Year 13 Programming Project– Momentum Collision Simulation

Contents:

Analysis:

Problem Identification

The purpose of the program I plan on creating is to simulate collisions between two objects on a flat surface to show people how momentum effects different types of collisions. It will be a momentum collision simulator written in Python 3.3.2 that allows a user to input the mass and velocity of different objects then receive an output for the total momentum, change in kinetic energy and the speed after the collision. Users will have the option for different types of collisions and an animation of the collision will be shown. This will all be displayed on a graphical user interface using Pygame 1.9.2a0. The program will work on Windows 7. The program will also output each step it took in working out the solutions so people understand how it works.

Currently there are no momentum collision simulations that store inputs and outputs then create a graph. This would be useful for comparing two different variables and checking if there is a relationship between them. For my program, I will store the data in a CSV file using Excel 2016 and will decide on which graphs are appropriate to display after further research.

This is a suitable as programming project because: it has multiple suitable stakeholders, is complex since it requires storing data, use of libraries and complex algorithms and it is solvable by computational methods.

Computational Methods

Why this project is suited to a computational solution:

My program is amenable to a computational approach because the problems it presents will be solved by constructing a series of mathematical algorithms. Together these algorithms will work out the final speed of objects after a collision by calculating the total momentum and change in kinetic energy using a combination of the user inputs (mass and speed) and mathematical formulas. Aspects of the solution that will require a computer to run are the GUI and storing the data entered to automatically create a graph. To make creating these algorithms and the rest of my program easier and more efficient I will use a range of computational methods.

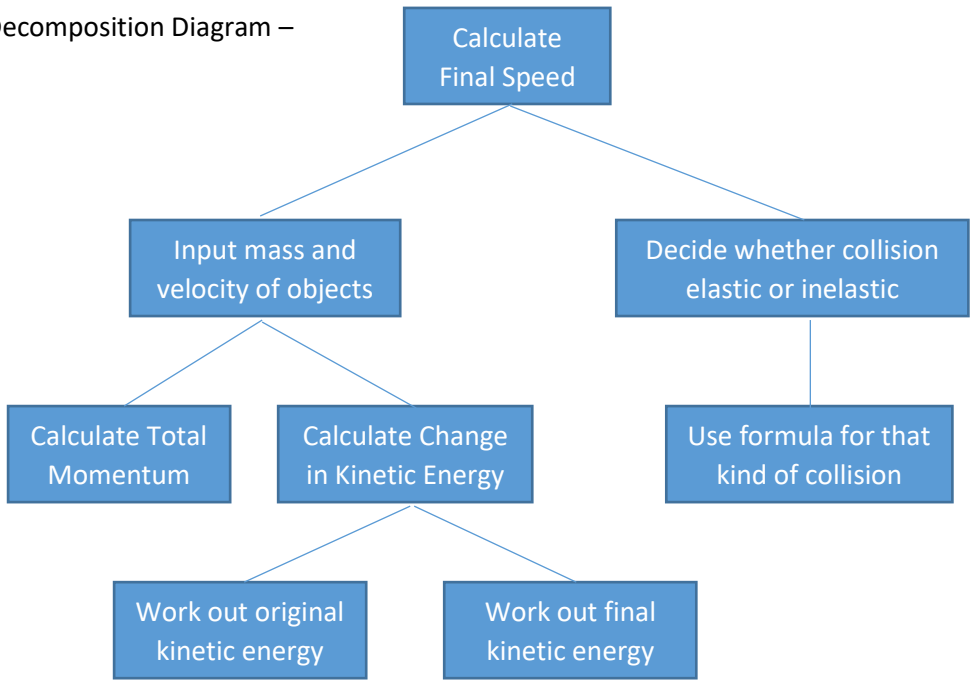
Computational methods the solution will use:

- Problem recognition – The main problem my project presents is writing algorithms to calculate the momentum, final speed and change in kinetic energy which makes it soluble by computational methods. I will solve these problems using different techniques such as problem abstraction and the divide and conquer approach.

- Abstraction – In order to simplify momentum collisions to a GCSE level I will remove unnecessary details such as friction between the objects and the ground and the objects will all travel at constant speed ignoring acceleration. The objects will be represented by the same image even if the weight increases since students will be able to understand what is happening without the size of the object changing to correspond with the weight. The background of the collision will be plain so it does not distract users from the collision taking place.

- Problem Decomposition – I will break down my problem into three main parts and then each of these parts into smaller steps because this will make it easier to solve and for other people to understand my development process. The first part will be writing classes using object oriented programming to work out the total momentum, change in kinetic energy and the speed after the collision with the inputs given. The second part will be writing a program to store the inputs and calculations in a file then use them to create a graph. The third part will be creating the graphical user interface using pygame to display the collision and outputs.

- Decomposition Diagram –



- Divide and Conquer – After completing these smaller, simpler steps I will then combine them into a complete, more complex solution in order to achieve the programs purpose.

- Heuristic Methods – In order to complete the project in the time given I will be creating a simpler version of a real collision by ignoring factors such us friction, acceleration and air resistance. The program is still sufficient for the purpose required so although not a perfectly accurate simulation it is still an adequate solution because the students using it do not need to consider these factors for GCSE and ignoring this speeds up the development process.

- Backtracking – While developing my final solutions I will interview my stakeholder to see if they have any suggestions, it is possible they will ask me to do something that requires me to try multiple options until I find a solution that is right for them.

- Visualisation – Before developing my algorithms, I will visualise them using a flow chart since this will make it easier for me to understand what they need to do which will allow me to program them more efficiently. I will also visualise my GUI before coding it by drawing diagrams of the layout allowing me to choose the correct layout before programming it; this will save time because it is easier to edit a diagram than code.

Stakeholders

A momentum collision simulator would benefit people teaching GCSE maths or physics to students because it could be used as a classroom tool to enable the students to better understand what happens during different kinds of collisions. Students could also use it to check the answers to homework questions or when revising since a teacher may not be available to help and the simulator will display a systematic guide on how it worked out the answers. For this reason, teachers could also send the simulator to students that have missed a lesson on momentum so they can catch up on the work they missed without attending an extra lesson since they can achieve a basic understanding of momentum collisions from the simulation.

Another demographic of this software would be people learning to code using python and pygame or programming teachers because the program could be used as an exemplar or broken down into a step-by-step tutorial. Users would need to already have some experience coding since it is too complex for beginners. These stakeholders would require the program to be easy for other people to understand which could be done by giving things suitable variable names, commenting on the code to explain individual parts and by putting the code in chronological order (linear) where possible.

I decided Mr Hutchinson, an A-level and GCSE physics teacher will be my main stakeholder because I feel my program is most suited to his needs and he will be able to give me useful advice regarding how I can develop my program in a way that will help students maximise their understanding of momentum. I will interview him to find out what these potential requirements are.

Interview

Questions:

Before interviewing Mr Hutchinson, I told him I plan to make a momentum collision simulation that teachers can use as a demonstration for GCSE physics students to understand how momentum works. I also mentioned data inputted and calculated will be stored to create a graph then asked him these questions:

1. Have you used simulations in lessons as a demonstration for students?
2. What aspects of the simulations you have used did you like?
3. What were the main issues with these simulations?
4. Would a momentum collision simulation be useful to students studying GCSE physics or maths? Why?
5. Which types of collisions would you like to be included in the simulation?
6. How would you like the objects colliding to be represented?
7. What information about the collisions would you like to be displayed?
8. What else would you like to see in a momentum collision simulator?
9. Is there anything else you would like to add?

Questions 1, two and three investigate what kind of simulations are currently used in physics lessons and which aspects of these examples are liked and disliked. This will allow me to create a program that includes features I know will be useful and help me to avoid adding unnecessary functions.

Questions 4 will make sure that my program is suited to its purpose and explains what GCSE students need from this simulation. This will ensure I do not make a program that is too complex or too simple for their educational needs.

Questions 5 to 8 allow my stakeholder to be more precise in the exact features he wants to be included which is important since as a teacher he has first-hand experience in what helps students to understand concepts and how my simulation can assist them.

Questions 9 will allow Mr Hutchinson to include any ideas he has for my project that I did not think of myself.

Answers:

1. Have you used simulations in lessons as a demonstration for students?

"Yes we use them for year 8 through to year 13"

2. What aspects of the simulations you have used did you like?

"The opportunity to vary conditions. Also those with big clear, projectable images are best. Students need to see the collision happen as much as see data varying."

3. What were the main issues with these simulations?

"Some were too slow or had poor graphics. Others were hard to manipulate and understand"

4. Would a momentum collision simulation be useful to students studying GCSE physics or maths? Why?

"Yes, Momentum is one of the toughest sections of GCSE Science as it is about vector quantities and it is not energy based."

5. Which types of collisions would you like to be included in the simulation?

"Linear only, head on. Also ability for objects to lock together becoming essentially one object."

6. How would you like the objects colliding to be represented?

"Simple rectangles would be acceptable. Nothing that would distract from the collision"

7. What information about the collisions would you like to be displayed?

"Vector if possible showing magnitude and direction of each object and the total momentum"

8. What else would you like to see in a momentum collision simulator?

"Maybe a quiz – multiple choice even to test students' knowledge"

9. Is there anything else you would like to add?

"Graph based output is not essential as total momentum never varies. However, showing a large positive and a small negative momentum object becoming one object of smaller positive momentum might work"

Analysis:

Interviewing my stakeholder has confirmed my project is required because Mr Hutchinson said himself and other teachers frequently use simulations but he also mentioned multiple issues the current simulations have. It is clear he wants my GUI to have "clear, projectable images" that are

very simple or just rectangles because one of the issues he has with similar programs is that the graphics are poor or too slow and distract from the important information which is the “data varying”.

From the interview, I now know GCSE students will only need to see elastic collisions taking place on flat surfaces however, inelastic collisions (“objects to lock together becoming essentially one object”) will also be useful and a vector above each object showing its magnitude and direction.

Mr Hutchinson also suggested I include a multiple-choice quiz instead of a graph however; this is too simple since it is not amenable to a computational approach whereas a graph is much more complex. For this reason, I will ignore this particular piece of advice.

Research

GCSE Momentum:

My stakeholder requested that I do a momentum collision simulation for GCSE physics students since I have completed my GCSEs however, the syllabus has changed so I must find out what current students need to learn so I may adapt my project to their needs.

On the AQA website, I found this learning specification for momentum:

4.5.7.1 Momentum is a property of moving objects

Content	Key opportunities for skills development
Momentum is defined by the equation: momentum = mass × velocity $[p = m v]$ momentum, p, in kilograms metre per second, kg m/s mass, m, in kilograms, kg velocity, v, in metres per second, m/s	WS 1.2 MS 3b, c Students should be able to recall and apply this equation.

4.5.7.2 Conservation of momentum

Content	Key opportunities for skills development
In a closed system, the total momentum before an event is equal to the total momentum after the event. This is called conservation of momentum. Students should be able to use the concept of momentum as a model to: • describe and explain examples of momentum in an event, such as a collision • (physics only) complete calculations involving an event, such as the collision of two objects.	AT 1, 2, 3 Investigate collisions between laboratory trollies using light gates, data loggers or ticker timers to measure and record data.

https://filestore.aqa.org.uk/resources/physics/specifications/AQA-8463-SP-2016.PDF

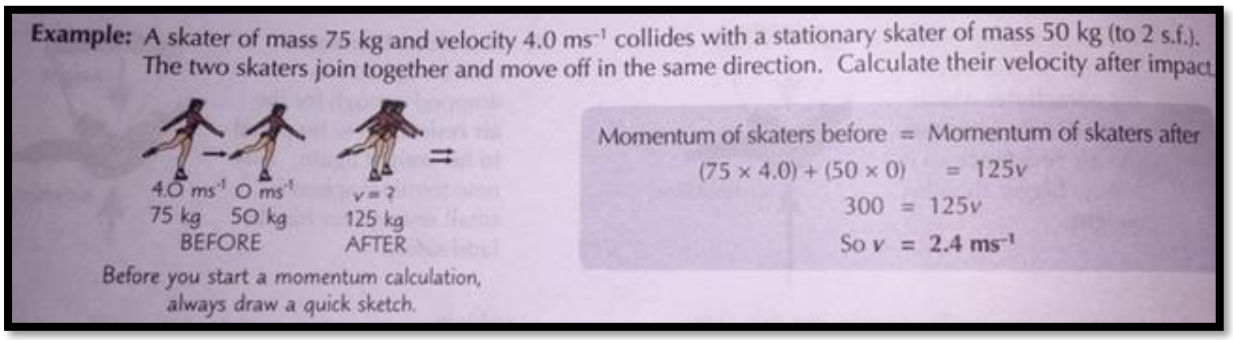
The concept of my project already covers the "Key opportunities for skills development" section but I will also need to make sure my success criteria considers the content section of this specification. On the same website I found out that, it is standard for GCSE physics students to round their calculations to two decimal places unless told otherwise so this is what my simulation will do.

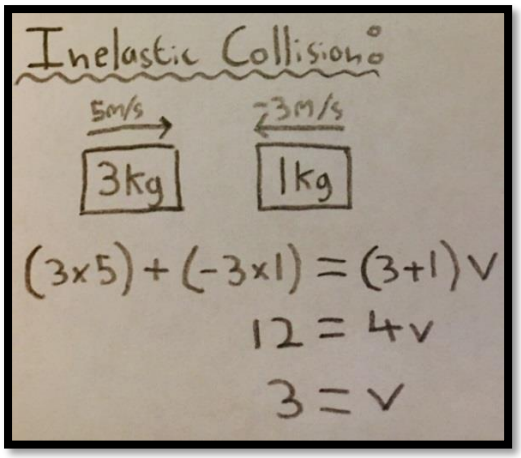
Collision Types and Equations:

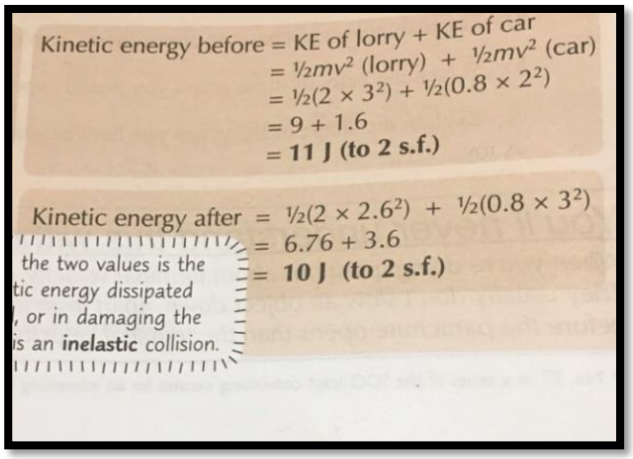
Mr Hutchinson asked me to include elastic and inelastic collisions so I need to research the difference between these collisions and how to work out their final velocities. In the GCP GCSE Physics Revision Guide, I found these definitions:

- Elastic Collision- momentum and kinetic energy are conserved
- Inelastic Collision- momentum is conserved, some kinetic energy transferred to other forms
- Momentum = mass x velocity
- Kinetic Energy = ½ x mass x velocity ^{2} = (momentum ^{2}) / (2 x mass)
- Kinetic Energy lost = Kinetic Energy before – Kinetic Energy after

The textbook also stated "in inelastic collisions the objects collide then move together at the same speed whereas in elastic collisions the objects bounce off of each other". The book also included an example of how to work out the final velocity in an inelastic collision, which I then practiced myself to help me better understand how to write my algorithm:







The method for elastic collisions was not in the textbook so I found a website that gave me this formula for final velocities:

$$v_{f1} = [(m_1 - m_2) \cdot v_{i1} + 2\, m_2 \cdot v_{i2}] / (m_1 + m_2)$$

$$v_{f2} = [2\, m_1 \cdot v_{i1} - (m_1 - m_2) \cdot v_{i2}] / (m_1 + m_2)$$

http://convertalot.com/elastic_collision_calculator.html

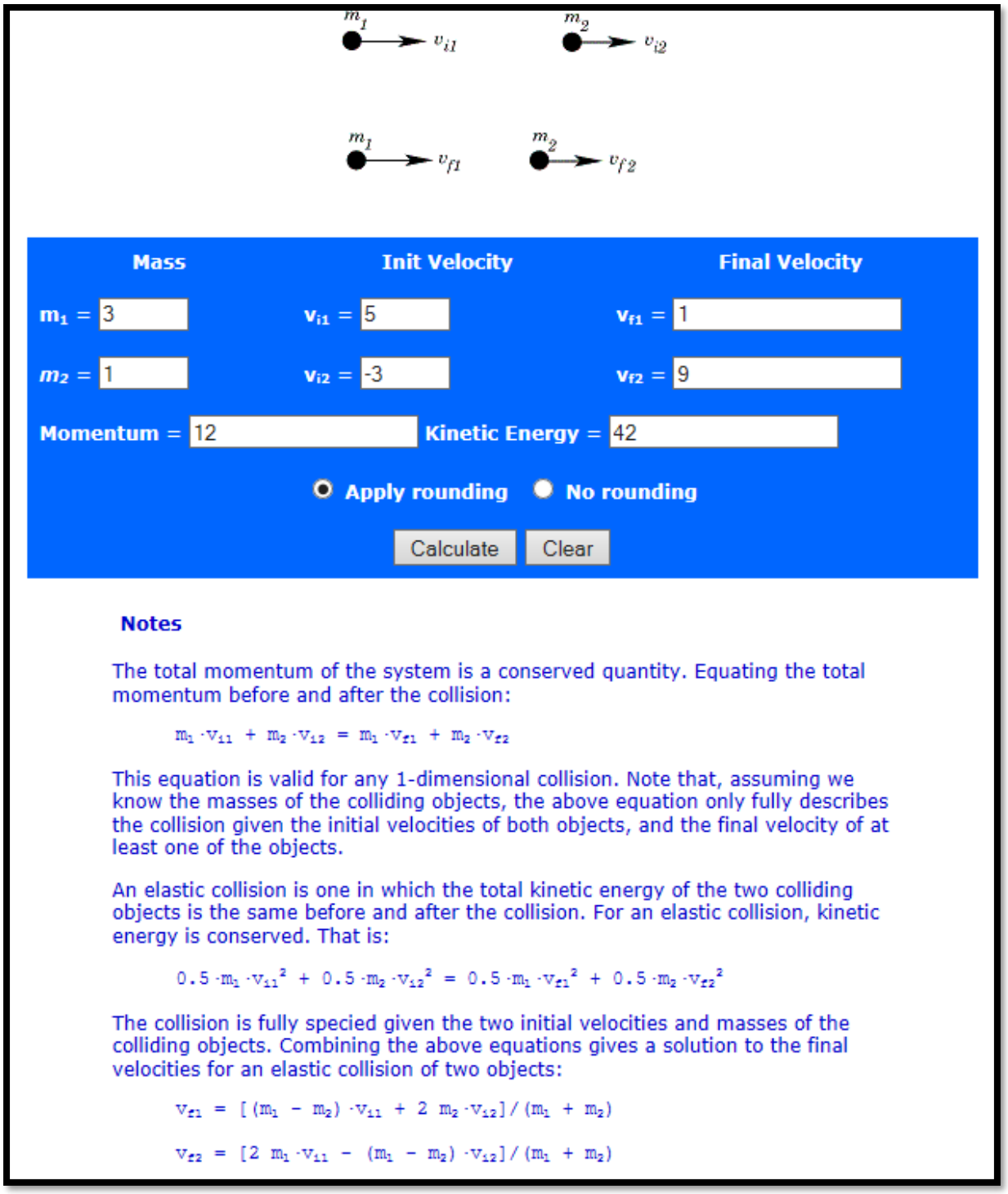
Simulations:

