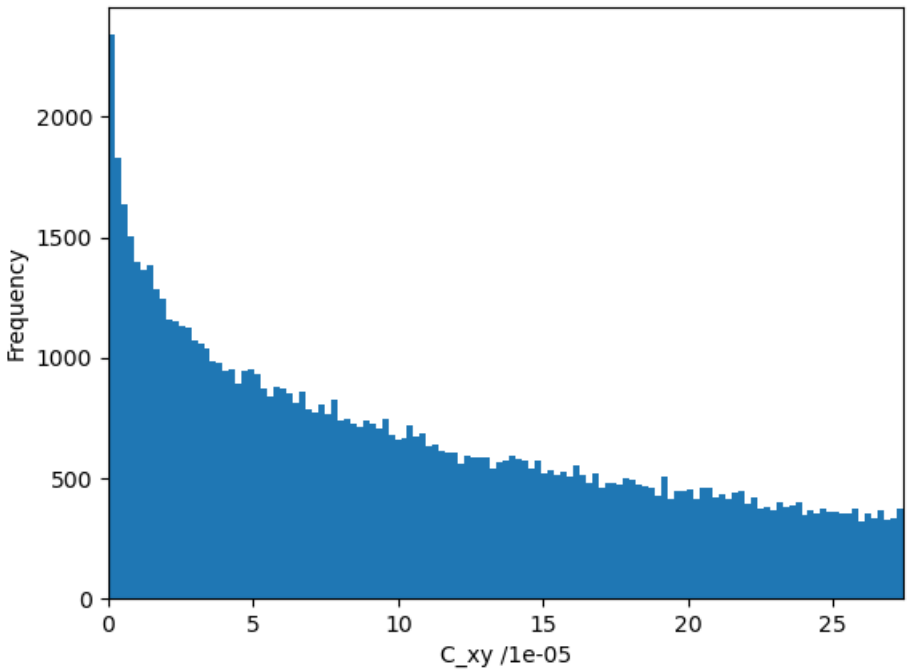

Figure 2.1.3: "*Histogram of the random values assigned to C_{xy} during the same simulation as the one from which Figure [2.1.1] was produced except that the MeFirst variable was implemented. In this case, the randomly generated numbers were multiplied with either $1 - C_{xx}$ or $1 - C_{xx} - C_{xz}$ with equal probability of either choice*" [1] (Equations 2.1.15(a) and 2.1.16(b)).

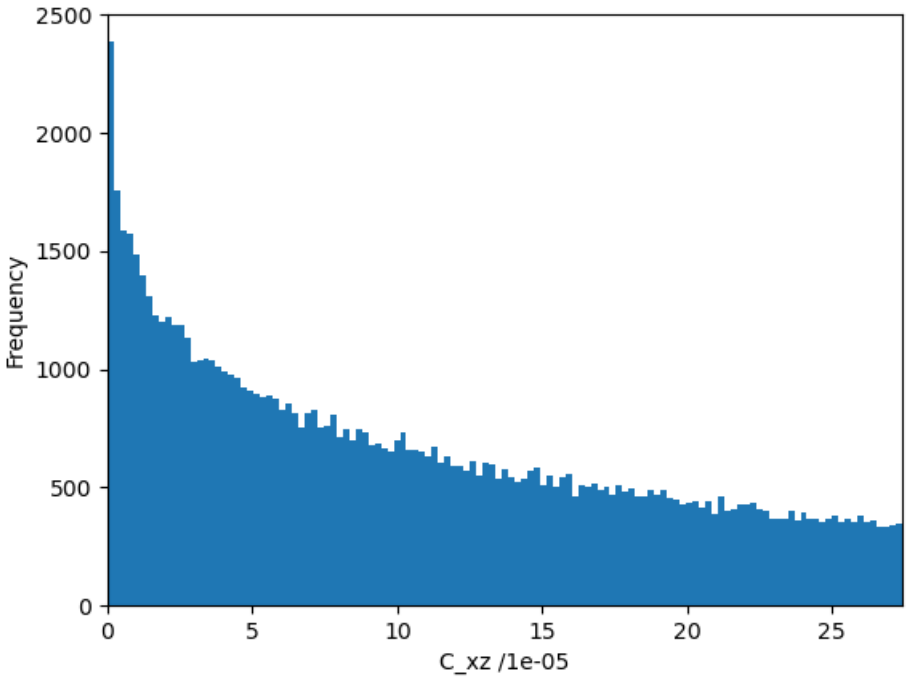

Figure 2.1.4: "*Histogram of the random values assigned to C_{xz} during the same simulation as the one from which Figure [2.1.2] was produced. In this case, the randomly generated numbers were multiplied with either $1 - C_{xx}$ or $1 - C_{xx} - C_{xy}$ with equal probability of either choice*" [1] (Equations 2.1.15(b) and 2.1.16(a)). "*Note how similarly this histogram looks compared to the one of C_{xy} (Figure [2.1.3]).*"

RandomMatrix was reconstructed and applied for each alpha particle in each simulation step. "*This involved the use of two loops, that is, one to loop over the alpha particles in the beam and the other to loop over simulation steps until no more particles remained in the beam. The former loop was a FOR loop while the latter a WHILE loop because the number of particles in the beam at any given simulation step was known while the number of simulation steps before no more particles remained in the beam was unknown*" [1, 6]. "*This implementation was achieved in the update_AlphaParticleMomentum() method*" of the AlphaParticles class. The construction of RandomMatrix itself used to be in that method in version 1.0 of the program [1], but was moved to its own method in the same class, generate_RandomMatrix_Momentum() for improved readability of the code.

"*The next step was to identify which alpha particles had lost enough kinetic energy to leave the beam and to remove them from the beam*" [1]. This was done using a numpy array mask. Alpha particles were removed from the AlphaParticlesInfoList_ID_X_Momentum numpy array mentioned earlier, which is the representation of alpha particle beam in the program, if the magnitudes of their momenta was less than or equal to the momentum corresponding to 1 eV of kinetic energy (see Equation 2.1.1). "*It was thought that 1 eV was a reasonable kinetic energy below which the alpha particles captured surrounding electrons and became helium atoms, thus leaving the beam*" [1]. The new number of particles in the beam was determined by evaluating the number of rows, that is, the shape[0] attribute, of the new AlphaParticlesInfoList_ID_X_Momentum numpy array. The

ParticleNumber variable was used to store the number of particles in the beam in each simulation step. The calculation and storing of the number of particles in the beam was achieved using the update_ParticleNum() method of the AlphaParticles class.

Unlike with version 1.0 of the program which considered all alpha particles to basically be at the same position in each simulation step [1], Equation 2.1.5 meant that the mean and maximum ranges could not be marked during the simulation as in version 1.0. This is why the x-position column in the AlphaParticlesInfoList_ID_X_Momentum numpy array, along with the PositionXArray numpy array, were implemented. Following the execution of the update_AlphaParticlePosition() method, the present x-position of each alpha particle is extracted and appended to a growing array, PositionXArray. PositionXArray thus has as many columns as there were simulation steps. PositionXArray is used to calculate the maximum range, the number of particles in the beam as a function of distance, and the mean range of the particles, in that order. The maximum range could be calculated easily by applying the max() numpy method twice to PositionXArray [7]. The number of particles in the beam as a function of distance would be calculated by first constructing a list of distances to consider from zero distance up to the maximum range that was just calculated. Each alpha particle has its own set of distances due to Equation 2.1.5. Thus, the number of particles that had maximum distances greater than each checked distance was calculated. And from this calculation the mean range was calculated. The mean range is defined as the distance at which only 50% of the initial number of particles remain in the beam [2]. When the initial number of particles is only 1, the mean range is ill-defined and becomes equal to the maximum range.

The aforementioned data would then be exported in files that were saved to a directory named "Data_[timestamp]", where [timestamp] is the timestamp that the program was started in the form YYYY-MM-DD_hh-mm-ss. A directory would be made and used because there are too many files to export to the same directory as that of the script file of the program. All those files had to be in one place rather than be mixed up with other files from other runs of the program.

The user may find that there are too many values to input into the command terminal. Thus, they have the option to provide their inputs via input files. One input file exists for each mode: "InputsForBeamAnalysis.txt" for the beam analysis mode, "InputsForRTGame.txt" for the radiotherapy game mode and "InputsForAttenuationQuiz.txt" for the attenuation quiz mode.

The simulation instance would take some time to run. Thus, the show_Progress() method in the AlphaParticles class was made. While in version 1.0 of the program it showed the progress in terms of the maximum kinetic energy of the particles remaining in the beam [1], the messages printed to the command terminal became too many after the multiprocessing of multiple simulation instances was implemented. These messages were replaced with a progress percentage of each instance that were printed to the terminal periodically by using a delay timer that was made using Python's time module [8]. However, the progress of each instance still depended on the maximum kinetic energy of the particles remaining in the beam.

2.1.1. Additional details about the beam analysis mode

In the beam analysis mode, the program would run one or more instances of the simulation described in Section 2.1 and calculate the mean and maximum ranges and the number of particles in the beam as a function of distance within each instance. It would then gather these results in one place for each of the three variables and, only for the ranges, calculate the average and 3 times the population standard deviation. Also, the values that each of the six off-diagonal elements of RandomMatrix had are gathered into one place for each off-diagonal element and used to plot a histogram of the values for each element. More details about how the program can run multiple instances of the simulation are in Section 2.2. The StatisticalAnalysis class was used to do this.

2.1.2. Additional details about the radiotherapy game mode

Just like for the x-positions and the PositionXArray, the alpha particles' momentum magnitudes would also be extracted and appended in another growing numpy array named AlphaParticleMomentumMagnitudes. This array was used to calculate the total kinetic energy that all of the alpha particles deposited into the medium, E_D mentioned in Section 1.4.5. E_D would be calculated by first calculating how much momentum each alpha particle lost per simulation step and then summing up all these differences. The result of this calculation would be the total momentum transferred to the medium because linear momentum is conserved. This total momentum would then be converted into a kinetic energy, E_D, using Equation 2.1.1.

For calculating the volume of the medium to which the alpha particle beam deposited kinetic energy, V_{med}, the program uses Equation 2.1.2.1.

$$V_{med} = hwR_{max}$$

(Equation 2.1.2.1)

where h and w are the height and width of the beam, respectively, and R_{max} is its maximum range, which would be calculated from the PositionXArray as explained in Section 2.1. Then Equations 1.4.5.2 and 1.4.5.1 would be calculated, in that order.

2.1.3. Additional details about the attenuation quiz mode

The program starts the attenuation quiz by telling the user what it is about. It then reads the "AttenuationQuiz_MaterialLibrary.csv" input file, which is a library of materials/media that the quiz can consider. An example of its contents is shown in Figure 2.1.3.1. The user can add or remove materials from it. Being a CSV file, its appearance is not as shown in Figure 2.1.3.1. The appearance shown is for clarity only. The data were acquired from the CRC Handbook of Chemistry and Physics [9] and National Institute of Standards and Technology [3]. The program makes sure that at least one material/medium is specified in the file before continuing with preparing the quiz. If at least one material is specified, it checks if the specifications are correct. That is, the Material name must be a string, the Atomic number must be an integer, and the other two variables must be floating point numbers.