To ensure that the project I create is a momentum collision simulation I needed to find out what features a program needs to be considered a simulation. On Cambridge dictionary, I found this definition:

https://dictionary.cambridge.org/dictionary/english/simulation

Example 1:

To continue my research I looked at existing momentum collision simulations to see the approaches other people took and to get ideas for my own project. First, I found an elastic collision calculator on convertalot.com. On this website, you enter the mass and speed of two different objects that are represented by particles. The site then displays the momentum and kinetic energy.



http://convertalot.com/elastic_collision_calculator.html

Features I will use:

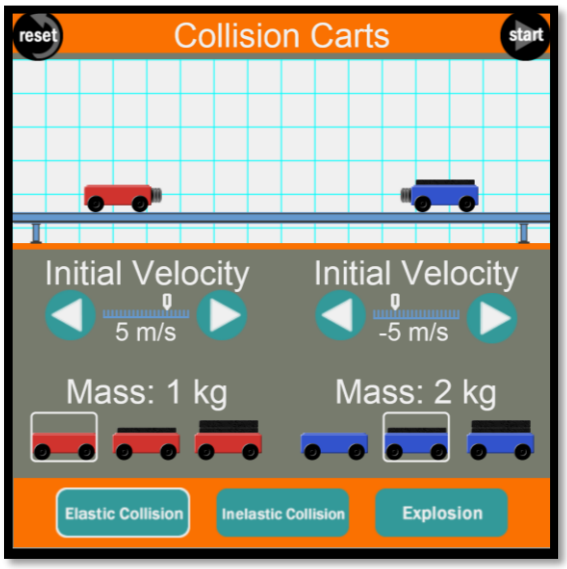
- Boxes where any values of mass and velocity can be inputted because if students are using it to check their calculations they will need to be able to enter specific values.
- Clear button to clear all the boxes because my stakeholder mentioned many other simulations are hard to manipulate and this feature will make it easier/quicker to do so.
- Kg and m/s as the units for mass and velocity because these are the standard SI units used in physics and students using the simulation will be able to convert other units into these ones.
- GUI displaying the change in kinetic energy because this will help users to understand the difference between elastic and kinetic collisions.
- GUI displaying total momentum because the total momentum is used to work out the final velocity so it will help users understand the calculations made.
- Explanation of how the final speed was calculated because students that missed a lesson on momentum are one of my stakeholders and they will need to understand how the outputs are calculated using the values inputted.

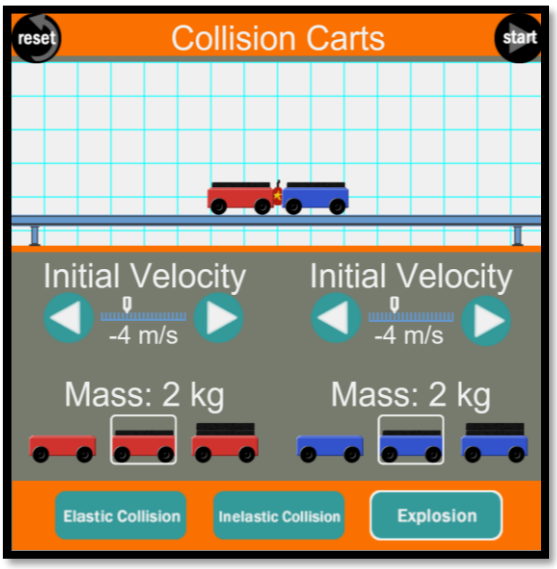
Features I will not use:

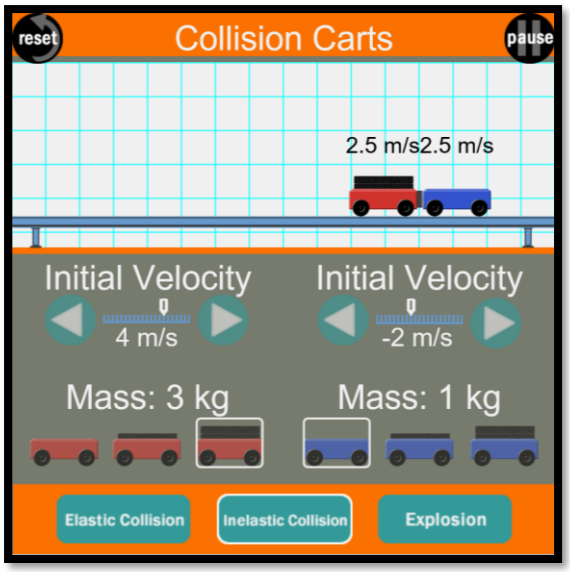
- Particles and still images to represent the objects colliding since without this my program could not be considered a simulation and users may find it more difficult to understand what is happening.
- A 'no rounding' option because I feel this feature would not help my program achieve its purpose so is unnecessary since students will just have to round the answer anyway.
- This example only calculates elastic collisions whereas I will also have an option for inelastic collisions because my stakeholder requested this option and it will make my project more complex.

Example 2:

Next, I looked at a collision cart simulator on www.physicsclassroom.com. This website is more useful than my previous example because it has the same target audience and includes more of the features I intended on including.







https://www.physicsclassroom.com/Physics-Interactives/Momentum-and-Collisions/Collision-Carts/Collision-Carts-Interactive

Features I will use:

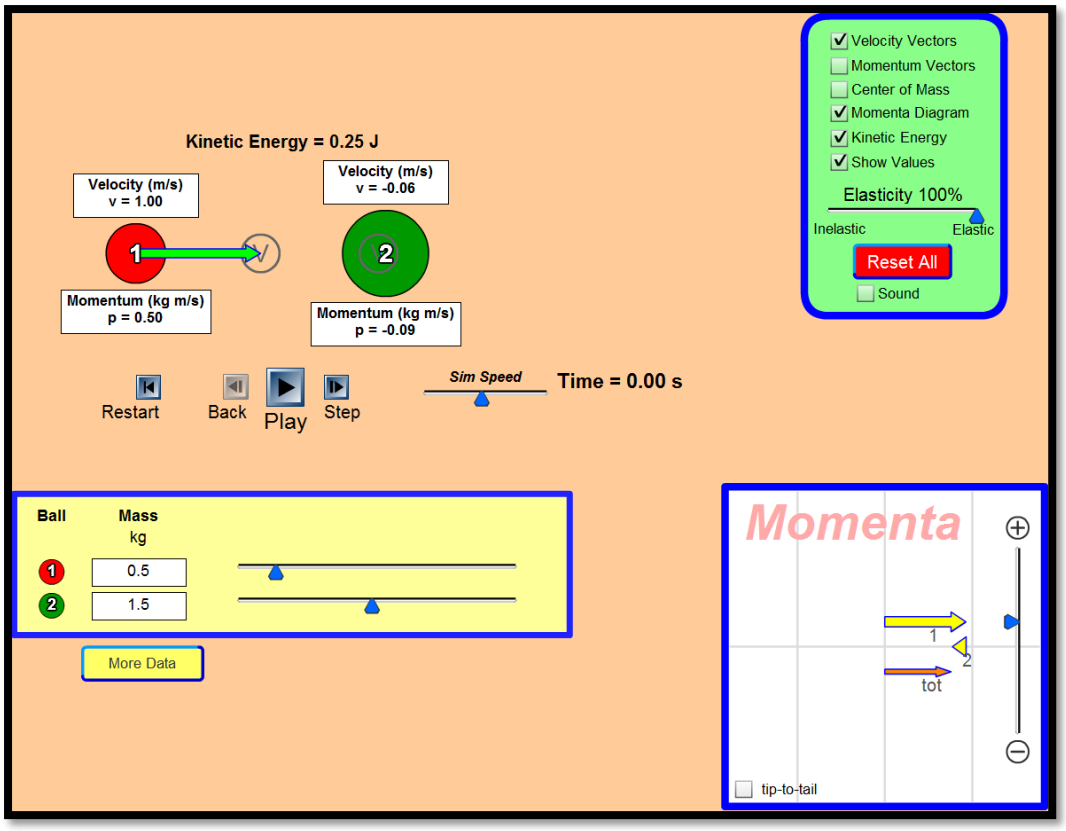
- Start and reset buttons because there needs to be a way to start the animation and because my stakeholder mentioned many other simulations are hard to manipulate and this feature will make it easy/quicker to do so.
- Simple, rectangular carts to represent the objects colliding because this is what my stakeholder requested so as not to distract from the varying data.
- GUI enabling users to change mass and initial velocity because this will make it more user friendly and allow people to simulate the collisions of different objects.
- Option for elastic or inelastic collisions because my program is intended to be used for educational purposes and this will demonstrate the difference between elastic and inelastic collisions.
- Kg and m/s as the units for mass and velocity because these are the standard SI units used in physics and students using the simulation will be able to convert other units into these ones.

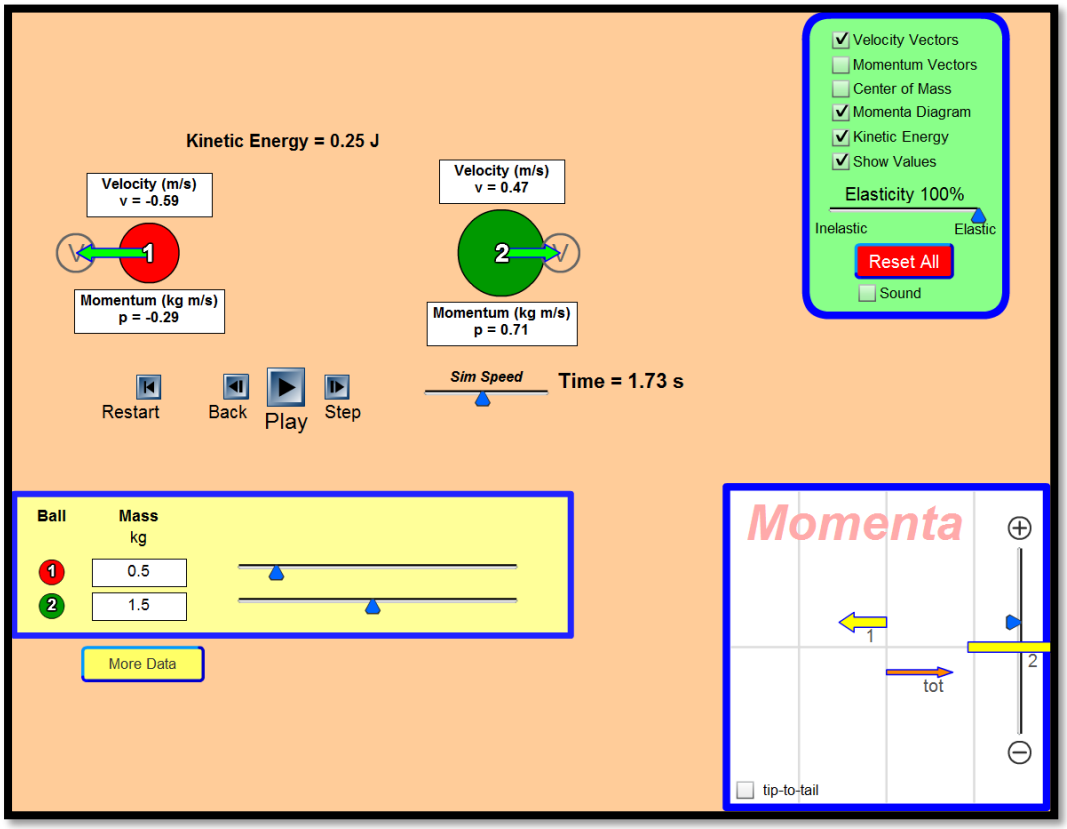
Features I will not use:

- Option for explosion collision because GCSE students do not need to learn about this and the calculation process is very similar to inelastic collisions so it would not add any complexity to my program.
- Pause button because I feel this is an unnecessary feature, as the objects will move slowly enough to see the collision and the varying data.
- Very limited ranges for mass and velocity because if students are using it to check their calculations they will need to be able to enter specific values.

Example 3:

This simulation is more complex than my previous examples. I decided to use it because it has a lot more options than example 1 and 2 so it may give me more ideas on what I should include and demonstrates some features I am planning to include that I have not yet seen examples of.





https://phet.colorado.edu/sims/collision-lab/collision-lab_en.html

Features I will use:

- A graph however, mine will be a graph of kinetic energy against time because this will show the difference between elastic and inelastic collisions.
- A timer because this will be needed to create a kinetic energy against time graph.
- Start and reset buttons because there needs to be a way to start the animation and because my stakeholder mentioned many other simulations are hard to manipulate and this feature will make it easy/quicker to do so.
- Option for elastic or inelastic collisions because my program is intended to be used for educational purposes and this will demonstrate the difference between elastic and inelastic collisions.
- Kg and m/s as the units for mass and velocity because these are the standard SI units used in physics and students using the simulation will be able to convert other units into these ones.
- Boxes where any values of mass and velocity can be inputted because if students are using it to check their calculations they will need to be able to enter specific values.
- Clear button to clear all the boxes because my stakeholder mentioned many other simulations are hard to manipulate and this feature will make it easy/quicker to do so.
- GUI displaying the change in kinetic energy because this will help users to understand the difference between elastic and kinetic collisions.

Features I will not use:

- Option to change the speed of the simulation because the objects will move slowly enough to see the collision and the varying data.
- Sound because this will not add anything to the simulation that makes it more educational.
- Displaying the centre of mass because this will not add anything to the simulation that makes collisions easier to understand.
- Slider to determine the elasticity because GCSE students only need to learn about completely elastic or inelastic collisions nothing in-between.
- Display options because my simulation will not include as many features as this example so does not require options.

Essential Features

Through interviewing my stakeholder and doing my own research I have concluded that these are the appropriate features for my program:

1. A way to input the mass and velocity of two different objects.
2. Options for elastic and inelastic collisions.
3. A graphical user interface with start, reset and clear buttons.
4. A graphical user interface that displays two objects with velocity vectors showing their magnitude and directions.
5. A way to display the change in kinetic energy and the total momentum.

6. A timer timing the collision.
7. A database that stores the data entered and values calculated.
8. A GUI displaying a graph of kinetic energy against time.
9. A way to end the program

Description and Justification of these essential features:

1. This will probably be done using an entry box on a GUI. Inputs of mass and velocity are needed because momentum is calculated by doing mass x velocity and this is a momentum collision simulation.
2. My stakeholder requested an option for elastic and inelastic collisions because GCSE students need to know the difference between them and how to calculate the final velocities for both these collision types. It will also add complexity to my project.
3. This is because my stakeholder mentioned the issue with many other simulations he has used is that they are difficult to manipulate but these features will make it easier to do so.
4. The vectors shall probably be represented by arrows above the object they apply to and they will help students using the simulation to understand what is happening since it will make the varying data and direction of movement clearer.
5. These need to be displayed because they are used to calculate the final velocity and will be used to create a graph. The change in kinetic energy will also show the difference between elastic and inelastic collisions.
6. This will time the length of the collision from the objects starting position to their end position and the value will be used for creating a graph. The objects travel at a constant speed so should technically never stop so I will need to add in limits.
7. The values stored will be used to create a graph.
8. Data will be retrieved from the database and I will create an algorithm to transform it into a graph. The graph will display kinetic energy against time because in elastic collisions, kinetic energy is conserved but in inelastic collisions it is not so the graph will demonstrate the difference between these collision types.
9. There needs to be away to end the program so that it does not continue forever enabling users to exit and continue with other work.

Limitations

The final solution will have some limitations such as:

- It will not have an option for explosion collisions because my simulation is aimed at students studying GCSE physics and explosion collisions are not on their specification. The algorithm for calculating this type of collision is very similar to the one for inelastic collisions so it would not add any complexity to me program either.
- It will only show collisions on a flat surface because GCSE students do not need to learn about angular collisions.
- It will only show collisions of objects traveling at a constant speed because the simulation is meant to demonstrate the idea of momentum and this will be easier for students to understand if the speeds are constant.
- The values inputted and outputted will only work in SI units because these are the standard units and GCSE students should be able to convert other units into SI units so there is no point in adding a unit converter to my program.

- The simulation will have to end after a certain amount of time because otherwise the objects would continue moving forever since they travel at a constant speed so wont decelerate.
- The simulation will not include sound because I feel this does not add anything to the project that makes it more suitable to the stakeholders needs since it is not educational and will likely just distract users from the varying data.
- Users will not be able to change the speed of the simulation or pause it because the objects will move slowly enough to see the collision and change in data by default so it is unnecessary.
- Users won't be able to change the amount my program rounds its calculations by because the simulation is meant to give students a basic understanding of how momentum collisions work and I don't feel that rounding settings will help me to achieve this goal.