	A	B	C	D
1	**Material name**	**Atomic number**	**Atomic weight (g/mol)**	**Mass density (g/cm^3)**
2	Lithium metal	3	6.941	0.534
3	Beryllium metal	4	9.012182	1.848
4	Aluminium metal	13	26.981538	2.6989
5	Titanium metal	22	47.867	4.51
6	Iron metal	26	55.845	7.874
7	Molybdenum metal	42	95.94	10.22
8	Palladium metal	46	106.42	12.02
9	Silver metal	47	107.8682	10.5
10	Tungten metal	74	183.84	19.3
11	Gold metal	79	196.96655	19.3
12	Lead metal	82	207.2	11.35
13				

Figure 2.1.3.1: An example of the "AttenuationQuiz_MaterialLibrary.csv" file's contents, ignoring their appearance.

The user is then asked how many media they would like to consider for the quiz. The program asks them this before the program selects media. Once the user specifies this number, the program randomly selects that number of media from the material library and shows the user which ones were selected. Particular media may be selected more than once.

The program then randomly determines the initial kinetic energy of the alpha particle beam and how much of it should be transferred through the total thickness of one or more media. At present, the allowed initial kinetic energies are greater than 1 eV and less than or equal to 1 keV. The random.uniform() method was used so that the high endpoint is included [10]. However, it also included the low endpoint. Thus, a **while** loop was implemented so that the initial kinetic energy was reselected in the case it was 1 eV. A small maximum energy was chosen to keep simulation times short. However, it may be increased if the simulation for the attenuation quiz is optimised and/or multiprocessing is implemented for it. The initial number of particles in the beam was chosen keeping in mind both that more particles take more time to simulate but also give more accurate results. An initial particle number of 100 was chosen thirdly because then the number of particles in the beam can easily be read as the percentage of particles in the beam.

Once the quiz is prepared, the user is asked the quiz question. An example question is: "What thickness(es) of the chosen material(s), *together*, in m, will transmit 66.60000000000001% of a beam of 100 alpha particles of energy 4.770e-04 MeV?" At this point, the user may tell the program to read material thicknesses from the *InputsForAttenuationQuiz.txt* input file or they may input them one by one via the command terminal. If the user chooses to provide the inputs via the input file, the program waits for the user to prepare their inputs and asks the user if they are ready for it to read the file. When the program reads the file, it makes sure that the it exists in the first place, as the user can move or delete it, and that its format is correct. A simulation for testing the user's predicted combination of material thicknesses is then run.

After the simulation has run, the program calculates the total thickness of material that transmitted the specified percentage of alpha particles. It can still calculate it even when there is not a datapoint at the specified percentage by linearly interpolating between existing datapoints. The program then compares the user's predicted total thickness to the correct thickness and tell them if they predicted correctly or incorrectly. If they predicted incorrectly, the user is told whether they predicted too much or too little thickness as well as a combination of correct individual material thicknesses. When multiple materials are being considered, there are infinitely many combinations of correct individual thicknesses. Thus, the program just increases the thickness of the last material if the user predicted too little total thickness or scales down all individual thicknesses by a common factor if the user predicted too much total thickness. The common factor is the ratio of the correct total thickness to the total predicted thickness.

2.1.4. Additional details about the program

2.1.4.1. Classes

The hierarchy of classes is shown in Figure 2.1.4.1.1.

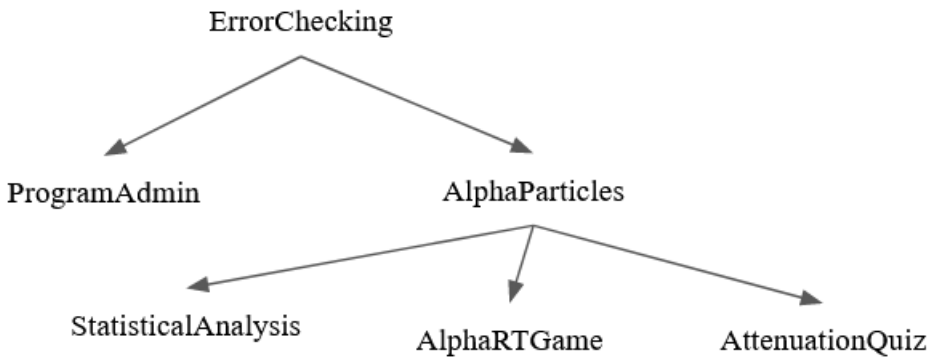

Figure 2.1.4.1.1: The hierarchy of OOP classes in the program. Arrows point from the parent class to the child class.

The purpose of the ErrorChecking module is simply to make sure that any floating point number inputs are within the range of floating point numbers that Python can handle. As of Python 3.8.5, only floating point numbers with magnitudes between and including $2.2250738585072014 \times 10^{-308}$ and $1.7976931348623157 \times 10^{308}$ can be handled. Floating point numbers with magnitudes less than the minimum magnitude are treated as 0.0 while those with magnitudes greater than the maximum magnitude are treated as infinity. Infinity is not a valid input for the program. Also, if the user wanted to input 0.0, they would do so instead of inputting a floating point number with a magnitude greater than 0.0 but less than the minimum magnitude. All other classes of the program use the functionality of the ErrorChecking class and are thus child classes of that class by one or more inheritance levels.

The ProgramAdmin class was made mainly to deal with inputs for the AlphaParticles, StatisticalAnalysis, AlphaRTGame and AttenuationQuiz classes before objects of those classes were

instantiated. It also deals with administrative tasks for the program such as, for example, initialising timers for the timing experiments and introducing the user to the program.

The AlphaParticles class defines the methods and variables that are required for all of the simulations in the program.

The StatisticalAnalysis class handles the objects of the AlphaParticles class from outside it. It was made for gathering the results from all simulation instances and statistically analysing them in beam analysis mode. It also has a method that is used for running multiple simulation instances in parallel with each other.

The AlphaRTGame and AttenuationQuiz classes are specialised for the radiotherapy game and attenuation quiz modes, respectively. They have methods that are used only in the corresponding modes.

2.1.4.2. Global functions

There are two global functions in the program. They are initialise_Program() and main().

The initialise_Program() function was made for defining parameters for the program that must be defined before the classes are even defined. Thus, it is executed before the classes are defined.

The main() function is responsible for executing all of the other code in the program except **import** statements.

2.1.4.3. *ExcelMacros_AlphaParticles2.xlsm*

The *ExcelMacros_AlphaParticles2.xlsm* file is a macro-enabled Microsoft Excel file that is used to quickly change the appearance of the CSV files that are output by the TimerAdmin program for the timing experiments. This file must be open so that the user can use its functionality. Its functionality, which is a macro, is accessed by pressing **Ctrl + Shift + E** on the keyboard. Doing so changes the appearance of the CSV file that it is used in from Figure 2.1.4.3.1 to the appearance shown in Figure 2.1.4.3.2. The CSV file that the macro is to be used in must be selected before the macro is used. The macro works on only Columns A to F in CSV files. Note that the changes made to the appearance of the CSV file on which the macro was used cannot be saved, so it is necessary to reuse the macro if needed.

	A	B	C	D	E	F	G	H	I	J
1		Description	Execution t	Measurem	Number of	Average total execution time per simulation instance /s				
2	9	Attenuatio	0.319995	Once	1	N/A				
3	8	AlphaParti	0.264295	Sum	9038	0.264295				
4	7	AlphaParti	2.92E-05	Average	1	N/A				
5	0	ErrorCheck	0	Once	1	N/A				
6	1	ProgramAc	0	Once	1	N/A				
7	2	AlphaParti	0	Once	1	N/A				
8	3	StatisticalA	0	Once	1	N/A				
9	4	AlphaRTGa	0	Once	1	N/A				
10	5	Attenuatio	0	Once	1	N/A				
11	6	main() fun	0	Once	1	N/A				
12										

Figure 2.1.4.3.1: One of the CSV files that were exported by the TimerAdmin program before any changes to its appearance were made.

	A	B	C	D	E	F
1		**Description**	**Execution time (end - start) /s**	**Measurement type**	**Number of times executed**	**Average total execution time per simulation instance /s**
2	9	AttenuationQuiz calculate_Data()	0.319995165	Once	1	N/A
3	8	AlphaParticles show_Progress() (sum)	0.264294863	Sum	9038	0.264294863
4	7	AlphaParticles show_Progress() (average)	2.92E-05	Average	1	N/A
5	0	ErrorChecking class definition	0	Once	1	N/A
6	1	ProgramAdmin class definition	0	Once	1	N/A
7	2	AlphaParticles class definition	0	Once	1	N/A
8	3	StatisticalAnalysis class definition	0	Once	1	N/A
9	4	AlphaRTGame class definition	0	Once	1	N/A
10	5	AttenuationQuiz class definition	0	Once	1	N/A
11	6	main() function definition	0	Once	1	N/A
12						

Figure 2.1.4.3.2: The CSV file shown in Figure 2.1.4.3.1 after the macro in the *ExcelMacros_AlphaParticles2.xlsm* file has been used by pressing Ctrl + Shift + E on the keyboard.

2.1.4.4. Standard libraries, external modules and other script files

The program uses a number of Python standard libraries and external modules as well as a script file specially for the program, named *TimerAdmin.py*.

2.1.4.4.1. *TimerAdmin.py*

TimerAdmin.py is responsible for administrating the timing experiments that were done for optimising the program and implementing multiprocessing where appropriate. Execution times of particular sections of code were measured within the main program's (Alpha Particles 2.0) script file using the **time** module [8] and added to a growing record of execution times using the TimerAdmin program. The TimerAdmin program also calculates the number of times the particular sections of code were executed by analysing the collected execution times. It also calculates the average execution time of those sections per simulation instance in when the main program runs in beam

analysis mode. Finally, it also deals with exporting the timing results into the same directory as that to which the results of the main program are exported.

2.1.4.4.2. *T1_Concatenation_numpy_pandas.py*

T1_Concatenation_numpy_pandas.py is a standalone program that was made purely for testing the execution times of a series of numpy array concatenations and pandas DataFrame concatenations. It was used for Timing Experiment T1, which is described in more detail in Section 2.2.

2.1.4.4.2. Standard libraries and external modules

The Python standard libraries and external modules that the main program and the TimerAdmin program uses are listed below in the order that their **import** statements appear in the script files.

Standard libraries:

- **math** [11]
  - The **math** module was used for accessing the value of π.
- **statistics** [12]
  - The **statistics** module was used for calculating the averages and population standard deviations of the mean and maximum ranges of the alpha particle beam in the beam analysis mode.
- **datetime** [13]
  - The **datetime** module was used for obtaining a timestamp for naming the files that the program as a whole outputted.
- **os** [14]
  - The **os** module was used to do the following things:
    - Make the directories to which all of the files that the whole program outputs were to be exported.
    - Provide a dynamic way of preventing the main program from being run as an imported module through the use of the **if __name__ == …** statement. The way that was used was dynamic because the name in place of the "**…**" was whatever the script file was named rather than a hardcoded name.
- **sys** [15]
  - The **sys** module was used to obtain the minimum and maximum magnitudes of the floating point numbers that Python can handle.
- **random** [10]
  - The **random** module was used for generating random numbers for the calculations in the simulations.
- **multiprocessing** [16]
  - The **multiprocessing** module was used for parallelising particular sections of code so that multiple instances of them can be run in parallel with each other, thereby reducing the total execution time of the program. More details about the use of this module are in Section 2.2.

- **time** [8]
    - The **time** module was used for measuring the execution time of particular sections of code in the process of optimising the program and implementing multiprocessing where appropriate.

External modules:

- **numpy** [17]
    - **numpy** was used mainly for dealing with arrays and for doing matrix operations.
- **matplotlib.pyplot** [18]
    - This module was used for making and exporting plots of the results of the simulations.
- **pandas** [19]
    - **pandas** was used mainly for reading data from the *AttenuationQuiz_MaterialLibrary.csv* file for the attenuation quiz mode of the main program as well as exporting timing results as CSV files.
    - It was also used for doing the calculations for keeping track of which medium each alpha particle was in in the attenuation quiz simulation.
- **psutil** [20]
    - This module was used for determining the number of physical CPU cores that the computer on which the program is run has. This number was used for defining the number of pooled processes that were made for parallelising particular sections of code using **multiprocessing.Pool()** [16]. Each instance of Python uses one physical core. The **cpu_count()** method of both the **multiprocessing** and **os** modules count the number of logical processors that the CPU has. Some computers have CPUs that have multiple logical processors per physical core, while some have only one per physical core. This means that more pooled processors than physical cores can be made on computers that have CPUs with multiple logical processors per physical core, thus potentially overloading the CPU.

2.1.4.5. Error-handling

"*The approach that was taken for implement error-handling into the program was to look at the code in the order that it would be executed and to think of what might go wrong with any particular section of code. [Besides the initialise_Program() function,] the code of the program is executed in the order it is called … in the main() function*" [1].

Errors are usually dealt with using **try-except** code blocks [21]. The first error-prone part of code is the first input the user provides, which is to choose a mode in which to run the program. While the **input()** function always returns a string, the program accepts only six inputs, which are **a**, **g**, and **q** and their uppercase versions. If the user does not provide one of these six inputs, the program exits. The user is not asked again because they can easily restart the program. However, the program treats errors after a simulation has run more leniently.

Many of the inputs are checked using an **assert** statement [22] inside a **try-except** block. The **except** statement catches the **AssertionError** error, which is raised when the user's input is not what is expected. Inputs provided via the command terminal or via an input file are checked in this way.

Inputs that are in the correct data type but are physically impossible, such as for example a negative initial kinetic energy or initial particle number, are dealt with after the **try-except** blocks by **if-elif** blocks. However, the functionality is still the same as the program still exits if the input is not acceptable. When the user is providing inputs via the command terminal, the program is a bit more lenient and asks the user again for an acceptable input via **while** loops. However, the program exits if the user provides an input of a wrong data type.

try-except blocks are mostly used for dealing with user inputs. The rest of the program follows on from the inputs, meaning that if the user inputs are valid then the rest of the program should not raise errors. This works even when the user provides invalid inputs via an input file. In this case, the program asks the user for a replacement input. Once the user provides a valid input, the program continues reading the input file.

2.2. Timing studies, optimisation and multiprocessing

2.2.1. Methodology of the timing studies

The execution times of all sections of code except those known to be executed extremely quickly such as the assignment statements were measured using the **time.time()** method of the **time** module [8]. The TimerAdmin program was used to gather the results into two Tables. One Table was for the sections of code that were timed while the other was for particular lines that were focused on considering the results in the first Table.

A seed of 0 was used for the random number generators for all of the timing studies so that the timing results were not affected by randomness. Note that each study consisted of multiple experiments.

The inputs below were used for optimising the beam analysis mode of the program. A default number of 500 bins were used for the histograms of the off-diagonal matrix elements of the randomised 3x3 matrix.

Initial kinetic energy of the alpha particles = 1e-4 MeV
Initial particle number = 20
Probability density function = basic
Whether or not to seed = y
Seed = 0
Atomic number of the medium = 1
Atomic weight of the medium = 1 g/mol
Mass density of the medium = 1 g/cm^3
Number of simulation instances = 2

The radiotherapy game mode was also optimised. However, it was already optimised to an extent because it uses some of the same code that the beam analysis mode uses. The inputs for the timing experiments that were done for this mode are shown in blue text below.

Initial kinetic energy of the alpha particles = 1e-2
Initial particle number = 50
Probability density function = basic
Whether or not to seed = y
Seed = 0
Beam height = 1 cm
Beam width = 1 cm

2.2.2. Results of the timing experiments and what they meant for designing the program

Before any optimisation of the analysis of an alpha particle beam, the execution times were as shown in Table 2.2.2.1. We see that the record_AlphaParticlePosition() method of the AlphaParticles class used the most time to be executed based on its total execution time of approximately 20.85 seconds. However, its average execution time was faster than the initialise_Simulation() method of the same class. Its total execution time was longer because it was called once per alpha particle per simulation step whereas the initialise_Simulation() method was called only once per simulation instance. We see that we must focus our initial optimisation efforts on the record_AlphaParticlePosition() and update_ParticleNum() methods of the AlphaParticles class.

Table 2.2.2.1: Execution times of the methods used when analysing an alpha particle beam given the beam analysis inputs mentioned in Section 2.2.1 in blue text. Each description contains the class and method names, where the first is specified before the second. Note: the average total execution time per simulation instance is in units of seconds.

Description	Execution time (end - start) /s	Measurement type	Number of times executed	Average total execution time per simulation instance
AlphaParticles record_AlphaParticlePosition() (sum)	20.41363859	Sum	16797	10.2068193
AlphaParticles update_ParticleNum() (sum)	14.96738362	Sum	16797	7.483691812
AlphaParticles update_AlphaParticleMomentum() (sum)	7.547026396	Sum	16797	3.773513198
AlphaParticles show_Progress() (sum)	6.325416565	Sum	16797	3.162708282
AlphaParticles generate_RandomMatrix_Momentum() (sum)	5.537341118	Sum	335873	2.768670559
StatisticalAnalysis plot_RandomMatrixOffDiagonals()	3.749872446	Once	1	N/A
AlphaParticles update_AlphaParticlePosition() (sum)	3.173489809	Sum	16797	1.586744905
AlphaParticles calculate_Data() (sum)	3.141838551	Sum	2	1.570919275
AlphaParticles calculate_Data() (average)	1.570919275	Average	1	N/A
StatisticalAnalysis plot_ParticleNum()	0.368077278	Once	1	N/A
AlphaParticles randomNum_0to1() (sum)	0.315664291	Sum	2015238	0.157832146
StatisticalAnalysis save_Data()	0.045878887	Once	1	N/A
AlphaParticles initialise_Simulation() (sum)	0.013091803	Sum	2	0.006545901
AlphaParticles initialise_Simulation() (average)	0.006545901	Average	1	N/A
AlphaParticles record_AlphaParticlePosition() (average)	0.001215315	Average	1	N/A
AlphaParticles update_ParticleNum() (average)	0.000891075	Average	1	N/A
AlphaParticles update_AlphaParticleMomentum() (average)	0.000449308	Average	1	N/A
AlphaParticles show_Progress() (average)	0.00037658	Average	1	N/A
AlphaParticles update_AlphaParticlePosition() (average)	0.000188932	Average	1	N/A
AlphaParticles generate_RandomMatrix_Momentum() (average)	1.65E-05	Average	1	N/A
AlphaParticles randomNum_0to1() (average)	1.57E-07	Average	1	N/A
ErrorChecking class definition	0	Once	1	N/A
AlphaParticles get_Results() (average)	0	Average	1	N/A
AlphaParticles get_Results() (sum)	0	Sum		0
StatisticalAnalysis calculate_and_get_Average() (average)	0	Average	1	N/A
StatisticalAnalysis calculate_and_get_Average() (sum)	0	Sum		0
StatisticalAnalysis calculate_and_get_PopulationStandardDeviation() (average)	0	Average	1	N/A
StatisticalAnalysis calculate_and_get_PopulationStandardDeviation() (sum)	0	Sum		0

The record_AlphaParticlePosition() method of the AlphaParticles class has only the code shown below all on one line. It is executed once per simulation step for each simulation instance.

```
self.PositionXList_DF = pd.concat([pd.DataFrame(self.PositionXList_DF), self.AlphaParticlesInfoList_ID_X_Momentum_DF.iloc[:, 1]], axis = 1)
```

It concatenates the x-positions of the alpha particles in the present simulation step to a pandas DataFrame that has a previous record of these x-positions. While it takes around 1 ms to do this, this must be done 16,797 times, and the execution times add up. A loop is not used to do each concatenation. Thus, multiprocessing is not suitable for concatenating the x-positions faster.

The update_ParticleNum() method of the AlphaParticles class has the code shown below. It, too, is also executed once per simulation step for each simulation instance.

```
self.AlphaParticlesInfoList_ID_X_Momentum_DF.iloc[:, 5] = ((self.AlphaParticlesInfoList_ID_X_
Momentum_DF.iloc[:, 2:5] ** 2).sum(axis = 1) ** (1/2))

self.AlphaParticlesInfoList_ID_X_Momentum_DF = self.AlphaParticlesInfoList_ID_X_Momentum_DF.i
loc[:,:][self.AlphaParticlesInfoList_ID_X_Momentum_DF.iloc[:, 5] > self.MinimumAlphaMomentum]

self.AlphaParticlesInfoList_ID_X_Momentum_DF = self.AlphaParticlesInfoList_ID_X_Momentum_DF.r
eset_index(drop = True)

self.ParticleNumber = self.AlphaParticlesInfoList_ID_X_Momentum_DF.shape[0]
```

Also, it, too, deals with pandas DataFrames. The execution times of the first three of the four lines of code shown above were measured (Table 2.2.2.2). It was seen that dealing with pandas DataFrames took a lot of time.

Table 2.2.2.2: The average execution time of each line of code in the update_ParticleNum() method of the AlphaParticles class. The execution times shown in this Table are in the order of the first three lines of code shown above.

Description	Execution time (end - start) /s
AlphaParticles update_ParticleNum() MomentumMagnitudeCalculation (average)	0.000493986
AlphaParticles update_ParticleNum() DataFrameExtraction (average)	0.000311453
AlphaParticles update_ParticleNum() ResetIndex (average)	5.95E-05

An alternative to using pandas DataFrame concatenation for keeping a record of the x-positions is to use a numpy array. Concatenations done using numpy arrays were found to be faster than concatenations done using pandas DataFrames (Figure 2.2.2.1). It appears that numpy is better optimised for the concatenation of columns than pandas is. Therefore, the first step in optimising the program was to convert all of the actions that dealt with pandas DataFrames into actions that dealt with numpy arrays.