Requirements

Stakeholder Requirements:

1. Clear graphical user interface displaying two objects on a flat surface
2. A simple way to vary the conditions of the objects
3. GUI displaying graph of kinetic energy against time
4. An obvious way to end the program

Software:

5. Windows 7 operating system
6. Python 3.3.2 interpreter
7. Pygame 1.9.2a0 for Python 3.3.2
8. Excel 2016

Hardware:

9. A computer with a fast enough processor to run the software, which most laptops and desktop computers have.
10. Monitor
11. Keyboard
12. Mouse

Justification:

1. Because a simulation needs to teach someone how to do something and in this scenario a graphical user interface is the best way to do this.
2. So that users can see how momentum varies with velocity and mass which teaches them how momentum calculations are made.
3. Because this would be very difficult to just display in python without a GUI and it is needed to highlight the difference between elastic and inelastic collisions.
4. So that it does not continue forever enabling users to exit and continue with other work. It needs to be obvious so that new users do not need instructions since these will not be included in my project.
5. Because this OS is supported by Python 3.3.2 which is the programming language I will be using and it is the system used in my school.

6. Because my program will be written in python since this is the language I am most proficient in.
7. Because my stakeholder would like my program to have a GUI and Pygame is a tool I have experience in using.
8. This will be used to store the data for the graph in a CSV file.
9. Because it will need to be capable of running the software required otherwise the program will not work properly. My stakeholder has had issues with other simulations being too slow.
10. To view the software used which will make it easier to manipulate.
11. To navigate software; this allows the user to enter different values into the GUI.
12. To navigate software; this allows the user to select different parts of the GUI.

Success Criteria

In order for my project to be suitable for my stakeholder while including all the essential features and requirements it will have to adhere to this criteria:

No.	Criteria:	Measurement/ Evidence:
1	Program must run successfully using Python 3.3.2 without errors occurring.	Run program multiple times entering different values e.g. values at validation boundaries.
2	Main window containing graphical user interface using Pygame 1.9.2a0 showing two objects (icons) on a flat surface.	Screenshot of GUI.
3	Simple, easy to navigate graphical user interface.	Get my stakeholder and other potential users to test the product and give feedback.
4	Entry widgets for the velocity and mass of both objects.	Screenshot of GUI, test whether values can entered.
5	Series of two radio buttons to allow users to choose between elastic and inelastic collisions.	Screenshot of GUI, test by selecting different options.
6	Algorithms written using object oriented programming that work out the correct momentum and kinetic energy.	Check whether correct calculations made using formula or calculators.
7	Algorithm working out the final velocities after an elastic collision.	Check whether correct calculations made using formula or calculators.
8	Algorithm working out the final velocity after an inelastic collision.	Check whether correct calculation made using formula or calculators.
9	Final velocity, total momentum and change in kinetic energy rounded to two decimal places.	For outputs count the number of digits after decimal places.
10	Widgets on GUI displaying the final velocity of both objects, the total momentum and the change in the kinetic energy with their appropriate unit.	Screenshot of GUI.
11	SI units (kg, m/s, kgm/s and J) used for velocity, mass, momentum and energy.	Check correct unit comes after values.
12	GUI including working start, reset and clear buttons.	Screenshot of GUI and test buttons.
13	Database that stores the velocities and masses and the calculated total momentum and change in kinetic energy for an inelastic collision.	Screenshot of database, check correct data entered by comparing to inputs.

14	Animation displayed on the GUI demonstrating the two objects colliding.	Screenshots of GUI.
15	Algorithm written using object oriented programming that calculates the time taken for the collision.	Check whether correct calculation made using formula or calculators.
16	Algorithm that works out the points to be plotted on a kinetic energy against time graph.	Check whether correct calculations made using formula.
17	GUI that displays the graph of kinetic energy against time.	Screenshot of GUI.
18	An obvious way to end the program.	Get my stakeholder and other potential users to test the product to ensure they find it easy to navigate.

Justification of success criteria:

1. This is because if there are errors in the code it will not be able to work out correct values which would mean the program could not fulfil its purpose of simulating a collision and the people using it as a learning tool may learn incorrect information. It must use python 3.3.2 so that the program works correctly on all computers because they can all use the same version.
2. Because the purpose of a simulation is to demonstrate an idea and in the case of a momentum collision simulation a GUI is the most effective way to do this.
3. This is because my stakeholder mentioned the issue with many other simulations he has used is that they are difficult to manipulate but this will make it easier to do so.
4. Because the formula for calculating momentum and kinetic energy involve mass and velocity so these values need to be inputted which is easiest to do via a GUI.
5. My stakeholder requested an option for elastic and inelastic collisions because GCSE students need to know the difference between them and how to calculate the final velocities for both these collision types. It will also add complexity to my project. Radio buttons are the best way to do this because everyone recognises how they work so it will be easy to understand.
6. Because these values are needed to calculate final velocity and to create the graph. Object oriented programming will be used because it is easier to manage, reuse and understand.
7. This is needed for the animation and to better students understanding of the effect an elastic collision has on the speed of two objects.
8. This is needed for the animation and to better students understanding of the effect an inelastic collision has on the speed of objects.
9. Because in the GCSE physics specification it states that calculations should be rounded to two decimal places unless instructed otherwise.
10. These values will better students understanding of the collision and the difference between elastic and inelastic collisions. They will be shown on a GUI because it is the clearest method of display.
11. Because these are the standard units used for these values and if students need them in an alternative unit they will already how to convert them.
12. This is because my stakeholder mentioned the issue with many other simulations he has used is that they are too slow or difficult to manipulate but these features will make it easier to do so.

13. Because this can be used to create graphs or for students to see a collection of how different values of masses and velocities effect inelastic collisions. I decided to only do this for inelastic collisions because although my stake holder requested a simulation for elastic and inelastic collisions, GCSE students only need to understand inelastic collisions.
14. This will demonstrate a momentum collision which is the purpose of my project. The GUI and animation make it easier to for students to understand since most people are visual learners.
15. This is needed for the kinetic energy against time graph. Object oriented programming will be used because it is easier to manage, reuse and understand.
16. This is needed to plot the results.
17. This is needed to show how kinetic energy varies before and after collisions to demonstrate the difference between elastic and inelastic collisions.
18. So that it does not continue forever enabling users to exit and continue with other work. It needs to be obvious so that new users do not need instructions since these will not be included in my project.

Decomposition:

Earlier I mentioned my program would be broken down into three sections to make it more solvable by computational methods. Each section will cover different parts of the success criteria:

- Section 1- In this section I will create the algorithms for calculating the final velocity, total momentum and change in kinetic energy. The values will be inputted and outputted via the python shell so no GUI will be used at this point. It will also set up the database for storing values entered. This will cover parts 1, 6, 7, 8, 9, 11 and 13 of the success criteria.
- Section 2- In this section I will create the algorithm for the kinetic energy against time graph which also requires an algorithm calculating the time taken for the collision. The graph will be displayed on a GUI. This will cover parts 15, 16, 17 and 18 of the success criteria as well as the parts covered in the previous section.
- Section 3- In this section I will create a GUI displaying the values inputted, the animation and the values calculated using Pygame. This will cover parts 2, 3, 4, 5, 10, 12 and 14 of the success criteria as well as the parts covered in the previous sections.

Section 1 Design:

Decomposition

Success criteria for this section:

1. Program must run successfully using Python 3.3.2 without errors occurring.
2. Algorithms written using object oriented programming that work out the correct momentum and kinetic energy.
3. Algorithm working out the final velocities after an elastic collision.
4. Algorithm working out the final velocity after an inelastic collision.
5. Final velocity, total momentum and change in kinetic energy rounded to two decimal places.
6. SI units (kg, m/s, kgm/s and J) used for velocity, mass, momentum and energy.
7. Database that stores the velocities and masses and the calculated total momentum and change in kinetic energy for an inelastic collision.

Justification:

I have chosen to cover these parts of my success criteria in this section because these are the key algorithms needed in a momentum collision simulation and other parts of my criteria like the GUI and graph would be impossible to create without these particular algorithms. I chose to break my design section down into this series of smaller problems to make it more suitable for computational solutions by using methods like visualisation and the divide and conquer approach.

Solution Structure

Versions:

Breaking down each section of my development process into different sub versions via decomposition will make the project more efficient. I plan to make roughly seven different sub versions of my code for section one but I may require more:

- V1.1 – calculating momentum and kinetic energy of both individual objects
- V1.2 – calculating final velocity for the objects depending on the collision type
- V1.3 – calculating the change in kinetic energy and allowing user input
- V1.4 – rounding outputs, displaying units, validation
- V1.5 – storing values in database
- V1.6 – sorting values in database
- V1.7 – final version

During development, I will break this section down into these sub versions to make it easier to understand and because this is the order the success criteria needs to be completed since each version requires the parts completed in the previous version. I included V1.7 (final version for section 1) to ensure that everything on the success criteria for this section is included so that I do not miss anything. Having multiple files for the versions also means that if one version stops working after development I will have a backup.

Database structure:

The database will be named 'database' so its purpose is clear and it will be a CSV file created using Microsoft Excel 2016. It will only be for inelastic collisions because that is the collision type GCSE students study. It will contain these six columns because these are enough to represent the varying data:

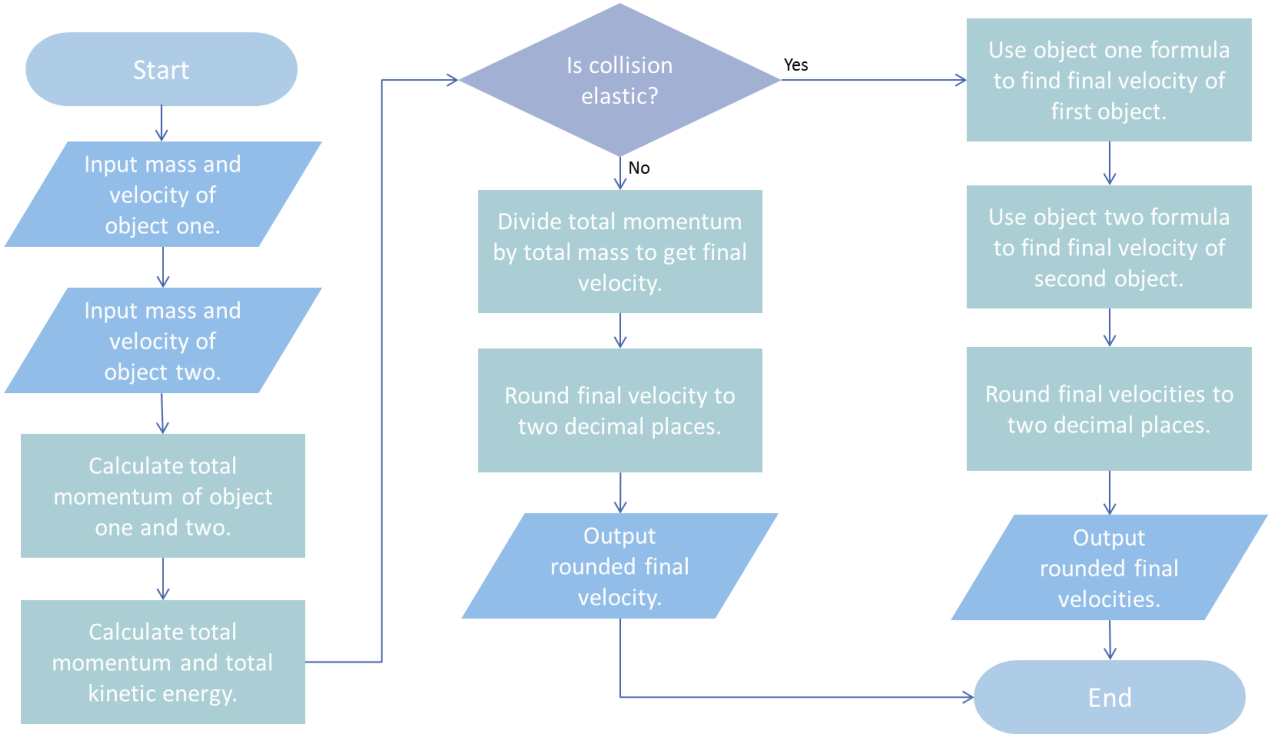
Mass 1	Velocity 1	Mass 2	Velocity 2	Momentum	Change in Ek
…	…	…	…	…	…

I decided use a CSV file instead of a text file or SQL database because the purpose of this database is to allow users to see how the data varies and to create a graph. In text files, the layout is not clear enough for users to view and I do not need to use an SQL database because they are more appropriate for relational databases but mine will just be a single table. Additionally, my program will always retrieve all data from the table which is faster when using CSV files instead of an SQL database.

Algorithms

I decided to write some algorithms for section 1 of my code; this allows me to plan ahead key variables and data structures that I may need and enables me to find out if there is any further research I need to carry out in order to adhere to my success criteria. This will also make my development process more efficient.

Calculating final velocity (flow chart):

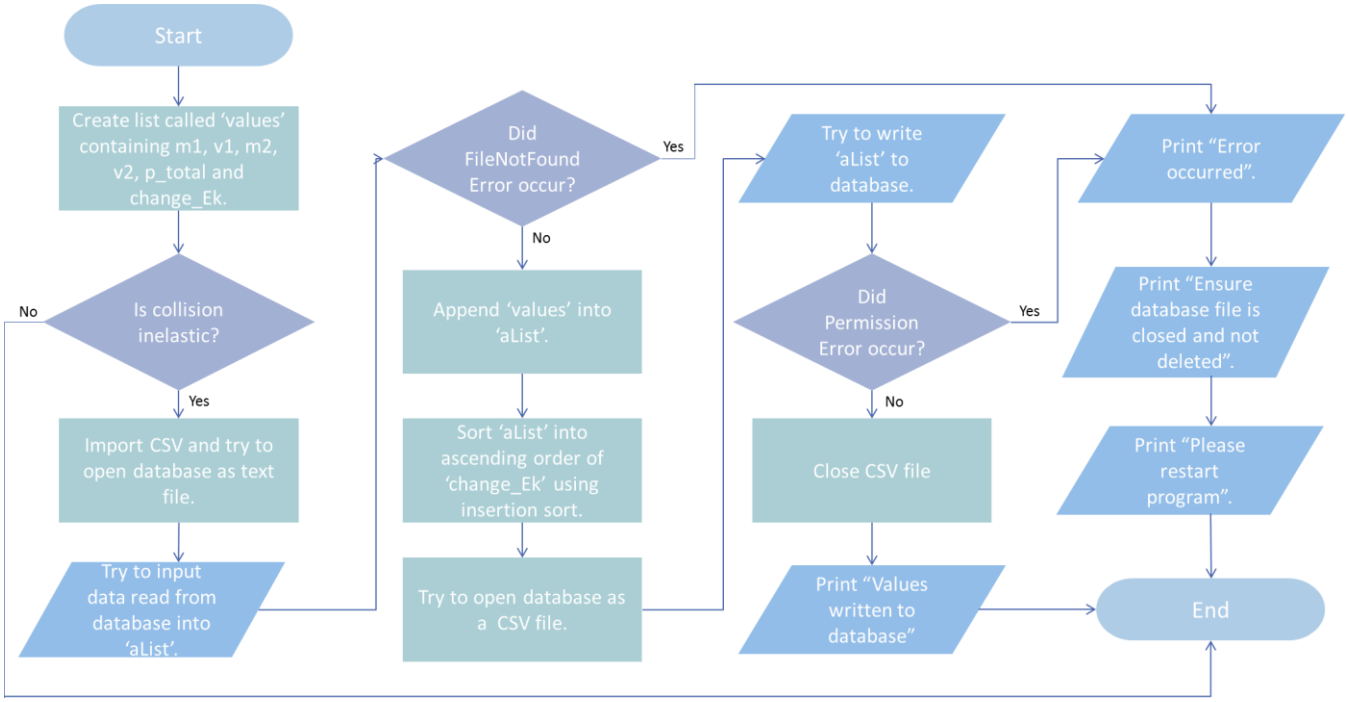


Calculating change in kinetic energy (pseudo code):

```
Function KineticEnergy(mass, velocity):
      Ek = 0.5 * mass * velocity ^2
      Ek = round(Ek, 2 decimal places)
      Return Ek
Endfunction
M1 = Input mass of object 1
V1 = Input velocity of object 1
M2 = Input mass of object 2
V2 = Input velocity of object 2
Ek1 = KineticEnergy(M1,V1)
Ek2 = KineticEnergy(M2,V2)
```

```
Total_Ek = Ek1 + Ek2
FinalV1 = final velocity of object 1
FinalV2 = final velocity of object 2
Ek1 = KineticEnergy(M1,FinalV1)
Ek2 = KineticEnergy(M2,FinalV2)
Total_Ek_After = Ek1 + Ek2
ChangeEk = Total_Ek - Total_Ek_After
Output (ChangeEk)
```

Writing to Database (flow chart): -



Insertion Sort (pseudo code): – I decided to use a sort to put the values in my database into ascending order of change in kinetic energy because this will make it clearer to understand how data varies as kinetic energy varies and easier to plot a graph. I will use this sort method because insertion sorts are relatively quick if the list is almost sorted (for best-case scenario time complexity close to O(n), n is number of items in list) and since the list is sorted each time an item is added it will always be almost sorted.

```
Procedure insertionSort (Ek)
    n = len(Ek)
    for index = 1 to n - 1
        currentvalue = Ek[index]
```

```
            position = index
            while position > 0 and Ek[position - 1] > currentvalue
                  Ek[position] = Ek[position - 1]
                  position = position - 1
            endwhile
            Ek[position] = currentvalue
      next index
endprocedure
```

Justification:

For my real program, these algorithms will be parts of functions and classes. These algorithms form a complete solution to Section 1 of my project because they cover all points on my success criteria. The 'Calculating final velocity' and 'Calculating kinetic energy' algorithms cover the first five parts of my success criteria for this section. The 'Writing to Database' and 'Insertion Sort' algorithms cover the last two parts of my success criteria for this section. Calculating kinetic energy will involve the values returned from the 'Calculating final velocity' algorithm and writing to the database will involve the sort algorithm and the values returned by the other algorithms. I will explain and justify the roles of the variables used in these algorithms in my data dictionary.

Usability Features

The usability features I plan to use to enable my stakeholders to achieve the specified goals of my product with effectiveness; efficiency and satisfaction are mostly relevant in sections two and three of my design. However, to improve ease of use of the first section of my project I will need to make sure that:

- Outputs are displayed as text in the shell and are in a clear font of a reasonable size so that people with vision difficulties can read them and screen readers for blind people can detect the text.
- Navigating the program is not overly complex because the program will not come with instructions and users will likely have no experience using the python shell. I will do this by clearly stating what should be inputted where and what the options are for input. For certain inputs I will show the options that can be chosen and if invalid data is inputted a statement will be printed and the user will be able to renter valid data.
- There is good communication with the user so that they understand what data inputted and outputted represents and what calculations took place to find the final velocities. I will do this by using print statements to label inputs and outputs and if an error occurs when reading or writing to the database a print statement will tell the user how to prevent this error from occurring.