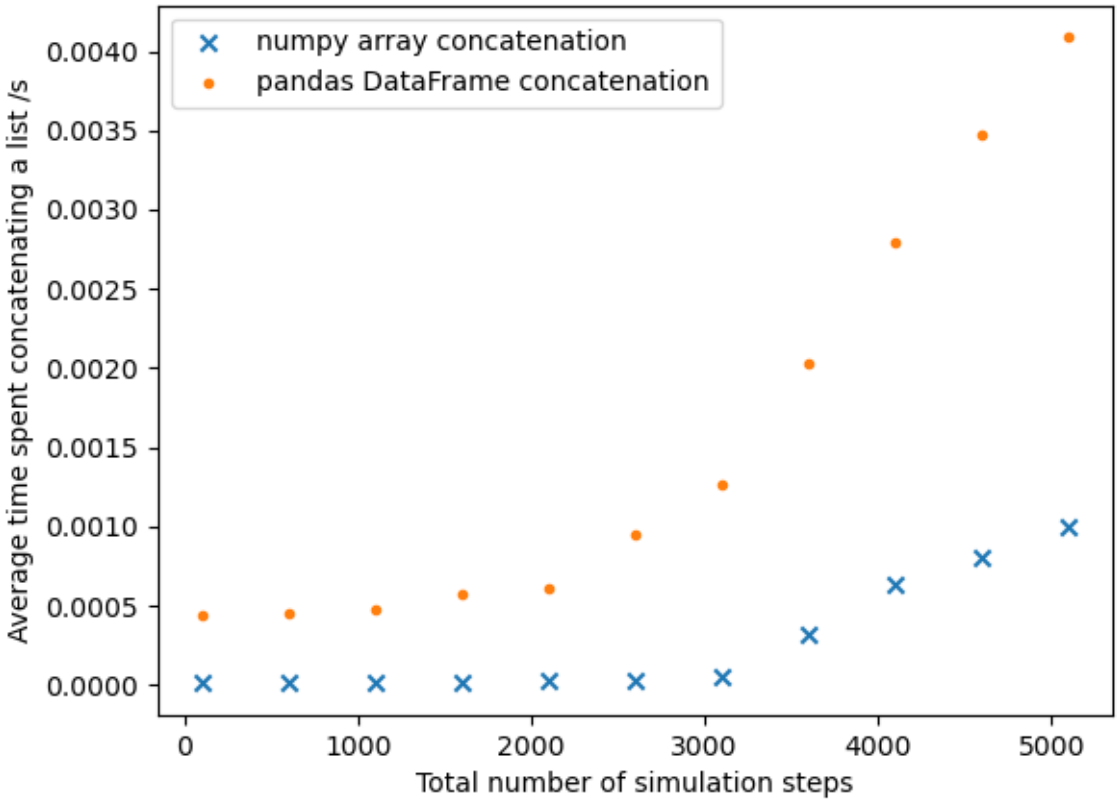

Figure 2.2.2.1: Average concatenation execution times of a numpy array and pandas DataFrame. A list was converted into a numpy array or pandas DataFrame and concatenated to a pre-existing numpy array or pandas DataFrame, respectively. See the accompanying *T1_Concatenation_numpy_pandas.py* file in the *TimingStudies* folder for the code that was used to do the timing experiment and produce this plot.

In the first step in the optimisation of the program, the use of pandas DataFrames was replaced with the use of numpy arrays until the part of the program where the results from the simulation start to be calculated. The program executed considerably faster, as shown in Table 2.2.2.3. The total execution time of the record_AlphaParticlePosition() method was almost 10 times faster than when pandas DataFrames were used to keep the record of the alpha particles' x-positions. The replacement code is shown below.

```
if np.array([self.AlphaParticlesInfoList_ID_X_Momentum[:, 1]]).T.shape[0] == self.PositionXList.shape[0]
:
    self.PositionXList = np.concatenate((self.PositionXList, np.array([self.AlphaParticlesInfoList_ID_X_
Momentum[:, 1]]).T), axis = 1)

else:
    self.AlphaParticlesInfoList_ID_X_Momentum_PositionXListOfPresentSimStep = np.concatenate((np.array([
self.AlphaParticlesInfoList_ID_X_Momentum[:, 1]]).T, np.zeros((self.PositionXList.shape[0] -
 np.array([self.AlphaParticlesInfoList_ID_X_Momentum[:, 1]]).T.shape[0], 1))), axis = 0)
    self.PositionXList = np.concatenate((self.PositionXList, self.AlphaParticlesInfoList_ID_X_Momentum_P
ositionXListOfPresentSimStep), axis = 1)
```

The update_ParticleNum() method was executed 54.6 times faster, where the calculation of the momentum magnitudes and the extraction of the alpha particles that had enough momentum to stay

in the beam being 41.8 and 77.2 times faster than the corresponding operations when pandas DataFrames were used (Table 2.2.2.4).

The **if-else** code block in the record_AlphaParticlePosition() method was necessary because only numpy arrays with the same dimensions in the axis of concatenation can be concatenated, unlike for pandas DataFrames where the missing rows are filled with NaNs which can be replaced later [23]. The **else** section of the code block is executed in the simulation step immediately after the step in which the first alpha particle has left the beam. This is when the shape[0] of Column 1 of the AlphaParticlesInfoList_ID_X_Momentum array becomes unequal to that of the PositionXList array, where the latter is equal to the initial number of alpha particles. From that point, the **else** section is always executed. Using an **if-else** statement instead of a **try-except** block with the **assert** statement did not make a noticeable difference in performance. Therefore, which code block to choose is a matter of preference. The **if-else** block is more like English than the **try-except** block with the **assert** statement even though they do the same thing.

The show_Progress() method's performance improved because it used a calculation of the momentum magnitudes for showing the progress of the simulation. The pandas DataFrame that was used in this calculation was replaced with a numpy array.

Table 2.2.2.3: Execution times of the methods used when analysing an alpha particle beam given the beam analysis inputs mentioned in Section 2.2.1 in blue text after the aforementioned replacement of the use of pandas DataFrames with the use of numpy arrays until the part of the program where the results from the simulation start to be calculated. Each description contains the class and method names, where the first is specified before the second. The methods we were focusing on optimising, as well as the show_Progress() method which showed significant performance improvement, are highlighted in yellow.

Description	Execution time (end - start) /s	Measurement type	Number of times executed	Average total execution time per simulation instance /s
AlphaParticles update_AlphaParticleMomentum() (sum)	5.749205351	Sum	16797	2.874602675
AlphaParticles generate_RandomMatrix_Momentum() (sum)	5.072166443	Sum	335873	2.536083221
StatisticalAnalysis plot_RandomMatrixOffDiagonals()	3.793835402	Once	1	N/A
AlphaParticles calculate_Data() (sum)	3.057856083	Sum	2	1.528928041
AlphaParticles record_AlphaParticlePosition() (sum)	2.861215353	Sum	16797	1.430607677
AlphaParticles calculate_Data() (average)	1.528928041	Average	1	N/A
StatisticalAnalysis plot_ParticleNum()	0.376959801	Once	1	N/A
AlphaParticles show_Progress() (sum)	0.361158371	Sum	16797	0.180579185
AlphaParticles update_AlphaParticlePosition() (sum)	0.320679665	Sum	16797	0.160339832
AlphaParticles update_ParticleNum() (sum)	0.274142265	Sum	16797	0.137071133
AlphaParticles randomNum_0to1() (sum)	0.247474432	Sum	2015238	0.123737216
AlphaParticles initialise_Simulation() (sum)	0.000997305	Sum	2	0.000498652
StatisticalAnalysis save_Data()	0.000997066	Once	1	N/A
AlphaParticles initialise_Simulation() (average)	0.000498652	Average	1	N/A
AlphaParticles update_AlphaParticleMomentum() (average)	0.000342276	Average	1	N/A
AlphaParticles record_AlphaParticlePosition() (average)	0.000170341	Average	1	N/A
AlphaParticles show_Progress() (average)	2.15E-05	Average	1	N/A
AlphaParticles update_AlphaParticlePosition() (average)	1.91E-05	Average	1	N/A
AlphaParticles update_ParticleNum() (average)	1.63E-05	Average	1	N/A
AlphaParticles generate_RandomMatrix_Momentum() (average)	1.51E-05	Average	1	N/A
AlphaParticles randomNum_0to1() (average)	1.23E-07	Average	1	N/A
ErrorChecking class definition	0	Once	1	N/A
AlphaParticles get_Results() (average)	0	Average	1	N/A
AlphaParticles get_Results() (sum)	0	Sum		0
StatisticalAnalysis calculate_and_get_Average() (average)	0	Average	1	N/A
StatisticalAnalysis calculate_and_get_Average() (sum)	0	Sum		0
StatisticalAnalysis calculate_and_get_PopulationStandardDeviation() (average)	0	Average	1	N/A
StatisticalAnalysis calculate_and_get_PopulationStandardDeviation() (sum)	0	Sum		0

Table 2.2.2.4: Average execution time of each line of code in the update_ParticleNum() method of the AlphaParticles class when it handled numpy arrays instead of pandas DataFrames. These results are from the same run of the program as were the results shown in Table 2.2.2.3.

Description	Execution time (end - start) /s
AlphaParticles update_ParticleNum() MomentumMagnitudeCalculation (average)	1.18E-05
AlphaParticles update_ParticleNum() NumpyArrayMask (average)	4.04E-06

As of Table 2.2.2.3, the calculate_Data() method of the AlphaParticles class still handled pandas DataFrames. We then focused on optimising the calculate_Data() method. Although it is executed only once per simulation instance, improving its performance can have a noticeable effect once many more simulation instances are run. It is faster to calculate the maximum value of a numpy array than of a pandas DataFrame with the same dimensions (Figure 2.2.2.2 and Table 2.2.2.5). This method was executed 30 times faster when it handled PositionXList, which is the record of the alpha particles' x-positions throughout the simulation, as a numpy array rather than as a pandas DataFrame. Also, using a numpy array avoided the need to reset the indices of the columns of a pandas DataFrame before data analysis.

```
self.PositionXList.max().max(): 0.0 s
)pd.DataFrame(self.PositionXList).max().max(): 0.013962268829345703 s
pd.DataFrame(self.PositionXList): 0.0 s
```

Figure 2.2.2.2: The execution time of calculating the maximum value in the PositionXList numpy array directly compared to the execution time of doing the calculation after first converting the array to a pandas DataFrame. The execution time of just the conversion was calculated to check whether or not the conversion itself contributed largely to the long execution time for the calculation using the DataFrame. This Figure is a snapshot of output in the command terminal. Note that the PositionXList numpy array was later renamed to PositionXArray to better reflect its datatype.

Table 2.2.2.5: A significant performance improvement in the execution time of the calculate_Data() method of the AlphaParticles class when it handled PositionXList as a numpy array rather than as a pandas DataFrame.

Description	Execution time (end - start) /s	Measurement type	Number of times executed	Average total execution time per simulation instance /s
AlphaParticles calculate_Data() (sum)	0.101725101	Sum	2	0.050862551
AlphaParticles calculate_Data() (average)	0.050862551	Average	1	N/A

While the plot_RandomMatrixOffDiagonals() method in the StatisticalAnalysis class has an execution time that is quite large for just one execution (Table 2.2.2.3), we were not sure how to optimise it because most of the time it spent was in plotting and exporting the histograms of the off-diagonal elements of RandomMatrix, the randomised 3x3 matrix that is used to update the alpha particles' momentum vectors during the simulation. Almost 1.4 as much time, on average, was used to make the plots than to export them (Table 2.2.2.6).

Table 2.2.2.6: Average and total execution times of parts of the plot_RandomMatrixOffDiagonals() method in the StatisticalAnalysis class.

Description	Execution time (end - start) /s
StatisticalAnalysis plot_RandomMatrixOffDiagonals_All() PlotAndExportRandomNumHistograms (sum)	3.990853548
StatisticalAnalysis plot_RandomMatrixOffDiagonals_All() PlotRandomNumHistograms (average)	0.386205276
StatisticalAnalysis plot_RandomMatrixOffDiagonals_All() ExportRandomNumHistograms (average)	0.278936982
StatisticalAnalysis plot_RandomMatrixOffDiagonals() ConcatenationForRandomNumHistograms (once)	0.039892912

Now that the methods used in all three modes of the program were sufficiently optimised, it was time to focus on implementing multiprocessing for the beam analysis mode so that multiple instances of the simulation can be run simultaneously, thus reducing the total execution time of the simulation. The code we are going to focus on is shown below because this is where multiple simulation instances are run.

```
for instance in range(0, SimInstances):
    # Run each simulation instance.
    alpha_beam_dict[instance].process_Simulation(instance) # Run the simulation instances and have the r
esults of each instance ready for collection.

    # Gather the results of each simulation instance.
    MaximumRanges_Dict[instance], ParticleNumDict_Distance_Dict[instance], MeanRanges_Dict[instance] = a
lpha_beam_dict[instance].get_Results() # Collect the results of the simulation instance.
```

All time.time() calls for the timing_experiments object must be protected with the **if __name__ == "__main__"** statement so that the error of the object not being defined is not raised by the slave processes. In the process of implementing multiprocessing, the process_Simulation() method had to be limited to only initialising and running the simulation instead of also calculating the data from each instance. Also, the get_Results() method had to be removed from the pool and the show_Progress() method was also redefined because running multiple simulation instances in parallel caused the command terminal to be flooded as a message of the progress of each instance was printed to the terminal. A delay timer was incorporated into the new definition of this method.

The execution time of the beam analysis mode of the program, minus the time spent waiting for user inputs, was analysed as a function of the number of simulation instances from 1 to 20 instances. All of the inputs into the program were kept the same while the number of simulation instances was varied (Figure 2.2.2.3). The maximum number of pooled processes was set to 1 less than the total number of physical CPU cores that the computer running the program had to avoid potentially stalling or overheating the computer. In this case, the maximum number of pooled processes was 5. It was aim of the timing experiment to determine how having one or more pooled processes running

simulation instances in parallel with each other rather than all instances running in series affected the execution time of the beam analysis mode of the program. Thus, two version of the Alpha Particles 2.0 program were used for the T2_MultiprocessingVsSerial Timing Experiment. A copy of the results of this experiment is available in the *T2_MultiprocessingVsSerial.xlsx* file in the TimingStudies directory. At the time that the experiment was done, both versions were identical except that one of them had multiprocessing Pool implemented while the other did not.

It was also found that the transfer of RAM contents to the hard drive when more RAM than what was available was needed increased the execution time of the program when it ran in serial mode. This was discovered when multiple Windows command terminals ran the serial version of the program at the same time but with different total number of simulation instances. This meant that it was better to use only one instance of Python for running the simulation instances in series.

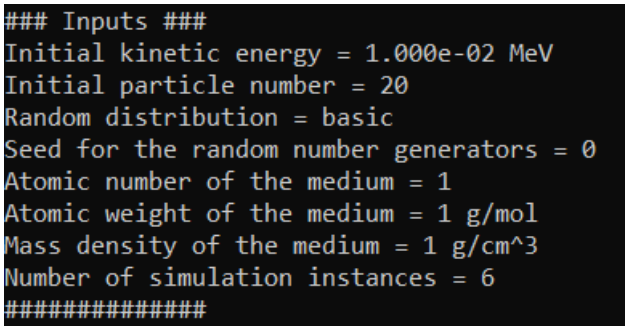

Figure 2.2.2.3: An example of the inputs used for analysing the execution time of the beam analysis mode of the program, minus the time spent waiting for user inputs. Only the number of simulation instances was varied. It was varied from 1 to 20. All of the other inputs stayed the same for all runs of the program.

It was found that the beam analysis mode of the program ran roughly 3 times faster when simulation instances were run up to 5 instances at a time in parallel compared to when they were all run in series (Figures 2.2.2.4 and 2.2.2.5). It was also found that how much faster the multiprocessing version of the program ran compared to the serial version had a period of 5 simulation instances (Figures 2.2.2.5 and 2.2.2.6). This number of 5 instances correlates with there being a maximum of 5 pooled processes running the instances. When there were more instances than pooled processes, one or more of the processes would run more than one instance in series. That is, each of the pooled processes ran up to 5 times less instances in series compared to the serial version of the program. It took around 16 s for the serial version of the program to run each instance (Figure 2.2.2.6). For only one instance, the multiprocessing version ran slightly slower probably because some milliseconds were spent preparing the pool of processes even though only one pooled process was to be used. However, the benefits of multiprocessing showed when more than one simulation instance were run.

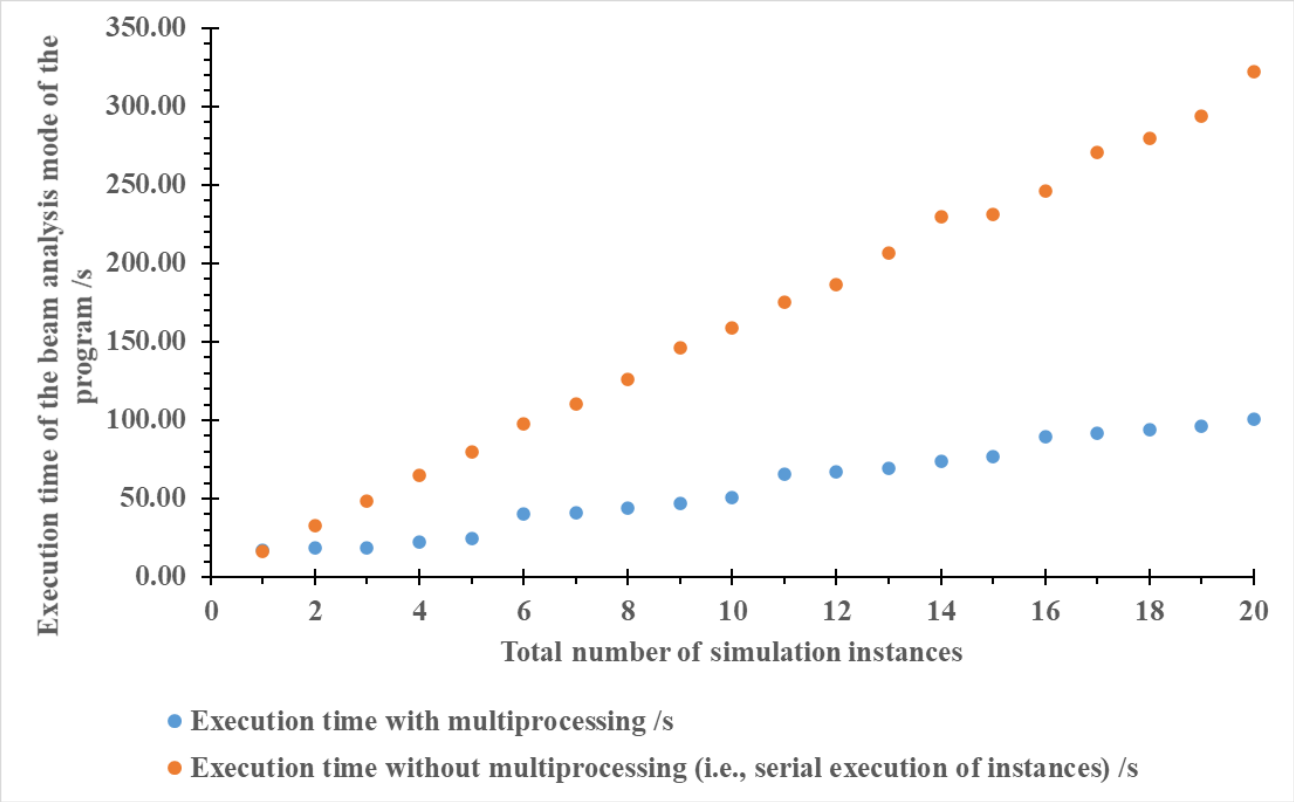

Figure 2.2.2.4: The total execution times of the beam analysis mode of the program when the simulation instances were run up to 5 instances at a time compared to when they were run in series.

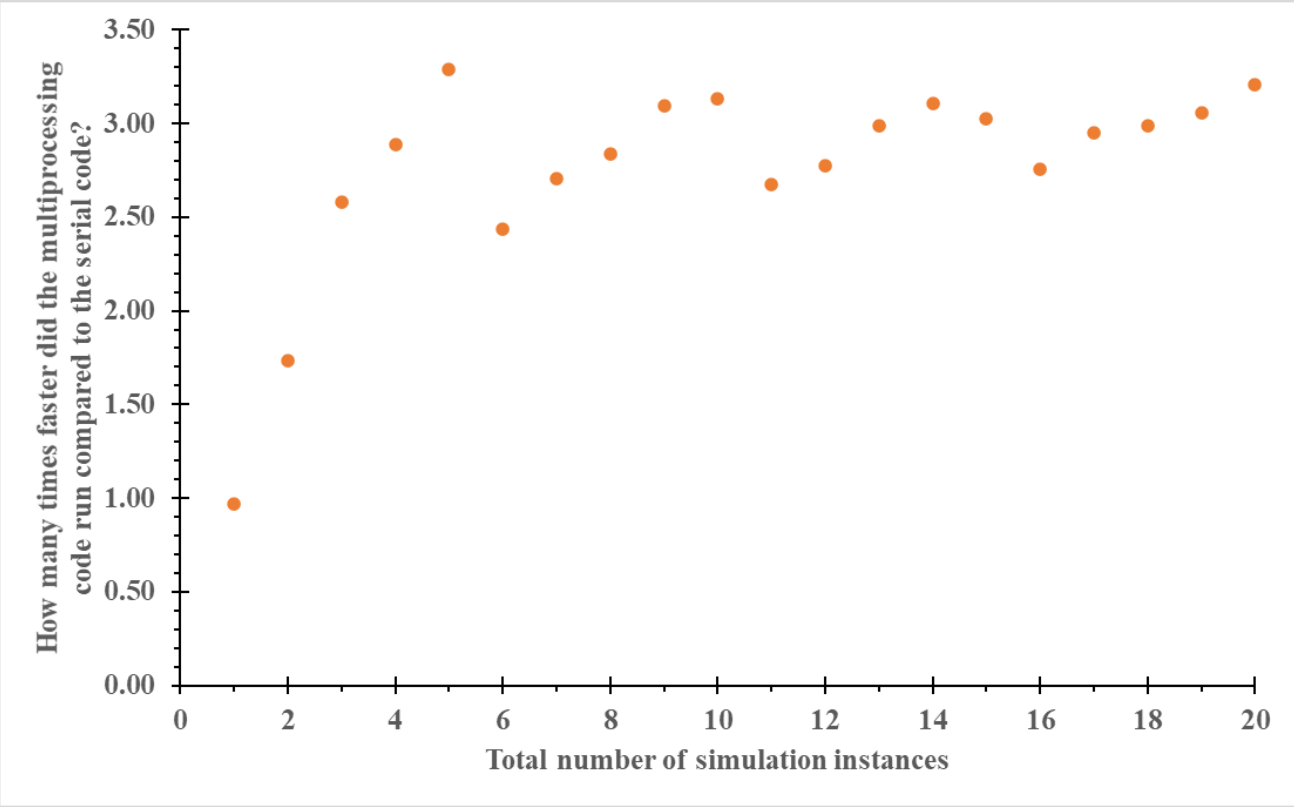

Figure 2.2.2.5: How much faster the beam analysis mode of the program ran when the simulation instances were run up to 5 instances at a time compared to when they were run in series in terms of total execution time.