Development Plan

Data Dictionary:

These are the key variables, data structures and classes I plan to use in section 1 of my program:

Name:	Data Type:	Description/Justification:
Momentum_Ek	Class	To calculate the initial momentum and kinetic energy of an object.
Collision	Class	To calculate the final velocities of the objects and change in kinetic energy after the collision.
Database	Class	To read information from the database and write the new data.
Sort	Function	Sorts the list to be written to the database into ascending order so that it is easier to understand.
Validation	Function	To ensure mass and velocity inputted are integers and that mass is greater than 0.
m1, m2	Float	To store the user input of the masses of both objects. Named m1 and m2 so people know which mass is for which object and because m is the symbol used to represent mass in physics.
v1, v2	Float	To store the user input of the initial velocities of both objects. Named v1 and v2 so people know which velocity is for which object and because v is the symbol used to represent velocity in physics.
collision _type	String	To store the users input as to whether the collision will be elastic or inelastic. It is a string because the value entered will be a word.
p1, p2	Float	To store the momentum of both objects. Named p1 and p2 so people know which momentum is for which object and because p resembles the Greek symbol 'rho' which represents momentum in physics.
Ek1, Ek2	Float	To store the kinetic energy of both objects. Named Ek1 and Ek2 so people know which kinetic energy is for which object and because Ek is the symbol used to represent kinetic energy in physics.
p_total	Float	To store the total momentum of both objects.
Ek_total	Float	To store the total kinetic energy of both objects.
inelastic_v	Float	To store the final velocity after an inelastic collision. Only needs one variable because objects after inelastic collision have the same speed.
elastic_v1	Float	To store the final velocity of object 1 after an elastic collision.
elastic_v2	Float	To store the final velocity of object 2 after an elastic collision.
final_Ek	Float	To store the total kinetic energy of both objects after the collision.
change_Ek	Float	To store the difference between Ek_total and final_Ek
Values	List	To store all the inputted and calculated values that need to be written to the database.
aList	List	To store all the data read from the database so that it can be sorted. Named aList because this is a standard name for lists.

I chose these variable names because they are suitable for their function and sum up their purpose for example the 'collision_type' variable stores the type of collision being processed. This will make it easier for me when I am developing my code and for people viewing or editing my code in the future. The variables that involve numbers will be floats not integers because the values entered and calculated may involve decimal places.

Validation:

Some of the variables I plan to use in my program will need to be specific data types or within certain boundaries in order for my program to run without errors:

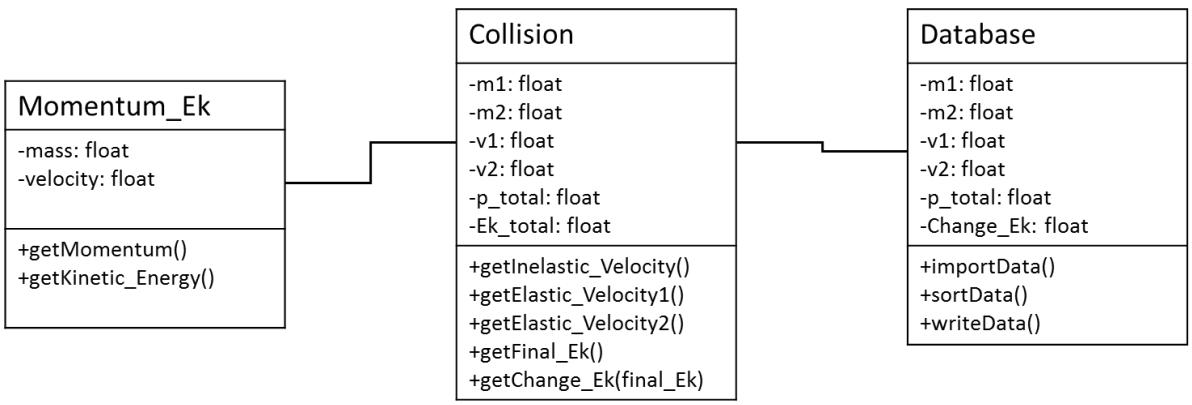
1. m1 > 0 and m2 > 0
2. v1 ≥ 0
3. v2 ≤ 0
4. m1, m2, v1 and v2 must be floats
5. collision_type = elastic or collision_type = inelastic

Justification:

1. Because mass is a scalar quantity so only has magnitude not direction meaning the mass of an object needs to be greater than zero for it to exist.
2. Because velocity is a vector quantity so it has magnitude and direction, v1 being positive means it moves the opposite direction to an object with negative velocity. For my program I will consider positive velocities to move right and negative velocities to move left but this will only be relevant for section 2 and 3. The velocity can be zero which would mean that the object remains stationary until the collision occurs.
3. Because velocity is a vector quantity so it has magnitude and direction, v2 being negative means it moves the opposite direction to v1 resulting in the objects colliding. The velocity can be zero which would mean that the object remains stationary until the collision occurs; if both objects have a velocity of zero a collision will not take place.
4. Because these values must be numbers not strings in order for the algorithms to work but some users may want to enter data that includes decimal places so the numbers must be floats not integers.
5. Because these are the two collision types my program will be able to calculate the final velocities for.

Class diagram:

For my final program the algorithms displayed earlier will be parts of classes since my project will be written using object oriented programming. “Object-oriented programming (OOP) is a programming language model organized around objects rather than actions and data rather than logic” – (Rouse, 2008). This diagram shows the classes, their attributes, their methods and the relationships between them.



Testing

Test Data:

The values in the input section of this table are the test data I will use to test that the function of a particular section of code is working so that all points on the success criteria are met. The 'output' and 'changes needed' column will be filled in during the testing stage of my development process and the first column shows which point on the success criteria this is testing.

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
1	Mass Inputted is > 0	m1 = 0 m2 = 0	Boundary	Invalid Mass Input Mass:		
1	Mass Inputted is > 0	m1 = 0.001 m2 = 0.001	Boundary	Valid Mass		
1	Mass Inputted is > 0	m1 = -5 m2 = -5	Error	Invalid Mass Input Mass:		
1	Mass Inputted is > 0	m1 = 5 m2 = 5	Normal	Valid Mass		
1	Mass inputted is float	m1 = five m2 = 5g	Error	Invalid Mass Input Mass:		
1	v1 ≥ 0	v1 = 0	Boundary	Valid Velocity		
1	v1 ≥ 0	v1 = 1	Boundary	Valid Velocity		
1	v1 ≥ 0	v1 = -1	Boundary	Invalid Velocity Input v1:		
1	v1 ≥ 0	v1 = 10	Normal	Valid Velocity		
1	v1 ≥ 0	v1 = -10	Error	Invalid Velocity Input v1:		
1	v1 is a float	v1 = ten	Error	Invalid Velocity Input v1:		
1	v2 ≤ 0	v2 = 0	Boundary	Valid Velocity		
1	v2 ≤ 0	v2 = -1	Boundary	Valid Velocity		
1	v2 ≤ 0	v2 = 1	Boundary	Invalid Velocity		
1	v2 ≤ 0	v2 = -10	Normal	Input v2:		
1	v2 ≤ 0	v2 = 10	Error	Valid Velocity		
1	v2 is a float	v2 = 10m/s	Error	Invalid Velocity		
2	Correct momentum calculated	m1 = 2 v1 = 3	Normal	p1 = 6 kg m/s		

2	Correct momentum calculated	m2 = 1.4787 v2 = -4.678	Normal	p2 = -6.92 kg m/s		
2	Correct kinetic energy calculated	m2 = 2 v2 = 3	Normal	Ek2 = 9 J		
2	Correct kinetic energy calculated	m1 = 1.4787 v1 = -4.678	Normal	Ek1 = 16.18 J		
4	Correct final velocity calculated for inelastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	inelastic_v = 3 m/s		
4	Correct final velocity calculated for inelastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	inelastic_v = 3.63 m/s		
3	Correct final velocities calculated for elastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	elastic_v1 = 1 m/s elastic_v2 = 9 m/s		
3	Correct final velocities calculated for elastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	elastic_v1 = 0.8 m/s elastic_v2 = 11.94 m/s		

I will also need to test the other points on the success criteria but these tests do not require test data and will be done by viewing the outputs and comparing values:

- I will test point 5 on the success criteria by viewing all the outputs and making sure there are only two digits after the decimal place.
- I will test point 6 on the success criteria by viewing the outputted values and making sure the correct unit comes after the values, I know which units are correct from my research in the analysis section.
- I will test point 7 on the success criteria by running my program multiple times and entering different data then open the database and compare the data to the outputs in the python shell.

Justification:

To test that the values inputted by the user are validated appropriately I will need to enter values that are at the variables boundaries to ensure I have used the correct inequality symbols in my code. I will also need to test normal and error data to make sure that valid data does work and invalid data does not in order to prevent errors from occurring during further development. The values I will use to test that correct calculations are being made I will use because these are the numbers used in the example momentum collision questions I found during my research so I know what the correct answers are. I will also check these calculations myself using the formulas and a calculator since I have taken GCSE physics.

Post Development Testing:

After finishing section one of my program and testing it using the table above I will then begin development of section two then section three. Although these sections focus on different parts of my project the functions of section one will still need to work in order for the final version to adhere to all points on my success criteria. This means that the parts I test during section one will need to be tested against the success criteria again after further development in other sections.

Stakeholder Input:

After developing section one of my program I plan to show it to Mr Hutchinson, my stakeholder, so that he can give some feedback on my project so far. This will allow him to make further suggestions, request any changes and to ensure that up until now my project does everything he would like it to do that was meant to be completed in section one.

Section 2 Design:

Decomposition

Success criteria for this section:

1. Algorithm written using object oriented programming that calculates the time taken for the collision.
2. Algorithm that works out the points to be plotted on a kinetic energy against time graph.
3. GUI that displays the graph of kinetic energy against time.
4. An obvious way to end the program.
5. The points on the success criteria of the previous section.

Justification:

I have chosen to cover these parts of my success criteria in this section because they are all required to create the kinetic energy against time graph; the graph is more important than the animation side of the GUI because the graph demonstrates the difference between elastic and inelastic collisions which is one of the key purposes of my program. I chose to break my design section down into this series of smaller problems to make it more suitable for computational solutions by using techniques such as heuristic methods and the divide and conquer approach.

Solution Structure

Versions:

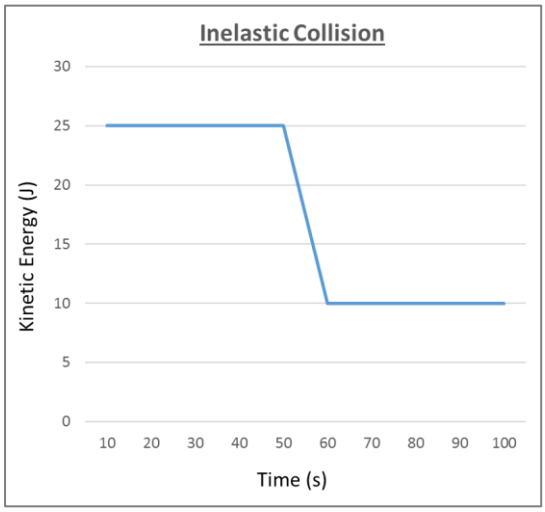
Breaking down each section of my development process into different versions via decomposition will make the project more efficient. I plan to make roughly four different sub versions of my code for section two but I may require more:

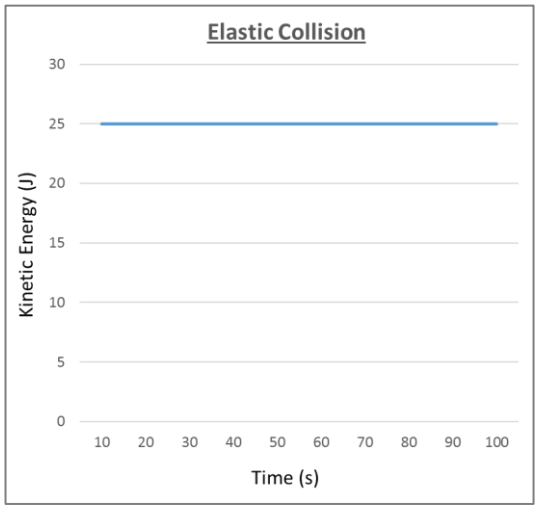
- V2.1 – Algorithm working out time taken for collision
- V2.2 – Algorithm working out points to be plotted on graph
- V2.3 – GUI displaying graph of kinetic energy against time
- V2.4 – Final version

During development, I will break this section down into these sub versions to make it easier to understand and because this is the order the success criteria needs to be completed since each version requires the parts completed in the previous version. I included V2.4 (final version for section 2) to ensure that everything on the success criteria for this section is included so that I do not miss anything and to make sure all the parts I developed in previous sections are still working. Having multiple files for the sub versions also means that if one stops working after development I will have a backup.

Graph layout:

The graph displayed on the GUI will be of kinetic energy against time because this will show how the total kinetic energy of the system varies as the collision takes place. This will demonstrate to students the difference between elastic and inelastic collisions because in elastic collisions kinetic energy is conserved so does not change whereas in inelastic collisions some kinetic energy is transferred to other forms. This means the graph for elastic collisions will just be a horizontal line whereas for inelastic it will begin as a vertical line then suddenly drop to another vertical line but lower on the axis. I used my file of data on Excel to create examples of the graphs that will be displayed on my GUI.





The graphs will look very similar to this but will instead be displayed on a graphical user interface using Pygame. I will make 100 seconds the maximum time for the collision because this will stop the graph continuing forever but is a short enough time for the collision to take place. My program will retrieve data more frequently than in these examples so the graph will be more precise.

Algorithms

I decided to write some algorithms for section 2 of my code; this allows me to plan ahead key variables and data structures that I may need and enables me to find out if there is any further research I need to carry out in order to adhere to my success criteria. This will also make my development process more efficient.

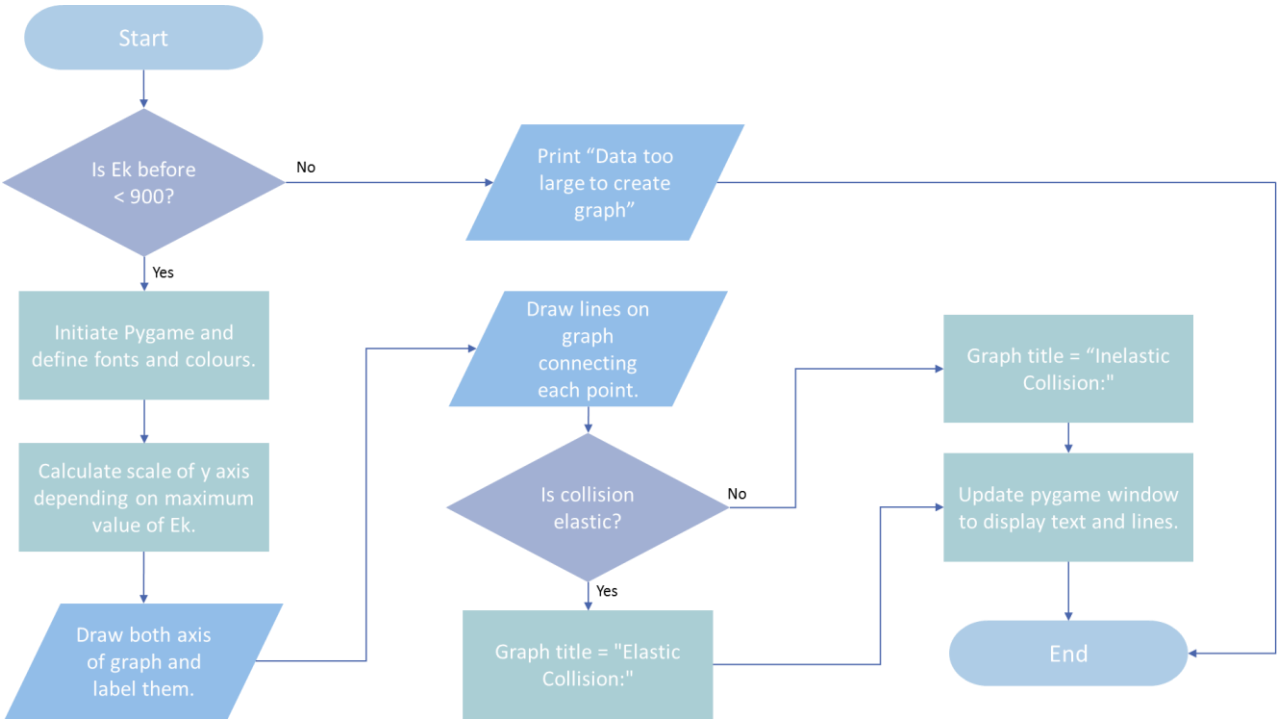
Calculating collision time (pseudo code):

```
x1 = 0
x2 = 500
time_taken = 0
maxtime = 1000
while x2 >= x1 and time_taken < maxtime
      x1 = x1 + v1
      x2 = x2 – v2
      time_taken = time_taken + 1
```

Finding values of points on graph (pseudo code):

```
points = []
for i in range (0 to time_taken + 1)
      point = [i, Ek_before]
      append point to points
for i in range (time_taken + 1 to maxtime)
      point = [i, Ek_after]
      append point to points
```

Displaying graph on GUI (flowchart):



Justification:

For my real program, these algorithms will be methods in a class called 'Graph'. These algorithms form a complete solution to Section 2 of my project because they cover all points on my success criteria for this section. The 'Calculating collision time' covers the first point on my success criteria, the 'Finding points on graph' algorithm covers the second point and the last algorithm covers the rest of the success criteria. I will need to develop these algorithms in the order they are written because each algorithm uses the outputs of the previous ones and the program for section 1. I will explain and justify the roles of the variables used in these algorithms in my data dictionary.

Usability Features

I plan to include usability features that will enable my stakeholders to achieve the specified goals of my product with effectiveness, efficiency and satisfaction. To improve ease of use of this section of my project I will need to:

- Give the graph a title and label the axis of the graph with the value it shows and its unit so that users can tell what the data plotted represents meaning they can interpret the graph to tell whether the collision is elastic or inelastic. This will also demonstrate to students the correct way to plot a graph, which they will need to know for their GCSE.
- Make sure that all text on the graph is in a clear font of a reasonable size so that people with vision difficulties can read them and screen readers for blind people can detect the text.
- Make the program easy to navigate because it will not come with instructions and users will likely have no experience using the python or pygame. I will do this by making the GUI for the graph appear automatically after the calculations from section 1 are made and then the graph will plot itself so no further user input is required.