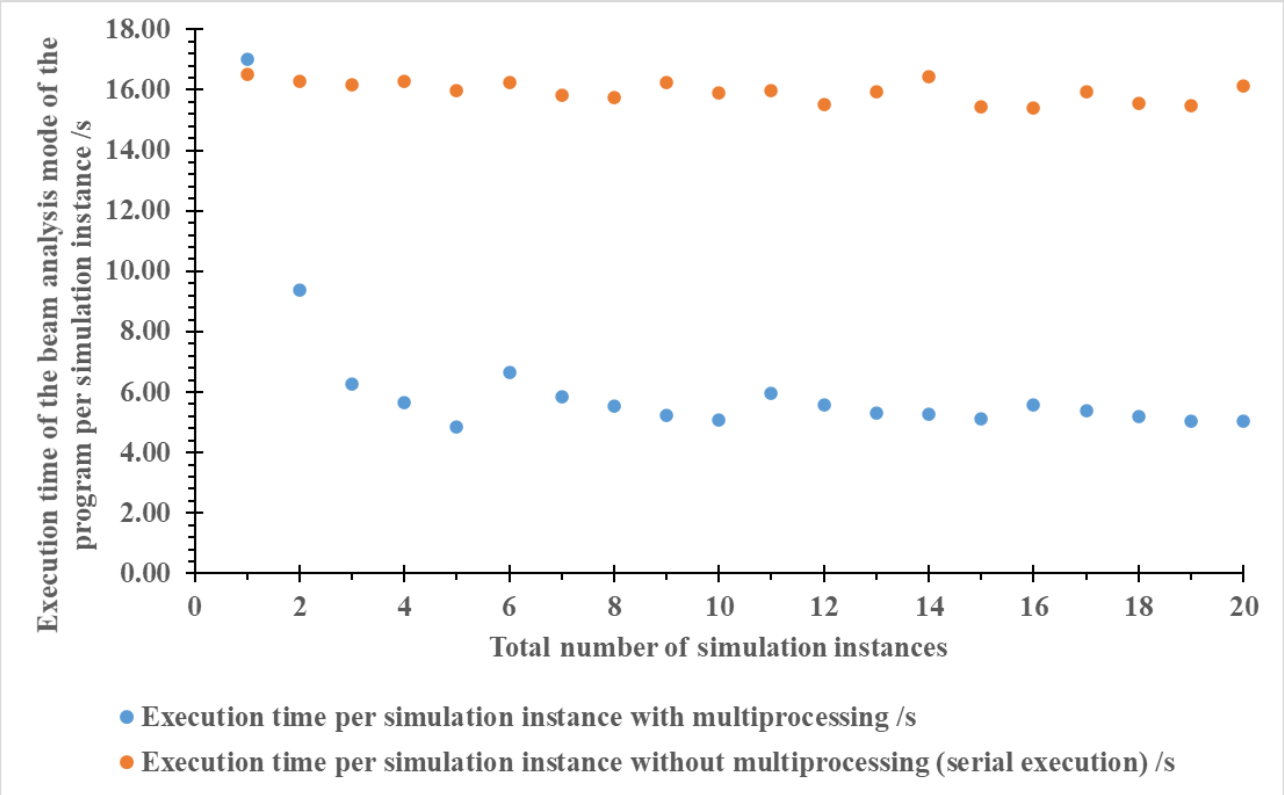

Figure 2.2.2.6: The total execution times of the beam analysis mode of the program per simulation instance when the simulation instances were run up to 5 instances at a time compared to when they were run in series.

Multiprocessing was also implemented twice for the radiotherapy game mode. The first time was for calculating the dose deposited into the human tissue medium by the alpha particle beam. The calculate_DoseToMedium() method of the AlphaRTGame class took the most time to be executed during the runtime of the mode (Table 2.2.2.7). It was also found that all, if not only most, of this execution time was spent using the MomentumTransfer_Array numpy array to calculate the total momentum transfer into the medium from the alpha particle beam (Table 2.2.2.8). Once a simulation is done, there is a huge array with a record of all of the alpha partices' momentum magnitudes at each simulation step, namely the AlphaParticleMomentumMagnitudes numpy array. The total kinetic energy transferred to the medium, which is equal to the total kinetic energy lost by the alpha particles (see Equation 2.1.1), is calculated from this array by calculating the change in momentum magnitude from one simulation step to the next for each alpha particle. However, it was not necessary that these calculations be done in order of simulation step. Thus, multiple multiprocessing.Pool() pooled processes each did some of the calculations. It was found that the calculate_DoseToMedium() method ran, on the average of 5 runs, 8.12 s faster than when all of the momentum transfer calculations were done serially (Table 2.2.2.8). The number of pooled processes that would be started for doing these calculations is equal to the maximum number of physical CPU cores that the

computer running the program has minus one. If the computer's CPU has only one core, then the program would use just that core.

Table 2.2.2.7: The execution times of the methods used in the radiotherapy game mode whose execution times could still be measured after multiprocessing had been implemented for the beam analysis mode.

Description	Execution time (end - start) /s	Measurement type	Number of times executed	Average total execution time per simulation instance /s
AlphaRTGame calculate_DoseToMedium()	25.68964052	Once	1	N/A
AlphaParticles show_Progress() (sum)	0.389547348	Sum	16797	0.389547348
AlphaRTGame game_Outcome()	0.008090973	Once	1	N/A
AlphaRTGame calculate_MaximumRange()	0.000995398	Once	1	N/A
AlphaParticles show_Progress() (average)	2.32E-05	Average	1	N/A
ErrorChecking class definition	0	Once	1	N/A
ProgramAdmin class definition	0	Once	1	N/A
AlphaParticles class definition	0	Once	1	N/A
StatisticalAnalysis class definition	0	Once	1	N/A
AlphaRTGame class definition	0	Once	1	N/A
AttenuationQuiz class definition	0	Once	1	N/A
main() function definition	0	Once	1	N/A

Table 2.2.2.8: Execution time of the radiotherapy game mode with a focus on the calculate_DoseToMedium() method of the AlphaRTGame class and sections of code in it.

Description	Execution time without multiprocessing, not including the time spent waiting for user inputs: /s	Execution time with multiprocessing, not including the time spent waiting for user inputs: /s	Reduction in execution time /s
The radiotherapy game mode of the program	93.12261486	85.97500458	7.147610283
AlphaRTGame calculate_DoseToMedium()	25.68964052	17.57256708	8.117073441
AlphaRTGame calculate_DoseToMedium() MomentumTransfer_Array calculation	25.68964052	17.57256708	8.117073441
AlphaRTGame calculate_DoseToMedium() MaximumRange calculation	0	0	0
AlphaRTGame calculate_DoseToMedium() AbsorbedDose calculation	0	0	0

Even after implementing multiprocessing for calculating the dose to the medium in the radiotherapy game mode, the simulation in that mode still ran too slowly. However, unlike the beam analysis

mode, the radiotherapy game mode runs only one simulation instance. The dose that the alpha particle beam deposits into the medium increases with both the initial particle number and the initial kinetic energy of the particles (Equation 1.4.5.1). Thus, it was reasonable to implement multiprocessing for simulating a high initial number of alpha particles. There was a **for** loop in the update_AlphaParticleMomentum() method in the AlphaParticles class, which the class for the radiotherapy game inherits, that could be multiprocessed. Once multiprocessing was implemented for the simulation part of the radiotherapy game as well, a 5-run average of 28.33122854 seconds were cut off from the execution time of the radiotherapy game mode as a whole, excluding the time spent waiting for user inputs (Table 2.2.2.9).

Table 2.2.2.9: The execution times of the radiotherapy game mode with and without multiprocessing implemented for simulating the alpha particles after the multiprocessing implementation done for the calculate_DoseToMedium() method.

Description	Execution time with multiprocessing only within the calculate_DoseToMedium() method, not including the time spent waiting for user inputs: /s	Execution time with multiprocessing both within the calculate_DoseToMedium() method and for simulating the alpha particles in the beam, not including the time spent waiting for user inputs: /s	Reduction in execution time /s
The radiotherapy game mode of the program	85.97500458	57.64377604	28.33122854

3. Results

3.1. Results of the program

When the user calls the AlphaParticles2_Main.py program, the main program, using Python, they will see the text below, which is shown in blue text.

The program is loading. Please wait ...

The program has loaded.
##
Alpha Particles 2.0
##
##
Welcome to the Alpha Particles simulator program, version 2.0, for Windows 10.
It calculates the mean and maximum ranges of an alpha particle beam through a medium. It also plots the number of alpha particles in the beam as a function of distance travelled through the medium.
The program also has a radiotherapy game and a beam attenuation quiz.

Please input values for the parameters when prompted.
The higher the initial number of particles, the more time the simulation will take to complete.
You are recommended to start with a small number for the initial number of particles and gradually increase this number towards your target value while noting the computation times.
If you need to stop a simulation before it completes, press Ctrl + c while in the command terminal.

Would you like to analyse an alpha particle beam (a), play the radiotherapy game (g), or do the beam attenuation quiz (q)?

From this point, they must decide which mode of the program they wish to run. If they answer “a” or “A”, they will choose to run the beam analysis mode. In this case, they will have the option to provide further inputs via an input via or via the command terminal. If they provide them via the *InputsForBeamAnalysis.txt* input file, they will see an example of the text shown below in blue text. Note that the **y** in bold is a user input. All of the other text is printed to the command terminal.

You have chosen to analyse an alpha particle beam.