- Make the line plotted on the graph thick enough for people to see but small enough so that it is clear which point on the graph it corresponds to.

Development Plan

Data Dictionary:

These are the key variables, data structures and classes I plan to use in section 2 of my program in addition to the ones from section 1:

Name:	Data Type:	Description/Justification:
Graph	Class	To calculate the time taken for the collision which is used to create a GUI in pygame displaying a graph of kinetic energy against time.
Time_taken	Float	To store the time taken for the two objects to reach each other/collide which can be used to create the graph.
point	List	To store the x (time) and y (kinetic energy) coordinate of a single point on the graph.
points	List	To store a list of all the points that will be plotted on the graph.
white	Class	Tuple class that stores the numbers representing white in the RGBA colour values. Used for background of pygame window.
black	Class	Tuple class that stores the numbers representing black in the RGBA colour values. Used for axis and text on graph.
blue	Class	Tuple class that stores the numbers representing blue in the RGBA colour values. Used for plotting points and joining them on graph.
font	Class	Font class built into the pygame module that will load and render fonts which I will use for the text that will be displayed on the GUI.
scale	Integer	To store the scale of the y axis e.g. scale = 10 if y axis goes up in steps of 10J. Scale will depend on the value of Ek_before because this will be the highest possible value on the y axis so scale will need to be big enough for graph to fit on the screen.
max_Ek	Float	To store value of Ek_before rounded to the nearest 10 or 100 (depending on the scale of the graph). This will be used to determine the length the y axis needs to be.
y_axis	Float	To store the length of the pygame window which will be used to determine the y position of the points that need to be plotted on the graph.
x_axis	Float	To store the width of the x axis which will be used to determine the x position of the points that need to be plotted on the graph.
message	String	To store the message that will be displayed as the title of the graph. The message will depend on whether the collision is elastic or inelastic.
text	Class	Surface class built into the pygame module that creates a new surface with text rendered on it. I will use this to display the 'message'.
x1	Float	To store the position of object 1, it will start at 0 and increase as the objects move closer together. The letter 'x' is used because in physics this is the symbol used to represent displacement.
x2	Float	To store the position of object 2, it will start at 50 and decrease as the objects move closer together. The letter 'x' is used because in physics this is the symbol used to represent displacement.
maxtime	Float	To store the maximum time for the collision, used to make sure that the program does not continue forever which could happen if the velocity of both objects is 0.

I chose these variable names because they are suitable for their function and sum up their purpose for example 'Time_Taken' stores the time taken for the collision to take place. This will make it easier for me when I am developing my code and for people viewing or editing my code in the future. Most of the variables that involve numbers will be floats not integers because the values calculated involve division and use variables that are floats themselves.

Validation:

There is very little validation needed in section 2 because the algorithms mostly use the values calculated in section 1 which will have already been validated. These are the new variables I need to validate in order for my program to run without errors:

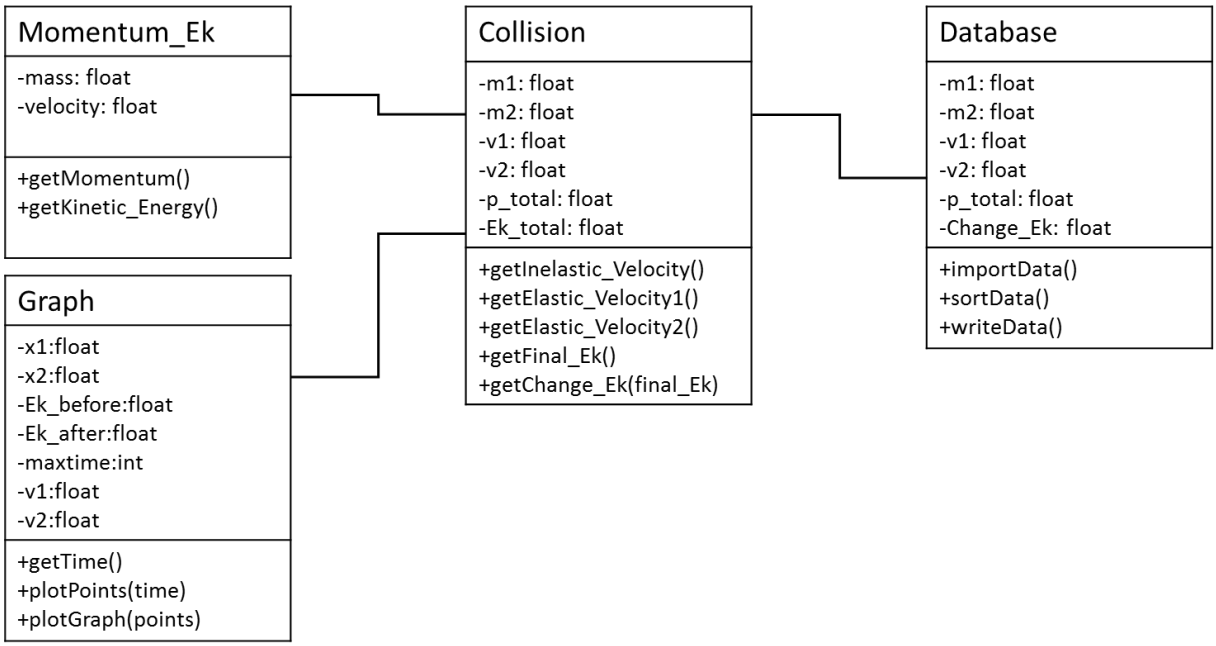
1. maxtime < 100
2. Ek_max < 900

Justification:

1. Because most collisions of objects travelling at a constant speed will continue forever in this model so I need to have a maximum time otherwise my program will be stuck in an infinite loop. This could also happen if both objects had a velocity of zero. I chose 100 seconds for the maximum collision time because from looking at GCSE Physics practice questions I can see that students do not normally need to work with velocities that are low enough for a collision to take longer than 100 seconds.
2. Because a window size greater than 1,000 is bigger than the resolution of the screens used in most schools and the minimum scale I will use is 1:1. If I include the gaps I will have to leave between the axis of the graph and the edges of the window the maximum length of the y axis must be 900.

Class diagram

For my final program, the algorithms displayed earlier will be parts of a class since my project will be written using object oriented programming. This diagram shows the new class from section 2, its attributes, methods and the relationships between it and the classes from section 1.



Testing

Test Data:

The values in the input section of this table are the test data I will use to test that the function of a particular section of code is working so that all points on the success criteria are met. The 'output' and 'changes needed' column will be filled in during the testing stage of my development process and the first column shows which point on the success criteria this is testing.

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
1	Maximum time < 1000	v1 = 5 v2 = -3	Normal	Time taken = 126		
1	Maximum time < 1000	v1 = 0 v2 = 0	Error	Time>1000 so set Time taken = 1000		
1	Maximum time < 1000	v1 = 0.5 v2 = -0.5	Boundary	Time taken = 1000		
1	Maximum time < 1000	v1 = 0.4 v2 = -0.5	Boundary	Time>1000 so set Time taken = 1000		
2	Correct points in list of points	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678 Type: Inelastic	Normal	Time taken = 90 List of cords [x,y], x starts at 106.74 then when y=time taken x changes to 38.37.		
2	Correct points in list of points	m1 = 3 v1 = 5 m2 = 1 v2 = -3 Type: Elastic	Normal	Time taken = 126 List of cords [x,y], x stays at 42, y increases from 0 to 1000		
3	Kinetic energy rounded correctly	m1 = 4 v1 = 3	Normal	Ek_before = 19.5 MaxEk = 20		

		m2 = 3 v2 = -1				
3	Kinetic energy rounded correctly	m1 = 7 v1 = 6 m2 = 3 v2 = -10	Normal	Ek_before = 276 MaxEk = 300		
3	Maximum Ek < 900	m1 = 4 v1 = 3 m2 = 3 v2 = -1	Normal	Ek_before = 19.5 MaxEk = 20 Graph Created		
3	Maximum Ek < 900	m1 = 7.6 v1 = 20.14 m2 = 3.74 v2 = -11.65	Error	Ek_before = 1777 Data too big to create graph		
3	Maximum Ek < 900	m1 = 100 v1 = 3 m2 = 36 v2 = -5	Boundary	Ek_before = 900 MaxEk = 900 Graph Created		
3	Maximum Ek < 900	m1 = 100 v1 = 3 m2 = 36.01 v2 = -5	Boundary	Ek_before = 900.1 Data too big to create graph		
3	Correct scale used for y axis	m1 = 7 v1 = 6 m2 = 3 v2 = -10	Normal	Ek_before = 276 Scale = 100		
3	Correct scale used for y axis	m1 = 4 v1 = 3 m2 = 3 v2 = -1	Normal	Ek_before = 19.5 Scale = 10		
3	Correct scale used for y axis	m1 = 7.6 v1 = 20.14 m2 = 3.74 v2 = -11.65	Error	Ek_before = 1777 Data too big to create graph		
3	Graph titled correctly	Collision Type: elastic	Normal	“Elastic Collision:”		
3	Graph titled correctly	Collision Type: inelastic	Normal	“Inelastic Collision:”		

I will also need to test the other points on the success criteria but these do not require test data and will be done by viewing the outputs and comparing values:

- I will test that the points on the graph are plotted correctly (SC 3) by printing the list of points in the python shell and comparing them to the graph displayed on the GUI. The most important points to check will be at the turning points on the graph. I will compare the graph on the GUI to the graphs I created in this design section to make sure the graph is the shape it should be depending on the collision type.
- I will make sure there is a way to exit the program without errors (SC 4) by selecting the exit button at the top right of the window to check whether this closes the GUI or whether it does not work or an error occurs.

Justification:

To test that the values inputted by the user are validated appropriately I will need to enter values that are at the variables boundaries to ensure I have used the correct inequality symbols in my code. I will also need to test normal and error data to make sure that valid data does work and invalid data does not in order to prevent errors from occurring during further development. The values I put in the 'expected result' column I calculated myself using the equations and a normal calculator; I then tested my calculations were correct using the momentum collision simulators used as examples in the analysis section.

Post Development Testing:

After finishing section two of my project and testing it using the table above I will then begin development of section three. Although this section focuses on different parts of my project the functions of section two will still need to work in order for the final version to adhere to all points on my success criteria. This means that the parts I test during section two will need to be tested against the success criteria again after further development.

Regression testing:

To adhere to point 5 on my success criteria for section 2 I will need to retest my algorithms from section 1. To do this I will use further test data that I will get from the test table I designed in section 1 so that I can make sure everything still works and that the further developments I have made have not affected my algorithms from the previous section.

Stakeholder Input:

I have decided that after developing section 2 of my project I will not review it with Mr Hutchinson because this section focuses on the graph of kinetic energy against time which is an element that I decided to include and was not requested by him in our interview.

Section 3 Design:

Decomposition

Success criteria for this section:

1. Main window containing graphical user interface using Pygame 1.9.2a0 showing two objects (icons) on a flat surface.
2. Simple, easy to navigate graphical user interface.
3. Entry widgets for the velocity and mass of both objects.
4. Series of two radio buttons to allow users to choose between elastic and inelastic collisions.
5. Widgets on GUI displaying the final velocity of both objects, the total momentum and the change in the kinetic energy with their appropriate unit.
6. GUI including working start, reset and clear buttons.
7. Animation displayed on the GUI demonstrating the two objects colliding.
8. The points on the success criteria of the previous sections.

Justification:

I have chosen to cover these parts of my success criteria in this section because the GUI displaying the animation and calculations requires algorithms I previously made so this section should be done last. I chose to break my design section down into this series of smaller problems to make it more suitable for computational solutions by using methods like abstraction and backtracking.

Solution Structure

Versions:

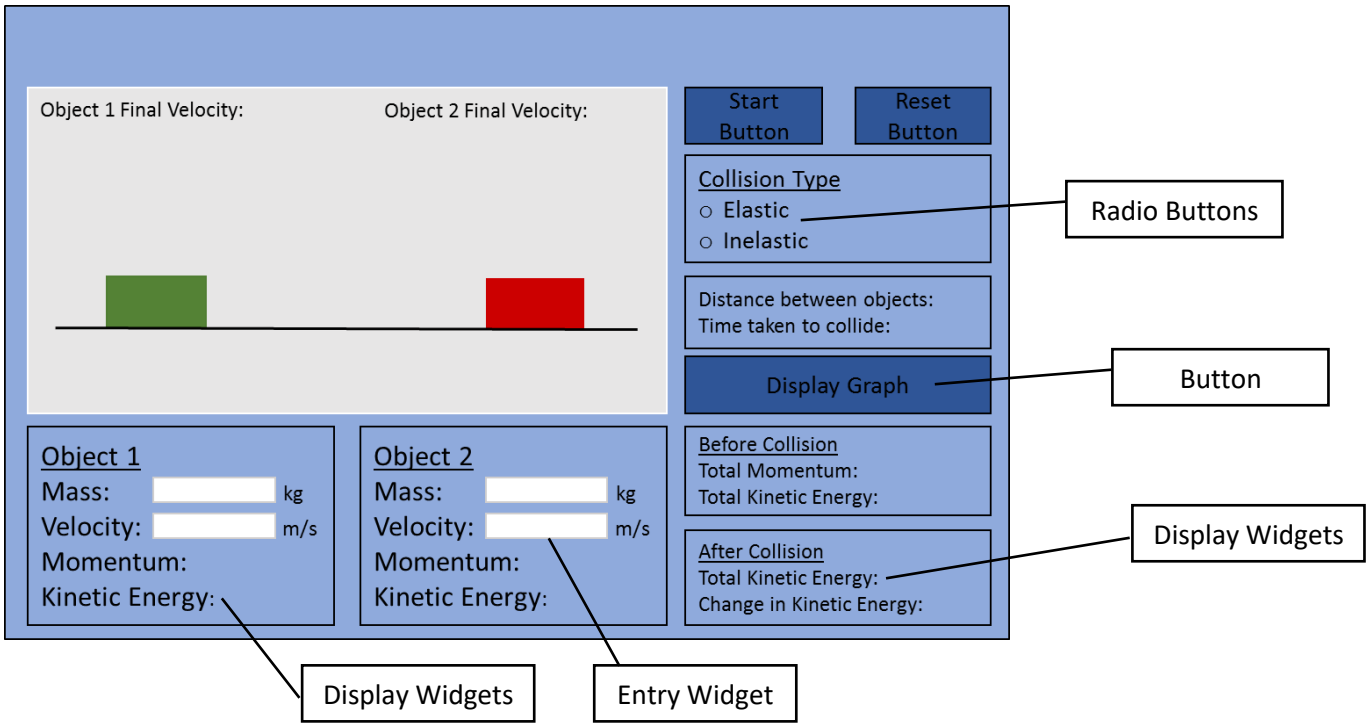
Breaking down each section of my development process into different versions via decomposition will make the project more efficient. I plan to make roughly six different sub versions of my code for section three but I may require more:

- V3.1 – Create main window of GUI allowing users to enter data
- V3.2 – Include start reset and clear buttons in the GUI
- V3.3 – GUI displaying results of calculations
- V3.4 – User Input using GUI
- V3.5 – GUI showing an animation of the collision
- V3.6 – Final Version

During development, I will break this section down into these sub versions to make it easier to understand and because this is the order the success criteria needs to be completed since each version requires the parts completed in the previous version. I included V3.6 (final version for section 3) to ensure that everything on the success criteria for all sections are included so that I do not miss anything and to make sure all the parts I developed in previous sections are still working since this will be my final program. Having multiple files for the sub versions also means that if one stops working after development I will have a backup.

GUI Layout:

According to my success criteria the GUI will need to include these elements:



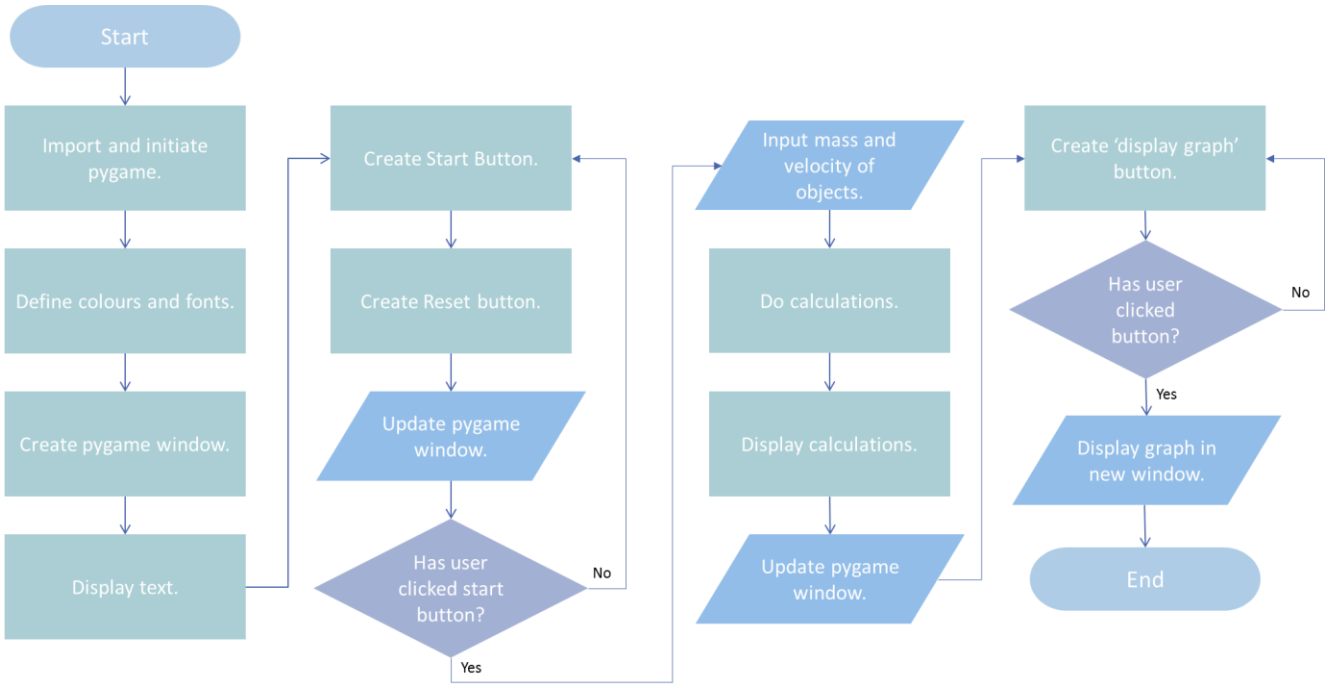
I chose to layout my GUI this way to make it easy to use while displaying all relevant information. I took some inspiration from the example simulations I found during my analysis. The entry widgets and buttons are roughly at the top of the window then the display widgets at the bottom because data needs to be entered before it can be displayed. Object 1 and its corresponding entry and displayed boxes are on the left then objects 2's on the right so that they are in order to make it more obvious which values are for which object. I put units at the end of the entry widgets so users know which value to enter.

I chose to make the objects two different primary colours so that they stand out and the user can tell the difference between them. The background will be light blue because this looks more interesting than a white background and separates the animation from the rest of the GUI. I made the buttons a darker blue so users can tell the difference between a button and an entry/display widget. All the text is black because this is the standard colour and it stands out against the background.

Algorithms

I decided to write some algorithms for section 3 of my code; this allows me to plan ahead key variables and data structures that I may need and enables me to find out if there is any further research I need to carry out in order to adhere to my success criteria. This will also make my development process more efficient.

Creating GUI (flow chart):



Animation (pseudo code):

```
Draw object 1 (x,y)
Draw object 2 (x,y)
distance1 = v1/2
distance2 = v2/2
for i in range (0 to time_taken):
      time.sleep(1)
      Draw object 1 (x+distance1,y)
      Draw object 2 (x-distance2,y)
      distance1 = distance1 + v1/2
      distance2 = distance2 + v2/2
for i in range (time_taken to maxtime):
      time.sleep(1)
      Draw object 1 (x+distance1,y)
      Draw object 2 (x-distance2,y)
      distance1 = distance1 + final_v1/2
      distance2 = distance2 + final_v2/2
```

Justification:

For my real program, these algorithms will be part of a function or methods in a class. These algorithms form a complete solution to Section 3 of my project because they cover all points on my success criteria for this section. The 'Creating GUI' algorithm covers the first 4 points on my success criteria and the 'Animation' algorithm covers the rest. I will need to develop these algorithms in the order they are written because creating the animation requires a GUI. I will explain and justify the roles of the variables used in these algorithms in my data dictionary.

Usability Features

I plan to include usability features that will enable my stakeholders to achieve the specified goals of my product with effectiveness, efficiency and satisfaction. To improve ease of use of this section of my project I will need to:

- Make sure that all text on the GUI is in a clear font of a reasonable size so that people with vision difficulties can read them and screen readers for blind people can detect the text.
- Make the GUI easy to navigate because the program will not come with instructions and users will likely have no experience using the python or pygame. I will do this by making the GUI appear automatically when the program is run and by labelling entry and display widgets appropriately.
- Make the colour of text stand out against the background colour so that it is easy to read allowing users to understand what is going on in the GUI.
- Make the buttons big enough that they are easy to click with the mouse which will make using the GUI more efficient.

Development Plan

Data Dictionary:

These are the key variables, data structures and classes I plan to use in section 3 of my program in addition to the ones from previous sections:

Name:	Data Type:	Description/Justification:
guiDisplay	Class	Class built into pygame module to create GUI window and update elements inside it.
Button	Function	Used for creating the buttons so that they change colour when the users mouse is hovering over them and then activates a function when the button is clicked.
Mouse	Class	Class built into pygame module that gives the position of the users mouse.
Click	Class	Class built into pygame module that detects when the user clicks the mouse.
Start	Function	Function that runs when the start button is clicked to make calculations and activate the animation.
blue	Class	Tuple class that stores the numbers representing blue in the RGBA colour values. Used for background of GUI window and buttons.
green	Class	Tuple class that stores the numbers representing green in the RGBA colour values. Used for representing object 1.
red	Class	Tuple class that stores the numbers representing red in the RGBA colour values. Used for representing object 2.

grey	Class	Tuple class that stores the numbers representing grey in the RGBA colour values. Used for displaying results of calculations made.
Small_font	Class	Font class built into the pygame module that will load and render fonts which I will use for the smaller text that will be displayed on the GUI.
Big_font	Class	Font class built into the pygame module that will load and render fonts which I will use for the bigger text that will be displayed on the GUI.
messages	List	Contains series of text that need to be displayed multiple times in the GUI.
Animation	Class	Class that displays the two objects in the GUI then when the start button is clicked makes the objects collide according to the calculations made in section 1.
distance1	Float	The distance object 1 has moved from its starting position.
distance2	Float	The distance object 2 has moved from its starting position.

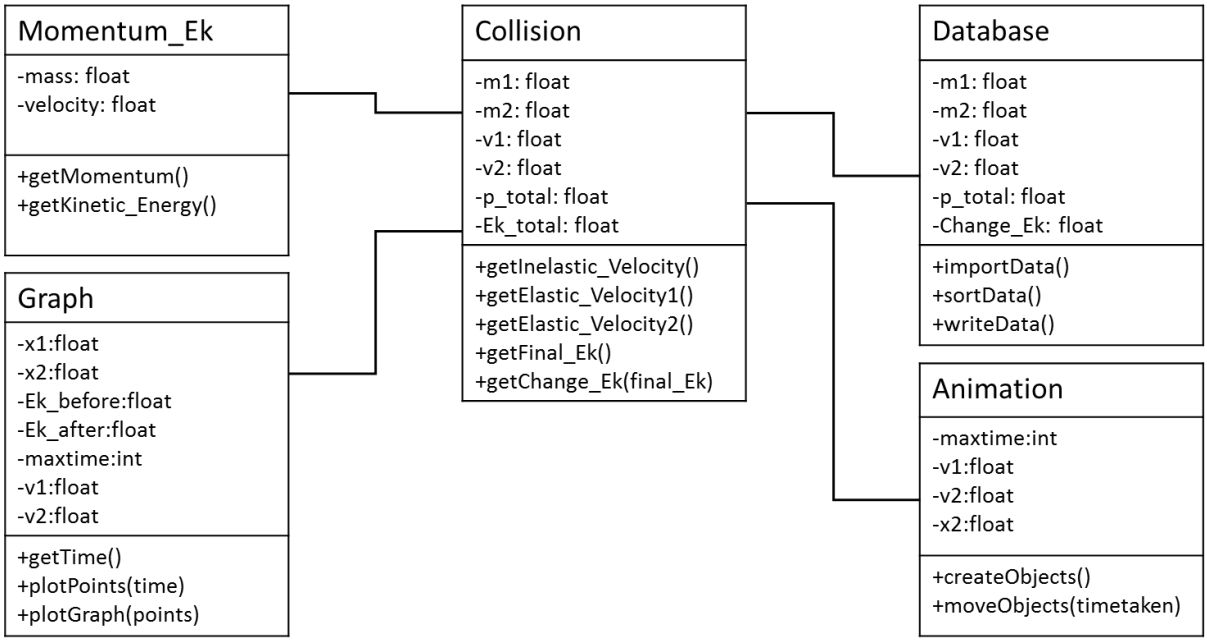
I chose these variable names because they are suitable for their function and sum up their purpose for example the 'Animation' class makes the objects move in the GUI. This will make it easier for me when I am developing my code and for people viewing or editing my code in the future.

Validation:

No further validation is needed in section 3 because the values displayed on the GUI will all have been calculated and validated in previous sections. I will need to make sure that this validation still works when users are entering values in the GUI instead of the python shell.

Class Diagram:

For my final program, the algorithms displayed earlier will be parts of a class since my project will be written using object oriented programming. This diagram shows the new class from section 3, its attributes, methods and the relationships between it and the classes from section 2.



Testing

Test Data:

The values in the input section of this table are the test data I will use to test that the function of a particular section of code is working so that all points on the success criteria are met. The 'output' and 'changes needed' column will be filled in during the testing stage of my development process and the first column shows which point on the success criteria this is testing.

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
5	Correct momentum calculated	m1 = 2 v1 = 3	Normal	Mometum: 6 kg m/s		
5	Correct momentum calculated	m2 = 1.4787 v2 = -4.678	Normal	Momentum: -6.92 kg m/s		
5	Correct kinetic energy calculated	m2 = 2 v2 = 3	Normal	Kinetic Energy: 9 J		
5	Correct kinetic energy calculated	m1 = 1.4787 v1 = -4.678	Normal	Kinetic Energy: 16.18 J		
5	Correct final velocity calculated for inelastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	Object 1 Final Velocity: 3 m/s Object 2 Final Velocity: 3 m/s		
5	Correct final velocity calculated for inelastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	Object 1 Final Velocity: 3.63 m/s Object 2 Final Velocity: 3.63 m/s		
5	Correct final velocities calculated for elastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	Object 1 Final Velocity: 1 m/s Object 2 Final Velocity: 9 m/s		
5	Correct final velocities calculated for elastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	Object 1 Final Velocity: 0.8 m/s Object 2 Final Velocity: 11.94 m/s		

Testing each point on success criteria:

1. Compare GUI displayed in window to GUI created in design section.
2. Click the buttons on the GUI to ensure this gives the correct outcome.
3. Compare data inputted to data displayed.
4. Select both options and click start to make sure they both work.
5. Use test table.
6. Click the buttons on the GUI to ensure this gives the correct outcome.
7. Start the animation of the collision for different values to make sure objects: move quicker for greater velocities, do not go over the edge of the window and stop after reaching the maximum time.
8. Regression testing.

Justification:

The values I put in the 'expected result' column of the test table I took from my test table for section 2 because the values should be the same except displayed in the GUI instead of the python shell. My test table only assesses point 5 on the success criteria since this is the only part that should result in specific data. This means that the rest of my success criteria will need to be tested by viewing the displayed GUI and comparing it to my design to ensure everything works.

Regression testing:

To adhere to point 8 on my success criteria for section 3 I will need to retest my algorithms from previous sections. To do this I will use further test data that I will get from the test table I designed in section 1 and 2 so that I can make sure everything still works and that the further developments I have made have not affected my algorithms from the previous section. The values that used to be printed in the python shell should now be displayed in the GUI.

Post Development Testing:

Section 3 will be the final section in the development of my project so after testing it no further development will be made so no more tests should be needed. However, if my stakeholder decides to use my program users may notice some faults that I did not discover during my testing which would mean that further development and testing are required to fix this.

Stakeholder Input:

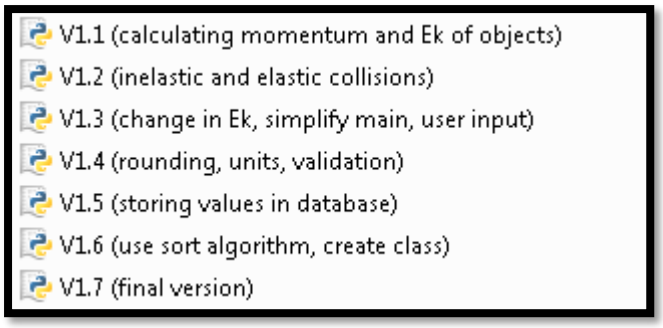
After developing section three of my program I plan to show it to Mr Hutchinson, my stakeholder, so that he can give some feedback since most of his suggestions in the interview referred to the GUI. This will allow him to make further suggestions, request any changes and to ensure that my project does everything he would like it to do since this should be the final version of my program.

Section 1 Development:

Calculating Momentum and Kinetic Energy

Development:

In my design for this section I planned the different parts of my success criteria I will cover in section 1 and broke them down into 7 different parts. To begin my development I created a folder for section 1 and within this a python file for each of these parts so that I know the order tasks need to be completed and to get an idea of how far along I am in the development process to ensure I don't run out of time:



Calculating momentum and kinetic energy is V1.1 and I began this part of my program by looking at the algorithm I created in my design section. In my algorithm I used a function to calculate the values however in my program I will use a class since a collision requires multiple objects. I called the class Momentum_Ek because it will calculate and return the momentum and kinetic energy of objects.

```
class Momentum_Ek(object): #Calculates the momentum and kinetic energy of a particular object using its mass and velocity

    #Constructor
    def __init__(self, mass, velocity):

        #Attributes
        self.mass = mass
        self.velocity = velocity

    #String method that will be called on print displaying mass and velocity of object and their units
    def __str__(self):
        report = "Mass: " + str(self.mass) + "\n"
        report = report + "Velocity: " + str(self.velocity)
        return report

    #Methods
    def getMass(self): #outputs the mass of this object
        return self.mass

    def getVelocity(self): #outputs the velocity of this object
        return self.velocity

    def getMomentum(self): #calculates and outputs the momentum of this object
        momentum = self.mass * self.velocity
        return momentum

    def getKinetic_Energy(self): #calculates and outputs the kinetic energy of this object
        Ek = 0.5 * self.mass * self.velocity**2
        return Ek
```

The getMomentum and getKinetic_Energy methods use the formulas I found during my research for their calculations.

I then wrote the main program that will use this class to create both objects and use the values returned to find the total momentum and kinetic energy of both objects. At this stage I do not need a user input so I will just declare the variables of velocity and mass.

```
## Main Program ##
m1 = 3
v1 = 5
print("Object 1")
Object1 = Momentum_Ek(m1,v1) #Created first object using Momentum_Ek class which calculates momentum and initial kinetic energy
p1 = Object1.getMomentum() #p used to represent momentum because it resebles the greek symbol 'rho' which is used to represent it in physics
Ek1 = Object1.getKinetic_Energy() #returns kinetic energy of first object before collision
print(Object1)
print("Momentum: ",p1)
print("Kinetic Energy: ",Ek1,"\n")

m2 = 1
v2 = -5
print("Object 2")
Object2 = Momentum_Ek(m2,v2)#Created second object using Momentum_Ek class which calculates momentum and initial kinetic energy
p2 = Object2.getMomentum() #p used to represent momentum because it resebles the greek symbol 'rho' which is used to represent it in physics
Ek2 = Object2.getKinetic_Energy()#returns kinetic energy of second object before collision
print(Object2)
print("Momentum: ",p2)
print("Kinetic Energy: ",Ek2,"\n")

p_total = p1 + p2 #Calculates total momentum of both objects
Ek_total = Ek1 + Ek2 #Calculates total kinetic energy of both objects
print("Total Momentum Before Collision: ",p_total)
print("Total Kinetic Energy Before Collision: ",Ek_total)


input("\n\nPress the enter key to exit.")
```

Review:

To test this part I will use the relevant section of the test table I created in my design section. I have not used any validation yet because momentum and kinetic energy can be any value.

```
Object 1
Mass: 2
Velocity: 3
Momentum:  6
Kinetic Energy:  9.0

Object 2
Mass: 1.4786
Velocity: -4.678
Momentum:  -6.916890799999999
Kinetic Energy:  16.178607581199998

Total Momentum Before Collision:  -0.9168907999999991
Total Kinetic Energy Before Collision:  25.178607581199998


Press the enter key to exit.
>>>
```

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
2	Correct momentum calculated	m1 = 2 v1 = 3	Normal	p1 = 6 kg m/s	p1= 6	Units
2	Correct momentum calculated	m2 = 1.4787 v2 = -4.678	Normal	p2 = -6.92 kg m/s	p2=-6.91689079	Rounding, units
2	Correct kinetic energy calculated	m2 = 2 v2 = 3	Normal	Ek2 = 9 J	Ek2= 9	Units

2	Correct kinetic energy calculated	m1 = 1.4787 v1 = -4.678	Normal	Ek1 = 16.18 J	Ek1= -16.178607	Rounding, units, put brackets around velocity because needs to be (-v) ² not -v² so that kinetic energy is not negative because that is not possible.

I will round the necessary values and add units after further development. The' getMass' and 'getVelocity' methods of the class are not needed so I will remove them since the mass and velocity inputted are static variables so will remain the same. Most of the main program consists of printing which will not be needed in the final version since the values will be displayed on a GUI.

Calculating Final Velocity

Development:

Calculating final velocity is V1.2 and I began this part of my program by looking at the algorithm I created in my design section and the equations for final velocity after elastic and inelastic collisions. I created a class called velocity that uses m1, v1, m2, v2 and p_total to calculate the final velocity. Different methods will be used depending on the collision type. Elastic collisions need two different methods because the final velocity is calculated differently depending on the object.

```
## Final Velocity Class ##
class Velocity(object): #Calculaates the final velocity of objects after elastic or inelastic collision

    #Constructor
    def __init__(self, m1, v1, m2, v2, p_total):

        #Attributes
        self.m1 = m1
        self.m2 = m2
        self.v1 = v1
        self.v2 = v2
        self.p_total = p_total

    #String method that will be called on print displaying the momentum before the collision
    def __str__(self):
        report = "Total Momentum Before Collision: " + str(self.p_total) + "\n"
        return report

    #Methods
    def getInelastic_Velocity(self): #Calculates final velocity of both objects after inelastic collision
        m_total = self.m1 + self.m2
        inelastic_v = p_total/m_total
        return inelastic_v

    def getElastic_Velocity1(self): #Calculates final velocity of first object after elastic collision
        m_total = self.m1 + self.m2
        elastic_v1 = ((self.m1-self.m2)*self.v1 + 2*self.m2*self.v2)/m_total
        return elastic_v1

    def getElastic_Velocity2(self): #Calculates final velocity of second object after elastic collision
        m_total = self.m1 + self.m2
        elastic_v2 = (2*self.m1*self.v1 - (self.m1-self.m2)*self.v2)/m_total
        return elastic_v2
```

I then further developed the main program to calculate the outcome of an elastic and inelastic collision using the set values and both the classes I have created so far. This is the section I added to the main program:

```
Collision = Velocity(m1, v1, m2, v2, p_total) #Creates object representing collision
inelastic_v = Collision.getInelastic_Velocity()#Calculates final velocity of object if it were inelastic
print(inelastic_v)
elastic_v1 = Collision.getElastic_Velocity1()#Calculates final velocity of object1 if it were elastic
print(elastic_v1)
elastic_v2 = Collision.getElastic_Velocity2()#Calculates final velocity of object2 if it were elastic
print(elastic_v2)
```

Review:

To test the correct final velocity is calculated I will use the relevant section of the test table I created in my design section. I have not used any validation yet because final velocity can be any value.