Would you like to read inputs from InputsForBeamAnalysis.txt? (y/n) **y**
Inputs
Initial kinetic energy = 1.000e-04 MeV
Initial particle number = 20
Random distribution = basic
Whether or not to seed = y
Seed for the random number generators = 0
Atomic number of the medium = 1
Atomic weight of the medium = 1 g/mol
Mass density of the medium = 1 g/cm^3

Number of simulation instances = 2
##############

If they choose to provide these inputs via the command terminal, they would see the text shown in blue below instead. Note that all of the user's inputs are shown in **bold blue text**.

You have chosen to analyse an alpha particle beam.

Would you like to read inputs from InputsForBeamAnalysis.txt? (y/n) **n**
What is the initial kinetic energy of the alpha particles, in MeV? **1e-4**

How many alpha particles are initially in the beam? **20**

Which probability density function would you like to use for the generation of random numbers for the simulation?
Options (type the name of the probability density function you wish to use):
> basic (the basic random number generator of Python's random module. It generates numbers between 0 and 1, including 0 but not 1)
> discrete (the user specifies the fineness of the discretisation. Numbers are generated with equal probability)
> triangular (ranges from 0 to 1 with a mode of 0. The probability of generating a number n between 0 and 1 decreases linearly from a maximum probability at 0 to zero probability at 1)
> uniform (includes both 0 and 1)
Choice: **basic**

Would you like to seed the random number generators using a seed you provide? (y/n) **y**
Please input an integer seed number for the random number generators of the program: **0**

You inputted a floating point number with a magnitude less than 2.2250738585072014e-308. Your inputted number will be treated as 0.0.

What is the atomic number of the medium? **1**
What is the atomic weight of the medium, in units of g/mol? **1**
What is the mass density of the medium, in units of g/cm^3? **1**

How many instances of the simulation would you like to run? **2**

A sample of the results of the simulation specified above are shown below. A plot of the number of alpha particles remaining in the beam as a function of the distance they penetrated into the medium would be received (Figure 3.1.1).

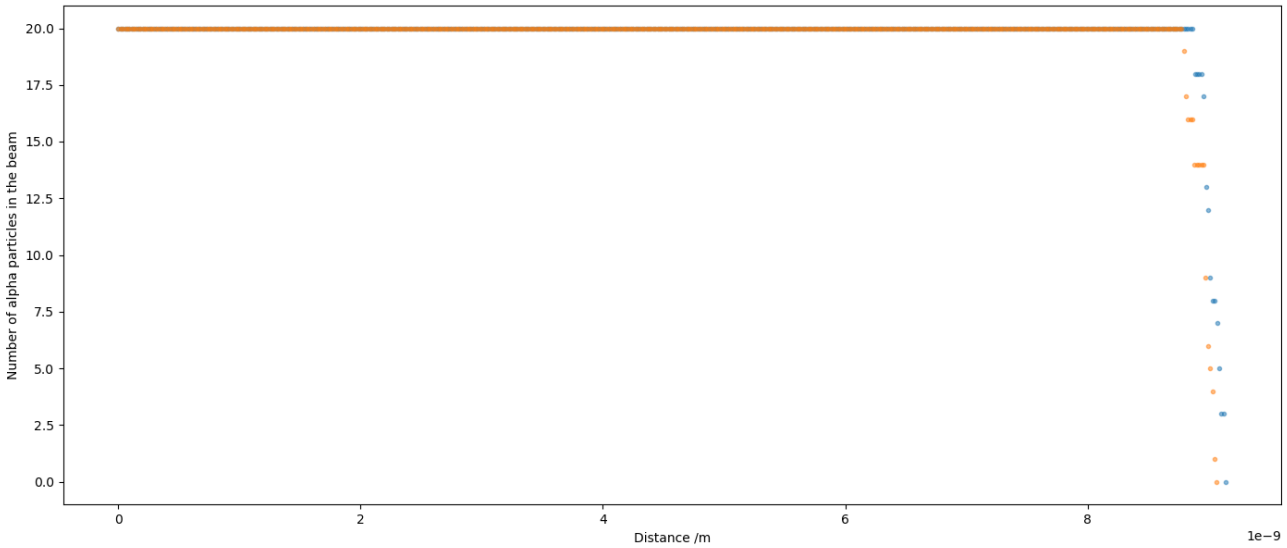

Figure 3.1.1: The number of alpha particles that remain in the beam as a function of the distance they travelled in the medium specified in the inputs shown above. Two instances of the simulation were run. The name of the file that has this plot is *AlphaRange*_[timestamp].*png*, where [timestamp] is the timestamp when the program started, formatted as YYYY-MM-DD_hh-mm-ss; for example, *AlphaRange_2020-10-25_22-24-20.png*.

The user would also receive a data file named *data*_[timestamp].*txt*, where [timestamp] is the timestamp when the program started; for example *data_2020-10-25_22-24-20.txt*. An example of its contents is shown below in blue text.

Program: AlphaParticles2.py
Author: Kyrollos Iskandar

Timestamp (YYYY-MM-DD_hh-mm-ss): 2020-10-25_22-24-20

Simulation mode: Analysis of an alpha particle beam

Inputs about the alpha particles
Initial kinetic energy = 0.0001 MeV
Initial number of alpha particles = 20
Probability density function used for random number generation: Python 3's basic distribution, where random numbers between 0 and 1 were randomly generated, and where 0 but not 1 were included in the set of numbers that could have been generated
Seed for the random number generators = 0

Inputs about the medium
Atomic number = 1.0
Atomic weight = 1.0 g/mol
Mass density = 1.0 g/cm^3

Inputs about the simulation
Number of simulation instances: 2

Results
Mean range = 8.939786622756467e-09 +/- 1.650801598152698e-10 m (Average +/- 3 * Population standard deviation)
Maximum range = 9.103910340305186e-09 +/- 1.1237401781000646e-10 m (Average +/- 3 * Population standard deviation)

Information about the randomised 3x3 matrix that was used to generate the data
# Diagonal elements:
C_xx = 0.9997257754941689
C_yy = 0.9997257754941689
C_zz = 0.9997257754941689

The values of the off-diagonal elements of the randomised 3x3 matrix are shown in the accompanying histograms.

End of the data file

They would also receive six PNGs file of histograms of the random values that each of six off-diagonal elements of RandomMatrix, the randomised 3x3 matrix that is used to update alpha particles' momentum vectors, had during the entire simulation. Examples of such histograms are Figures 2.1.3 and 2.1.4.

If the user chooses to run the radiotherapy game mode, they would see the text shown below in blue text in the command terminal if they choose to provide their inputs via an input file. If they choose to provide their inputs via the command terminal, they would see text similar to the text shown above for the beam analysis mode in the case where they must provide their inputs via the command terminal.

Would you like to analyse an alpha particle beam (a), play the radiotherapy game (g), or do the beam attenuation quiz (q)? **g**
You have chosen to play the radiotherapy game.

You must deposit 0.08 Gy of radiation dose into tissue.
The tissue has the following characteristics:
> Effective atomic number = 7.0437959999999995
> Atomic weight = 13.98812810888 g/mol
> Mass density = 1.02 g/cm^3

Goal: Define an alpha particle beam that can deliver 0.08 Gy of radiation dose to the tissue.
> Note: The beam's cross-sectional area is square-shaped.

Would you like to read inputs from InputsForRTGame.txt? (y/n) **y**
Inputs
Initial kinetic energy = 1.000e-02 MeV
Initial particle number = 50
Random distribution = basic
Seed for the random number generators = 0

Beam height = 1.0 cm
Beam width = 1.0 cm
Atomic number of the medium = 7.0437959999999995
Atomic weight of the medium = 13.98812810888 g/mol
Mass density of the medium = 1.02 g/cm^3
Number of simulation instances = 1
##############

After the simulation, the user would see an example of the text shown below in blue text in the command terminal. Note that no files are exported in this mode except for the CSV files of the results of the timing experiments of the code sections used for running that mode.

Analysing the data ...
Absorbed dose = 0.08054784408849448 Gy
Well done! A dose between 0.076 and 0.084 Gy was delivered.

Execution time of the program minus the time spent waiting for user inputs = 63.92450428009033 s

The simulation is done! Please check the outputted files. They are in a folder named Data_2020-10-25_22-38-07/ in the same directory as the program file.

If the user chooses to run the attenuation quiz mode, they would see an example of the text shown below in blue text in the command terminal if they choose to provide their inputs via an input file. If they choose to provide their inputs via the command terminal, they would see text similar to the text shown above for the beam analysis mode in the case where they must provide their inputs via the command terminal.

Would you like to analyse an alpha particle beam (a), play the radiotherapy game (g), or do the beam attenuation quiz (q)? **q**
You have chosen to do the beam attenuation quiz.

In this quiz, an alpha particle beam of a random energy will be generated.
Also, a medium and a beam transmission fraction will be randomly chosen for you to consider.
Your task is to predict the thickness(es) of the material(s) that will transmit the aforementioned fraction of the beam. A material attenuates the beam more if it is thicker.
You may specify the number of materials you would like to consider, but the program will select them.
How many media would you like to the quiz to consider? **3**

Chosen material 1: Silver metal
Atomic number 1 = 47
Atomic weight 1 = 107.8682 g/mol
Mass density 1 = 10.5 g/cm^3

Chosen material 2: Beryllium metal
Atomic number 2 = 4

Atomic weight 2 = 9.012182000000001 g/mol
Mass density 2 = 1.848 g/cm^3

Chosen material 3: Lithium metal
Atomic number 3 = 3
Atomic weight 3 = 6.941 g/mol
Mass density 3 = 0.534 g/cm^3

What thickness(es) of the chosen material(s), *together*, in m, will transmit 1.4000000000000001% of a beam of 100 alpha particles of energy 4.087e-05 MeV?
Note that the thickness of the last material you specify will be extended if it is too thin.
Also note that you may add more materials to the 'AttenuationQuiz_MaterialLibrary.csv' file if you want the quiz to possibly consider other materials. However, you must adhere to the format of that file.

Would you like to read material thicknesses from the input file named 'InputsForAttenuationQuiz.txt'? (y/n) **y**

You have chosen to let the program read the material thicknesses from a file.
The program will read from a file named 'InputsForAttenuationQuiz.txt' in the same directory as the script file of this program. Please make sure it is formatted as shown by example below ...
Ignore this line: ########## Input file for the attenuation quiz ##########
Medium 1 thickness = 1e-9 m
Medium 2 thickness = 1e-9 m
Medium 3 thickness = 1e-9 m
Medium 4 thickness = 1e-9 m
Medium 5 thickness = 1e-9 m
Medium 6 thickness = 1e-9 m
Medium 7 thickness = 1e-9 m
Medium 8 thickness = 1e-9 m
Medium 9 thickness = 1e-9 m
Medium 10 thickness = 1e-9 m
Ignore this line: #### End of Input file for the attenuation quiz #####

The only contents of the file you are permitted to change are *the numbers* at the right side of the '=' signs. This excludes the units.
You may also add more 'Medium [number] thickness ...' lines if you wish to. Please make sure the numbers are ordered and each line is formatted as shown above.
When you are ready for the program to read the input file, please say so ...
Are you ready for the program to read the input file? Do not answer if you are not yet ready. (y)
y
The program will now read as many of your inputs from the input file named 'InputsForAttenuationQuiz.txt' as the number of media you said you wanted to consider ...