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
4	Correct final velocity calculated for inelastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	inelastic_v = 3 m/s	inelastic_v = 3	Units
4	Correct final velocity calculated for inelastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	inelastic_v = 3.63 m/s	inelastic_v = 3.6292562116	Units, Rounding

3	Correct final velocities calculated for elastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	elastic_v1 = 1 m/s elastic_v2 = 9 m/s	elastic_v1 = 1 elastic_v2 = 9	Units
3	Correct final velocities calculated for elastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	elastic_v1 = 0.8 m/s elastic_v2 = 11.94 m/s	elastic_v1 = 0.802112423 elastic_v2 = 11.93651242	Units, Rounding

I will round the necessary values and add units after further development because it is not needed in this stage of development. I also need to label some of the outputs so it is clear what they show. Other than minor errors like mistypes, no major errors occurred at this stage.

Calculating Change in Kinetic Energy

Development:

Calculating change in kinetic energy is V1.3 and I began this part of my program by looking at the algorithm I created in my design section. In my design I used a function but in my program I will create a class that works out the final kinetic energy and use this to work out the change in kinetic energy.

```
## Kinetic Energy Class ##
class Kinetic_Energy(object): #Calculaates the kinetic energy after the collision and the change in kinetic energy

    #Constructor
    def __init__(self, m1, v1, m2, v2, Ek_total):

        #Attributes
        self.m1 = m1
        self.m2 = m2
        self.v1 = v1
        self.v2 = v2
        self.Ek_total = Ek_total

    #String method that will be called on print displaying the kinetic energy before the collision
    def __str__(self):
        report = "Total Kinetic Energy Before Collision: " + str(self.Ek_total) + "\n"
        return report

    #Methods
    def getFinal_Ek(self): #Calculates final kinetic energy of both objects after collision
        object1_Ek = 0.5 * self.m1 * (self.v1)**2
        object2_Ek = 0.5 * self.m2 * (self.v2)**2
        final_Ek = object1_Ek + object2_Ek
        return final_Ek

    def getChange_Ek(self,final_Ek): #Calculates change in kinetic enery which requires kinetic energy after collision
        change_Ek = self.Ek_total - final_Ek
        return change_Ek
```

I then decided that since the Kinetic_Energy class uses the same attributes as the Velocity class I could combine them to one class called Collision that carries out the functions of both classes.

```
## Collision Class ##
class Collision(object): #Calculaates the final velocity, final kinetic energy and change in Ek of objects after elastic or inelastic

    #Constructor
    def __init__(self, m1, v1, m2, v2, p_total, Ek_total):

        #Attributes
        self.m1 = m1
        self.m2 = m2
        self.v1 = v1
        self.v2 = v2
        self.p_total = p_total
        self.Ek_total = Ek_total

    #String method that will be called on print displaying the momentum before the collision
    def __str__(self):
        report = "Total Momentum before Collision: " + str(self.p_total) + "\n"
        report = report + "Total Kinetic Energy before Collision: " + str(self.Ek_total) + "\n"
        return report

    #Methods
    def getInelastic_Velocity(self): #Calculates final velocity of both objects after inelastic collision
        m_total = self.m1 + self.m2
        inelastic_v = p_total/m_total
        return inelastic_v

    def getElastic_Velocity1(self): #Calculates final velocity of first object after elastic collision
        m_total = self.m1 + self.m2
        elastic_v1 = ((self.m1-self.m2)*self.v1 + 2*self.m2*self.v2)/m_total
        return elastic_v1

    def getElastic_Velocity2(self): #Calculates final velocity of second object after elastic collision
        m_total = self.m1 + self.m2
        elastic_v2 = (2*self.m1*self.v1 - (self.m1-self.m2)*self.v2)/m_total
        return elastic_v2

    def getFinal_Ek(self, final_v1, final_v2): #Calculates final kinetic energy of both objects after collision
        object1_Ek = 0.5 * self.m1 * (final_v1)**2
        object2_Ek = 0.5 * self.m2 * (final_v2)**2
        final_Ek = object1_Ek + object2_Ek
        return final_Ek

    def getChange_Ek(self,final_Ek): #Calculates change in kinetic enery which requires kinetic energy after collision
        change_Ek = self.Ek_total - final_Ek
        return change_Ek
```

I then further developed the main program to use this class to display the change in kinetic energy depending on the collision type and allowed the user to input values and choose the collision type. I decided to add user input now because in my design for section 1 I planned to write the code that will validate the user input so although in my final version values will be inputted via a GUI I needed a way to input data in this section.

```
## Main Program ##

print("----- MOMENTUM COLLISION SIMULATION -----") #Allows user to input values of mass and velocity for both objects
m1 = int(input("Mass of object 1: "))
v1 = int(input("Velocity of object 1: "))
m2 = int(input("Mass of object 2: "))
v2 = int(input("Velocity of object 2: "))
collision_type = input("Collision Type(elastic/inelastic): \n") #Allows user to select collision type

Object1 = Momentum_Ek(m1,v1)
p1 = Object1.getMomentum()
Ek1 = Object1.getKinetic_Energy()

Object2 = Momentum_Ek(m2,v2)
p2 = Object2.getMomentum()
Ek2 = Object2.getKinetic_Energy()

print("--Object 1--")
print(Object1,"\nMomentum: ",p1,"\nKinetic Energy: ",Ek1,"\n")

print("--Object 2--")
print(Object2,"\nMomentum: ",p2,"\nKinetic Energy: ",Ek2,"\n")

p_total = p1 + p2
Ek_total = Ek1 + Ek2
Collision = Collision(m1, v1, m2, v2, p_total, Ek_total) #Creates object representing collision
print(Collision) #Outputs total momentum and kinetic energy

if collision_type == 'inelastic':
    inelastic_v = Collision.getInelastic_Velocity() #Calculates final velocity after inelastic collision
    final_Ek = Collision.getFinal_Ek(inelastic_v, inelastic_v) #Calculates final kinetic energy after inelastic collision
    change_Ek = Collision.getChange_Ek(final_Ek) #Uses final kinetic energy to find change in kinetic energy
    print("Velocity After Collision: ",inelastic_v)
else:
    elastic_v1 = Collision.getElastic_Velocity1()#Calculates final velocity of object 1 after elastic collision
    elastic_v2 = Collision.getElastic_Velocity2()#Calculates final velocity of object 2 after elastic collision
    final_Ek = Collision.getFinal_Ek(elastic_v1, elastic_v2)#Calculates final kinetic energy after elastic collision
    change_Ek = Collision.getChange_Ek(final_Ek)#Uses final kinetic energy to find change in kinetic energy
    print("Velocity of Object 1 after Collision: ",elastic_v1,"\nVelocity of Object 2 after Collision: ",elastic_v2)

print("Kinetic Energy after Collision: ",final_Ek,"\nChange in Kinetic Energy: ",change_Ek)

input("\n\nPress the enter key to exit.")
```

Review:

To test this part I will use the relevant section of the test table I created in my design section. I will not use the invalid data tests yet because so far I have not added validation.

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 3
Velocity of object 1: 4
Mass of object 2: 2
Velocity of object 2: -1
Collision Type(elastic/inelastic):
inelastic
--Object 1--
Mass: 3
Initial Velocity: 4
Momentum:  12
Kinetic Energy:  24.0

--Object 2--
Mass: 2
Initial Velocity: -1
Momentum:  -2
Kinetic Energy:  1.0

Total Momentum before Collision: 10
Total Kinetic Energy before Collision: 25.0

Velocity After Collision:  2.0
Kinetic Energy after Collision:  10.0
Change in Kinetic Energy:  15.0


Press the enter key to exit.
```

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 3
Velocity of object 1: 4
Mass of object 2: 2
Velocity of object 2: -1
Collision Type(elastic/inelastic):
elastic
--Object 1--
Mass: 3
Initial Velocity: 4
Momentum:  12
Kinetic Energy:  24.0

--Object 2--
Mass: 2
Initial Velocity: -1
Momentum:  -2
Kinetic Energy:  1.0

Total Momentum before Collision: 10
Total Kinetic Energy before Collision: 25.0

Velocity of Object 1 after Collision:  0.0
Velocity of Object 2 after Collision:  5.0
Kinetic Energy after Collision:  25.0
Change in Kinetic Energy:  0.0


Press the enter key to exit.
```

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
2	Calculating change in kinetic energy for elastic collision.	m1 = 3 v1 = 4 m2 = 2 v2 = -2	Normal	Change in Ek =0 J	Change in Ek = -3.552e-15	For elastic collision this should always be 0, need to round, add units.
2	Calculating change in kinetic energy for inelastic collision.	m1 = 3 v1 = 4 m2 = 2 v2 = -2	Normal	Change in Ek =15 J	Change in Ek =15	Add units

I will add units and round the calculated values at the end of this section of development because these features are not needed until then.

Validation

Development:

Validation of the user input is required because certain variables can only be certain values for the algorithms to work. I explained the reasons for the validation of these variables in my design for section 1. I decided to use a function to validate the user input because there are multiple inputs that require the same validation so this will avoid repeating code.

```
print("----- MOMENTUM COLLISION SIMULATION -----")
def Validation(message,value): #Function that validates the user inputs,
                               #function recieves the message to be displayed and the type of data being inputted
    while True:
        try: #Allows user to try to input a value after the displayed message
            userInput = float(input(message)) #Input must be a float because velocity and mass don't need to be whole numbers
        except ValueError: #If the value entered is not a float a ValueError will occur so a statement will be printed and the loop repeated
            print("This is not a number\n")
            continue
        if value == 'm' and userInput <= 0: #If the value being inputted is a mass it needs to be greater than
            print("Mass must be greater than 0\n") #needs to be greater than 0 because mass can't be less than 0
            continue
        if value == 'v1' and userInput < 0: #If the value being inputted is v1 it needs to be greater or equal to 0
            print("Velocity of object 1 must be greater than or equal to 0\n")#So always moving in one direction (velocity is vector)
            continue
        if value == 'v2' and userInput > 0: #If the value being inputted is v2 it needs to be smaller or equal to 0
            print("Velocity of object 2 must be greater than or equal to 0\n") #So moving towards other object so collision occurs
            continue
        else:
            return userInput
            break

m1 = Validation("Mass of object 1: ","m")
v1 = Validation("Velocity of object 1: ","v1")
m2 = Validation("Mass of object 2: ","m")
v2 = Validation("Velocity of object 2: ","v2") #Runs validation function
```

Review:

To test that the validation works and allows users to enter data again if they entered it incorrectly I will use the relevant sections of the test table I made in my design section.

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 0
Mass must be greater than 0

Mass of object 1: 0.001
Velocity of object 1: -1
Velocity of object 1 must be greater than
or equal to 0

Velocity of object 1: ten
This is not a number

Velocity of object 1: 0
Mass of object 2: -5
Mass must be greater than 0

Mass of object 2: five
This is not a number

Mass of object 2: 5g
This is not a number

Mass of object 2: 5
Velocity of object 2: -1
Collision Type(elastic/inelastic):
```

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 5
Velocity of object 1: 1
Mass of object 2: 5
Velocity of object 2: 1
Velocity of object 2 must be greater than
or equal to 0

Velocity of object 2: 10
Velocity of object 2 must be greater than
or equal to 0

Velocity of object 2: 10m/s
This is not a number

Velocity of object 2: 0
Collision Type(elastic/inelastic):
```

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
1	Mass Inputted is > 0	m1 = 0 m2 = 0	Boundary	Invalid Mass Input Mass:	Mass must be greater than 0, input mass	None
1	Mass Inputted is > 0	m1 = 0.001 m2 = 0.001	Boundary	Valid Mass	Valid	None
1	Mass Inputted is > 0	m1 = -5 m2 = -5	Error	Invalid Mass Input Mass:	Mass must be greater than 0	None
1	Mass Inputted is > 0	m1 = 5 m2 = 5	Normal	Valid Mass	Valid	None
1	Mass inputted is float	m1 = five m2 = 5g	Error	Invalid Mass Input Mass:	This is not a number, input mass	None
1	v1 ≥ 0	v1 = 0	Boundary	Valid Velocity	Valid	None
1	v1 ≥ 0	v1 = 1	Boundary	Valid Velocity	Valid	None
1	v1 ≥ 0	v1 = -1	Boundary	Invalid Velocity Input v1:	Must be greater or equal to 0, input v1	None
1	v1 ≥ 0	v1 = 10	Normal	Valid Velocity	Valid	None
1	v1 ≥ 0	v1 = -10	Error	Invalid Velocity Input v1:	Must be greater or equal to 0, input v1	None
1	v1 is a float	v1 = ten	Error	Invalid Velocity Input v1:	This is not a number, input v1	None
1	v2 ≤ 0	v2 = 0	Boundary	Valid Velocity	Valid	None
1	v2 ≤ 0	v2 = -1	Boundary	Valid Velocity	Valid	None

1	v2 ≤ 0	v2 = 1	Boundary	Invalid Velocity Input v2:	Must be less than or equal to 0, input v2	None
1	v2 ≤ 0	v2 = -10	Normal	Valid Velocity	Valid	None
1	v2 ≤ 0	v2 = 10	Error	Invalid Velocity Input v2:	Must be less than or equal to 0, input v2	None
1	v2 is a float	v2 = 10m/s	Error	Invalid Velocity Input v2:	This is not a number, input v2	None

The collision type does not need to be validated because in the final version the user will only be able to select only one of these options in the GUI and in my current program no errors occur if neither of these options is chosen it just defaults to an elastic collision.

I also tested entering the velocity of both objects as 0 since this means the objects never collide; the output for everything was 0 for both collision types so validation is not needed for this since no errors occur.

Database

Development:

I created a CSV file using excel which I justified in my analysis and design for this section. I named the file database so its purpose is clear and put the headings for each column in the first row so users know what each part represents. I then looked at the algorithm I created in my design section for the database and used this to help me write my program.

```
def Database(m1,v1,m2,v2,p_total,change_Ek): #Function that reads the current infomation from the database and writes new values to it
    try:
       import csv
       with open("database.csv", 'rt') as text_file: #opens text file
          reader = csv.reader(text_file)
          aList = list(reader) #puts data from database into list
    except FileNotFoundError: #If an error occurs a message is displayed to tell the user how to prevent it then ends the program
       print("Please make sure the CSV file is not open and has not been deleted.")
       quit()
    values = [str(m1),str(v1),(str(m2),str(v2),str(p_total),str(change_Ek)] #creates list containing the new values
    aList.append(values) #appends new values to the current list
    print(aList)
    try:
        import csv
        with open("database.csv", 'w', newline='') as csv_file: #opens the text file
            writer = csv.writer(csv_file, delimiter=',')
            writer.writerows(aList) #writes the list to the file
            csv_file.close()
    except PermissionError: #If an error occurs a message is displayed to tell the user how to prevent it then end the program
        print("The data could not be added to the database.")
        print("Please make sure the 'database' file is closed and re-start the program.")
        quit()
```

During development when I run the program errors occurred if I had the CSV file open so I used try and except to stop this from happening. This function reads the current information from the database and writes new data to it. I updated the main program to use this function when the collision type is inelastic.

```
if collision_type == 'inelastic': #Prints calculations of inelastic collisions and stores info in database
    inelastic_v = Collision.getInelastic_Velocity()
    final_Ek = Collision.getFinal_Ek(inelastic_v, inelastic_v)
    change_Ek = Collision.getChange_Ek(final_Ek)
    print("Velocity After Collision: ",inelastic_v,"m/s")
    Database(m1,v1,m2,v2,p_total,change_Ek) #Runs database function
```

Review:

To test that the database works I opened the CSV file and compared the data stored in it to the data outputted in the python shell and it was correct.

	A	B	C	D	E	F
1	Mass 1	Velocity 1	Mass 2	Velocity 2	Momentum	Change in Ek
2	3	5	1	-3	12	24
3	50	100	14	0	5000	250000
4	1	2	3	-4	-10	13.5
5	5	6	3	0	30	33.75
6	1	5	7.6	-3	-17.8	28.27
7	5	8.4356	9.5453	-4.5	-0.77	274.53
8	4	5	3	-8.3	-4.9	151.63
9	3	5	1	-3	12	24

Final Section 1 Version

Final Development and Testing:

To finish development of section 1 of my program I needed to add the correct units to the outputted values and round them to two decimal places in order to adhere to all points on my success criteria for this section:

1. Program must run successfully using Python 3.3.2 without errors occurring.
2. Algorithms written using object oriented programming that work out the correct momentum and kinetic energy.
3. Algorithm working out the final velocities after an elastic collision.
4. Algorithm working out the final velocity after an inelastic collision.
5. Final velocity, total momentum and change in kinetic energy rounded to two decimal places.
6. SI units (kg, m/s, kgm/s and J) used for velocity, mass, momentum and energy.
7. Database that stores the velocities and masses and the calculated total momentum and change in kinetic energy for an inelastic collision.