The program has read the following thicknesses and replaced invalid inputs with valid ones where appropriate ...
Medium 1 thickness = 1e-9 m
Medium 2 thickness = 1e-8 m
Medium 3 thickness = 2e-9 m

Let us test your prediction ...

After the simulation, the user would see an example of the text shown below in blue text in the command terminal.

The simulation has ended.

Results

You predicted incorrectly. Too few alpha particles would be transmitted! With the given media, a total thickness of 3.462e-09 m was required. Please try again.

Here is one combination of correct material thicknesses ...

Required thickness of Medium 1 = 2.663e-10 m

Required thickness of Medium 2 = 2.663e-09 m

Required thickness of Medium 3 = 5.326e-10 m

Please check Data_2020-10-25_22-43-25/ for a plot of the number of alpha particle remaining in the beam as a function of distance penetrated through the medium.

The simulation is done! Please check the outputted files. They are in a folder named Data_2020-10-25_22-43-25/ in the same directory as the program file.

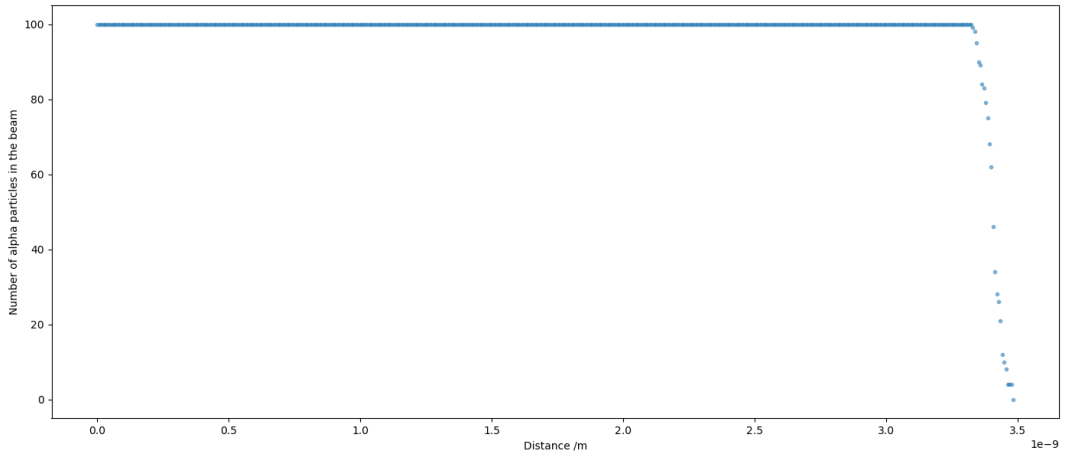

Figure 3.1.2: An example of the plot of the number of alpha particles remaining in the beam as a function of the distance they penetrated into the media from their starting x-position that is exported after the attenuation quiz simulation. The PNG file of the plot has a name similar to that shown in Figure 3.1.1.

4. Discussion

4.1. Bugs and other abnormal behaviour of the program

The user should not interrupt the program using **KeyboardInterrupt** while one or more multiprocessing pool processes are running. On Windows, this is doable by pressing **Ctrl + c** on the keyboard, and doing this would not stop the program but will cause the command terminal to be somewhat in a state of confusion. If the user does this, they must then close the command terminal and reopen it.

Also, timing measurements of code that is executed within multiprocessing.Pool() pooled processes are not taken because the **__name__** attribute is not **"__main__"**, but rather **"__mp_main__"** within the pooled processes. The timing measurements are taken only when **__name__** is **"__main__"**, which is when the program is run directly from the command terminal. The consequence of this is that the TimingAdmin program cannot analyse those measurements as they are not taken in the first place. In the serial version of the program, that is, before multiprocessing was implemented, all of the timing measurements could be taken.

Also, while the cross-section σ, that is, the probability that a particular alpha particle will interact with an electron in the medium, depends on the alpha particle's kinetic energy, the expression for the cross-section that is used in the program (Equation 4.1.1) is not correct. It does not even have the correct units. The correct units are units of length squared. However, it is a better approximation than the expression that was used before it (Equation 4.1.2) because the latter expression led to mean and maximum ranges of hundreds or thousands of metres being calculated, which is not possible for alpha particles of the kinetic energies that were specified.

$$\sigma = \frac{2\pi m_\alpha r_e^2}{p_\alpha^2}$$

(Equation 4.1.1)

$$\sigma = \pi r_e^2$$

(Equation 4.1.2)

where $m_\alpha = 6.64465723 \times 10^{-21}$ kg is the mass of one alpha particle, $r_e = 2.8179403227 \times 10^{-15}$ m is the classical electron radius and p_α is the momentum magnitude of the alpha particle in units of kg m s^{-1} [3, 9].

4.2. Further work

The program has the potential to give more accurate results if relativistic mechanics is implemented, especially when the initial kinetic energy of the alpha particles is a few MeV; 10 MeV, for example. However, implementing relativistic mechanics requires an implementation of velocity and time and a redefinition of Equations 2.1.1 to 2.1.5, 2.1.7 to 2.1.8 and 2.1.10 to 2.1.16(a) – (c). For example, the relativistic expression of the kinetic energy of an alpha particle is

$$E_K = (\gamma - 1)m_{0\alpha}c^2$$

(Equation 4.2.1)

where γ, the Lorentz factor, is

$$\gamma = \left(1 - \frac{v_\alpha^2}{c^2}\right)^{-\frac{1}{2}}$$

(Equation 4.2.2)

where $m_{0\alpha} = 6.64465723 \text{ x } 10^{-21}$ kg is the mass of an alpha particle at zero velocity, $c = 2.99792458 \text{ x } 10^8$ m s^{-1} is the speed of light in vacuum, and v_α is the velocity of the alpha particle [9, 24]. This means that the approach to the simulation of alpha particles travelling through a medium must be reformulated before being implemented as code.

Also, a correct expression for the cross-section of alpha particles is yet to be found and implemented.

Additionally, there are still sections of the script file that are yet to be multiprocessed such as the simulation of the attenuation quiz. This simulation may run faster if multiprocessing is implemented for simulating the alpha particles just as the multiprocessing for simulating alpha particles in the radiotherapy game simulation (Section 2.2.2).

Also, the following optimisations can be done:

- Let the simulation for the attenuation quiz end once there are less alpha particles in the beam than the number specified by the transmission fraction. This means that the simulation does not need to run for longer than it needs to, thus reducing the execution time of the simulation.
- Let the mean range of an alpha particle beam be determined by looking only at the last 1/3 of the distance values in the list of distances to be checked considering that the rapid descent in the number of alpha particles in the beam is always in that section of the plot (Figure 3.1.1).

Finally, some work may go into implementing a graphical user interface (GUI) for the program to make it somewhat easier to use.

5. Conclusion

The Alpha Particles program, version 2.0, simulates a beam of alpha particles travelling through one or more media and can even statistically analyse the penetration distance of the beam. The program has potential for improvement in terms of the accuracy of its results as well as further optimisations and multiprocessing.

6. References

1. Iskandar, K., *The Alpha Particles program version 1.0*. 2020, 124 La Trobe St, Melbourne VIC 3000: RMIT University.
2. Krane, K.S., *Introductory Nuclear Physics*. Reprint ed. 2014, Durga Printo Graphics, Delhi: John Wiley & Sons, Inc.
3. National Institute of Standards and Technology. *Atomic Weights and Isotopic Compositions for All Elements*. [cited 2020 2 April 2020]; Available from: https://physics.nist.gov/cgi-bin/Compositions/stand_alone.pl?ele=&ascii=html&isotype=all.
4. Fayek, H., *Week 6 Object-Oriented Programming*. 2020, Haytham Fayek: 124 La Trobe St, Melbourne VIC 3000.
5. Johnston, P., et al., *Lecture 2: Fundamentals and extensions to radiation transport*, in *PHYS2139 – Radiotherapy Physics and Modelling*. 2020: RMIT University.
6. Fayek, H., *Week 3 Control Flow*. 2020, Haytham Fayek: 124 La Trobe St, Melbourne VIC 3000.
7. The SciPy community. *numpy.ndarray.max*. 2020 29 June 2020; Available from: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.max.html.
8. Python Software Foundation. *time — Time access and conversions*. 2020 18 October 2020; Available from: https://docs.python.org/3/library/time.html.
9. *CRC handbook of chemistry and physics : a ready-reference book of chemical and physical data*. 97 ed, ed. W.M. Haynes, D.R. Lide, and T.J. Bruno. 2017, 6000 Broken Sound Parkway NW, Suite 300 Boca Raton, FL 33487-2742: Taylor & Francis Group, LLC.
10. Python Software Foundation. *random — Generate pseudo-random numbers*. 2020 11 September 2020; Available from: https://docs.python.org/3/library/random.html.
11. Python Software Foundation. *math — Mathematical functions*. 2020 28 September 2020; Available from: https://docs.python.org/3/library/math.html.
12. Python Software Foundation. *statistics — Mathematical statistics functions*. 2020 14 October 2020; Available from: https://docs.python.org/3/library/statistics.html.
13. Python Software Foundation. *datetime — Basic date and time types*. 2020 17 September 2020; Available from: https://docs.python.org/3/library/datetime.html.
14. Python Software Foundation. *os — Miscellaneous operating system interfaces*. 2020 17 September 2020; Available from: https://docs.python.org/3/library/os.html.
15. Python Software Foundation. *sys — System-specific parameters and functions*. 2020 18 September 2020; Available from: https://docs.python.org/3/library/sys.html.
16. Python Software Foundation. *multiprocessing — Process-based parallelism*. 2020 19 October 2020; Available from: https://docs.python.org/3/library/multiprocessing.html.
17. NumPy. *NumPy v1.19.0*. 2020; Available from: https://numpy.org/.
18. Hunter, J., et al. *Matplotlib: Visualization with Python*. 15 September 2020 [cited 2020; Available from: https://matplotlib.org/.
19. the pandas development team. *pandas*. pandas 2020 [cited 2020; Available from: https://pandas.pydata.org/.
20. Python Software Foundation. *psutil 5.7.2*. 2020 15 July 2020 [cited 2020 23 October 2020]; Available from: https://pypi.org/project/psutil/.
21. Programiz. *Python Exception Handling Using try, except and finally statement*. Available from: https://www.programiz.com/python-programming/exception-handling.
22. W3Schools. *Python assert Keyword*. 2020 [cited 2020; Available from: https://www.w3schools.com/python/ref_keyword_assert.asp.
23. the pandas development team. *pandas.DataFrame.fillna*. pandas 2014 [cited 2020; Available from: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html.
24. Halliday, D., R. Resnick, and J. Walker, *Fundamentals of Physics: Extended*. 7th ed. 2005, 111 River Street, Hoboken, NJ 07030-5774: John Wiley & Sons, Inc.