I labelled on my code where each point on the success criteria has been met.

```
## Mommentum & Kinetic Energy CLass ##
class Momentum_Ek(object): #Calculaates the momentum and kinetic energy of a particular object using its
                           #mass and velocity

    #Constructor
    def __init__(self, mass, velocity):

        #Attributes
        self.mass = mass
        self.velocity = velocity

    #String method that will be called on print displaying mass and velocity of object and their units
    def __str__(self):
        report = "Mass: " + str(self.mass) + " kg\n"
        report = report + "Initial Velocity: " + str(self.velocity) + " m/s"
        return report

    #Methods
    def getMomentum(self): #Calculates and outputs the momentum of this object
        momentum = round(self.mass * self.velocity, 2) #rounds answer to two decimal places
        return momentum

    def getKinetic_Energy(self): #Calculates and outputs the kinetic energy of this object
        Ek = round(0.5 * self.mass * (self.velocity)**2, 2) #Rounds answer to two decimal places
        return Ek

## Collision Class ##
class Collision(object): #Calculaates the final velocity, final kinetic energy and change in Ek of objects
                         #after elastic or inelastic collision

    #Constructor
    def __init__(self, m1, v1, m2, v2, p_total, Ek_total):

        #Attributes
        self.m1 = m1
        self.m2 = m2
        self.v1 = v1
        self.v2 = v2
        self.p_total = p_total
        self.Ek_total = Ek_total

    #String method that will be called on print displaying the momentum before the collision and their units
    def __str__(self):
        report = "Total Momentum before Collision: " + str(self.p_total) + " kg m/s \n"
        report = report + "Total Kinetic Energy before Collision: " + str(self.Ek_total) + " J \n"
        return report

    #Methods
    def getInelastic_Velocity(self): #Calculates final velocity of both objects after inelastic collision
        m_total = self.m1 + self.m2 #Calculates total mass
        inelastic_v = round(p_total/m_total, 2)#Uses total momentum and total mass to calculate final v,
                                               #rounds to two decimal places
        return inelastic_v

    def getElastic_Velocity1(self): #Calculates final velocity of first object after elastic collision
        m_total = self.m1 + self.m2 #Calculates total mass
        elastic_v1 = ((self.m1-self.m2)*self.v1 + 2*self.m2*self.v2)/m_total #uses the formula I researched
        elastic_v1 = round(elastic_v1, 2) #Rounds to two decimal places
        return elastic_v1

    def getElastic_Velocity2(self): #Calculates final velocity of second object after elastic collision
        m_total = self.m1 + self.m2 #Calculates total mass
        elastic_v2 = (2*self.m1*self.v1 - (self.m1-self.m2)*self.v2)/m_total #uses formula I researched
        elastic_v2 = round(elastic_v2, 2) #Rounds to two decimal places
        return elastic_v2

    def getFinal_Ek(self, final_v1, final_v2): #Calculates final kinetic energy of both objects after collision
        object1_Ek = 0.5 * self.m1 * (final_v1)**2
        object2_Ek = 0.5 * self.m2 * (final_v2)**2 #For inelastic collision both objects will have the same Ek
        final_Ek = round(object1_Ek + object2_Ek, 2) #Adds kinetic energies of both objects together to find total final Ek,
                                                     #rounds to two decimal places
        return final_Ek

    def getChange_Ek(self,final_Ek): #Calculates change in kinetic enery which requires total kinetic energy after collision
        change_Ek = round(self.Ek_total - final_Ek, 2) #Round change in kinetic energy to two decimal places
        return change_Ek
```

2

4

3

5

```
print("----- MOMENTUM COLLISION SIMULATION -----")
def Validation(message,value): #Function that validates the user inputs,
                               #function recieves the message to be displayed and the type of data being inputted
    while True:
        try: #Allows user to try to input a value after the displayed message
            userInput = float(input(message)) #Input must be a float because velocity and mass don't need to be whole numbers
        except ValueError: #If the value entered is not a float a ValueError will occur so a statement will be printed and the loop repeated
            print("This is not a number\n")
            continue
        if value == 'm' and userInput <= 0: #If the value being inputted is a mass it needs to be greater than
            print("Mass must be greater than 0\n") #needs to be greater than 0 because mass can't be less than 0
            continue
        if value == 'v1' and userInput < 0: #If the value being inputted is v1 it needs to be greater or equal to 0
            print("Velocity of object 1 must be greater than or equal to 0\n")#So always moving in one direction (velocity is vector)
            continue
        if value == 'v2' and userInput > 0: #If the value being inputted is v2 it needs to be smaller or equal to 0
            print("Velocity of object 2 must be greater than or equal to 0\n") #So moving towards other object so collision occurs
            continue
        else:
            return userInput
            break

def Database(m1,v1,m2,v2,p_total,change_Ek): #Function that reads the current infomation from the database and writes new values to it
    try:
       import csv
       with open("database.csv", 'rt') as text_file: #opens text file
          reader = csv.reader(text_file)
          aList = list(reader) #puts data from database into list
    except FileNotFoundError: #If an error occurs a message is displayed to tell the user how to prevent it then ends the program
       print("Please make sure the CSV file is not open and has not been deleted.")
       quit()
    values = [str(m1),str(v1),str(m2),str(v2),str(p_total),str(change_Ek)] #creates list containing the new values
    aList.append(values) #appends new values to the current list
    try:
        import csv
        with open("database.csv", 'w', newline='') as csv_file: #opens the text file
            writer = csv.writer(csv_file, delimiter=',')
            writer.writerows(aList) #writes the list to the file
            csv_file.close()
    except PermissionError: #If an error occurs a message is displayed to tell the user how to prevent it then end the program
        print("The data could not be added to the database.")
        print("Please make sure the 'database' file is closed and re-start the program.")
        quit()

m1 = Validation("Mass of object 1: ","m")
v1 = Validation("Velocity of object 1: ","v")
m2 = Validation("Mass of object 2: ","m")
v2 = Validation("Velocity of object 2: ","v") #Runs validation function

collision_type = input("Collision Type(elastic/inelastic): \n") #Determines the type of collision

Object1 = Momentum_Ek(m1,v1) #Created first object using Momentum_Ek class which calculates momentum and initial kinetic energy
p1 = Object1.getMomentum() #p used to represent momentum because it resebles the greek symbol 'rho' which is used to represent it in physics
Ek1 = Object1.getKinetic_Energy() #returns kinetic energy of first object before collision

Object2 = Momentum_Ek(m2,v2)#Created second object using Momentum_Ek class which calculates momentum and initial kinetic energy
p2 = Object2.getMomentum() #p used to represent momentum because it resebles the greek symbol 'rho' which is used to represent it in physics
Ek2 = Object2.getKinetic_Energy()#returns kinetic energy of second object before collision

print("--Object 1--")
print(Object1,"\nMomentum: ",p1,"kg m/s\nKinetic Energy: ",Ek1,"J\n") #Prints values calculated and their units

print("--Object 2--")
print(Object2,"\nMomentum: ",p2,"kg m/s\nKinetic Energy: ",Ek2,"J\n") #Prints values calculated and their units

p_total = round(p1 + p2, 2) #Total momentum of both objects rounded to two decimal places
Ek_total = round(Ek1 + Ek2, 2) #Total initial kinetic energy of both objects rounded to two decimal places
Collision = Collision(m1, v1, m2, v2, p_total, Ek_total) #Creates object using Collision class that calculates the final velocities + Change_Ek
print(Collision) #Uses string method of collision class to print total momentum and kinetic energy before the collision

if collision_type == 'inelastic': #Prints calculations of inelastic collisions and stores info in database
    inelastic_v = Collision.getInelastic_Velocity()
    final_Ek = Collision.getFinal_Ek(inelastic_v, inelastic_v)
    change_Ek = Collision.getChange_Ek(final_Ek)
    print("Velocity After Collision: ",inelastic_v,"m/s")
    Database(m1,v1,m2,v2,p_total,change_Ek) #Runs database function
else: #Prints calculation of elastic collisions
    elastic_v1 = Collision.getElastic_Velocity1()
    elastic_v2 = Collision.getElastic_Velocity2()
    final_Ek = Collision.getFinal_Ek(elastic_v1, elastic_v2)
    change_Ek = Collision.getChange_Ek(final_Ek)
    print("Velocity of Object 1 after Collision: ",elastic_v1,"m/s\nVelocity of Object 2 after Collision: ",elastic_v2,"m/s")

print("Kinetic Energy after Collision: ",final_Ek,"J\nChange in Kinetic Energy: ",change_Ek,"J") #Prints values calculated and their units

input("\n\nPress the enter key to exit.") #Allows user to end the program smoothly by pressing the enter key
```

1

1

7

4

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 4.345
Velocity of object 1: 6.4564
Mass of object 2: 1.4787
Velocity of object 2: -4.678
Collision Type(elastic/inelastic):
inelastic
--Object 1--
Mass: 4.345 kg
Initial Velocity: 6.4564 m/s
Momentum:  28.05 kg m/s
Kinetic Energy:  90.56 J

--Object 2--
Mass: 1.4787 kg
Initial Velocity: -4.678 m/s
Momentum:  -6.92 kg m/s
Kinetic Energy:  16.18 J

Total Momentum before Collision: 21.13 kg m/s
Total Kinetic Energy before Collision: 106.74 J

Velocity After Collision:  3.63 m/s
Kinetic Energy after Collision:  38.37 J
Change in Kinetic Energy:  68.37 J


Press the enter key to exit.
```

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 4.345
Velocity of object 1: 6.4564
Mass of object 2: 1.4787
Velocity of object 2: -4.678
Collision Type(elastic/inelastic):
elastic
--Object 1--
Mass: 4.345 kg
Initial Velocity: 6.4564 m/s
Momentum:  28.05 kg m/s
Kinetic Energy:  90.56 J

--Object 2--
Mass: 1.4787 kg
Initial Velocity: -4.678 m/s
Momentum:  -6.92 kg m/s
Kinetic Energy:  16.18 J

Total Momentum before Collision: 21.13 kg m/s
Total Kinetic Energy before Collision: 106.74 J

Velocity of Object 1 after Collision:  0.8 m/s
Velocity of Object 2 after Collision:  11.94 m/s
Kinetic Energy after Collision:  106.79 J
Change in Kinetic Energy:  -0.05 J


Press the enter key to exit.
```

I will use the test table to test that everything on the success criteria still works after further development and that all the errors that occurred in previous tests have been fixed. I will not test validation again because I have not developed it since the last test because everything worked.

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
2	Correct momentum calculated	m1 = 2 v1 = 3	Normal	p1 = 6 kg m/s	p1 = 6 kg m/s	None
2	Correct momentum calculated	m2 = 1.4787 v2 = -4.678	Normal	p2 = -6.92 kg m/s	p2 = -6.92 kg m/s	None
2	Correct kinetic energy calculated	m2 = 2 v2 = 3	Normal	Ek2 = 9 J	Ek2 = 9 J	None
2	Correct kinetic energy calculated	m1 = 1.4787 v1 = -4.678	Normal	Ek1 = 16.18 J	Ek1 = 16.18 J	None
4	Correct final velocity calculated for inelastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	inelastic_v = 3 m/s	inelastic_v = 3 m/s	None
4	Correct final velocity calculated for inelastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	inelastic_v = 3.63 m/s	inelastic_v = 3.63 m/s	None
3	Correct final velocities calculated for elastic	m1 = 3 v1 = 5 m2 = 1 v2 = -3	Normal	elastic_v1 = 1 m/s elastic_v2 = 9 m/s	elastic_v1 = 1 m/s elastic_v2 = 9 m/s	None
3	Correct final velocities calculated for elastic	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678	Normal	elastic_v1 = 0.8 m/s elastic_v2 = 11.94 m/s	elastic_v1 = 0.8 m/s elastic_v2 = 11.94 m/s	None

Stakeholder Testing:

I do not require much stakeholder input at this stage in my development because most of Mr Hutchinson's requirements focus on the GUI however I showed him my program so far to ensure the algorithms perform the correct calculations and he confirmed that they do.

Review Summary:

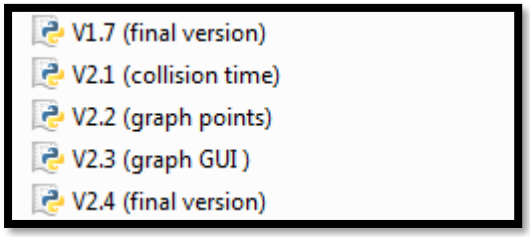
For the most part my development for this section closely followed my design with very few changes. For the main algorithms in my program, the process is the same expect for the addition of some new variables and minor changes. I also changed the database class to a function because there was no need for multiple methods. I changed some variable names to better suit their purpose and added some different test data to my test table when I discovered other aspects of my program that needed testing. After testing my final version for section 1 against the success criteria to ensure that everything works and making sure it includes the usability features I mentioned in my design, I am now ready to move onto developing section 2 of my project.

Section 2 Development:

Calculating Time Taken

Development:

In my design for this section I planned the different parts of my success criteria I will cover and broke them down into 4 different parts. To begin my development I created a folder for section 2 and within this a python file for each of these sub versions so that I know the order tasks need to be completed and to get an idea of how far along I am in the development process to ensure I don't run out of time:



To begin my development for this section I copied my final code from section 1 into V2.1 then looked at the algorithm I created for collision time in my design section. I decided to make this algorithm part of a class called 'Graph' with the method 'getTime' because this will make the software easier to maintain and provides a clear modular structure. In order to calculate the time taken for the objects to collide I needed to make them a set distance apart.

```
class Graph(object): #Creates a graph of kinetic energy against time

    #Constructor
    def __init__(self, x1, x2, Ek_before, Ek_after, v1, v2):

        #Attributes
        self.x1 = x1
        self.x2 = x2
        self.Ek_before = Ek_before
        self.Ek_after = Ek_after
        self.v1 = v1
        self.v2 = v2

    #Methods
    def getTime(self): #Calculates the time taken for the objects to collide
        time_taken = 0
        while self.x2 >= self.x1:
            self.x1 = self.x1 + self.v1
            self.x2 = self.x2 - self.v2
            time_taken = time_taken + 1
        return time_taken
```

```
x1 = 0.0 #initial positon of object 1
x2 = 500.0 #initial position of object 2
```

After testing this program I found that some values of v1 and v2 never collide or take too long to collide so I added 'maxtime' as an attribute of this class to validate this and prevent errors from occurring. I set the maximum time to 1000 seconds because this is enough time for most collisions of this distance apart. This is the updated code for the 'getTime' method including this validation:

```
def getTime(self): #Calculates the time taken for the objects to collide
    time_taken = 0
    while self.x2 >= self.x1 and time_taken < self.maxtime:#detects collision and limits time
        self.x1 = self.x1 + self.v1 #each sec the object moves a certain distance (it's velocity)
        self.x2 = self.x2 + self.v2
        time_taken = time_taken + 1 #increase time taken as objects move
    return time_taken
```

I also changed 'self.x2 = self.x2 - self.v2' to 'self.x2 = self.x2 + self.v2' because self.v2 is already always negative. The object moves by the value of its velocity in meters each second because the unit of its velocity is meters per second (m/s).

Review:

To test that the time taken for the objects to collide is calculated correctly I will use the relevant section of the test table I created in my design section.

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 23
Velocity of object 1: 5
Mass of object 2: 12
Velocity of object 2: -3
Collision Type(elastic/inelastic):
elastic
--Object 1--
Mass: 23.0 kg
Initial Velocity: 5.0 m/s
Momentum:  115.0 kg m/s
Kinetic Energy:  287.5 J

--Object 2--
Mass: 12.0 kg
Initial Velocity: -3.0 m/s
Momentum:  -36.0 kg m/s
Kinetic Energy:  54.0 J

Distance between objects:  500.0 m

Total Momentum before Collision: 79.0 kg m/s
Total Kinetic Energy before Collision: 341.5 J

Velocity of Object 1 after Collision:  -0.49 m/s
Velocity of Object 2 after Collision:  7.51 m/s
Kinetic Energy after Collision:  341.16 J
Change in Kinetic Energy:  0.34 J
63 s

Press the enter key to exit.
```

← Time taken

I decided to change the set the distance between the objects to 1000 meters because the purpose of this section is to display a graph of kinetic energy against time to demonstrate the difference between collision types. I increased the value so that the collision would take longer meaning the graph will look less skewed. I changed some of the expected results in my test table to accommodate for these changes. I also temporarily removed some of the irrelevant print statements to make testing more efficient.

```
----- MOMENTUM COLLISION SIMULATION
Mass of object 1: 10
Velocity of object 1: 5
Mass of object 2: 10
Velocity of object 2: -3
Collision Type(elastic/inelastic):
elastic
Distance between objects:  1000.0 m

Time Taken: 126 s


Press the enter key to exit.
```

```
----- MOMENTUM COLLISION SIMULATION
Mass of object 1: 5
Velocity of object 1: 0
Mass of object 2: 5
Velocity of object 2: 0
Collision Type(elastic/inelastic):
inelastic
Distance between objects:  1000.0 m

Time Taken: 1000 s


Press the enter key to exit.
```

```
----- MOMENTUM COLLISION SIMULATION
Mass of object 1: 100
Velocity of object 1: 0.5
Mass of object 2: 100
Velocity of object 2: -0.5
Collision Type(elastic/inelastic):
inelastic
Distance between objects:  1000.0 m

Time Taken: 1000 s


Press the enter key to exit.
```

```
----- MOMENTUM COLLISION SIMULATION
Mass of object 1: 50
Velocity of object 1: 0.4
Mass of object 2: 50
Velocity of object 2: -0.5
Collision Type(elastic/inelastic):
elastic
Distance between objects:  1000.0 m

Time Taken: 1000 s


Press the enter key to exit.
```

The collision type and mass do not affect the time of the collision so I made these any valid value.

SC:	Function:	Input/ Test:	Test Type:	Expected Result:	Output:	Changes Needed:
1	Maximum time < 1000	v1 = 5 v2 = -3	Normal	Time taken = 126	Time taken: 126 s	None
1	Maximum time < 1000	v1 = 0 v2 = 0	Error	Time>1000 so set Time taken = 1000	Time taken: 1000 s	Tell user when time has been set to 1000s.
1	Maximum time < 1000	v1 = 0.5 v2 = -0.5	Boundary	Time taken = 1000	Time taken: 1000 s	None
1	Maximum time < 1000	v1 = 0.4 v2 = -0.5	Boundary	Time>1000 so set Time taken = 1000	Time taken: 1000 s	Tell user when time has been set to 1000s.

Points on Graph

Development:

Finding the coordinates for the points that will be plotted on my graph is V2.2. I began this part of my program by looking at the algorithm I created in my design section. I have decided to include this algorithm in the graph class as a new method called plotPoints.

New method in graph class:

```
def plotPoints(self,time): #needs time taken to know when Ek changes
    points = [] #creates list that will store the points
    for i in range (0,time+1):
        point = [i,self.Ek_before] #[x = time, y = Ek]
        points.append(point)
    for i in range (time+1,self.maxtime): #after collision
        point = [i,self.Ek_after] #[x = time, y = Ek]
        points.append(point)
    return points
```

I developed this method by following the algorithm I designed in pseudo code. I originally planned to make the scale on the x axis 10 so record the kinetic energy every 10 seconds. I realised that this would produce a graph that was much less precise than if the time was taken every second so I changed it to do this. This will take longer but the computer is fast enough that it will not make a noticeable difference to my program.

Part of main program that uses new method:

```
graph = Graph(x1, x2, Ek_total, final_Ek, maxtime, v1, v2)
time_taken = graph.getTime()
points = graph.plotPoints(time_taken)
```

Review:

To test that the list of points returned by the method is correct I will use a print statement to display them and use the relevant section of the test table I created in my design section. The list of points is too long to display on this document so for each test I will show the start and end of the list and the points where the object collides. I also temporarily removed some of the irrelevant print statements.

```
Mass of object 1: 4.345
Velocity of object 1: 6.4564
Mass of object 2: 1.4787
Velocity of object 2: -4.678
Collision Type(elastic/inelastic):
inelastic
Total Momentum before Collision: 21.13 kg m/s
Total Kinetic Energy before Collision: 106.74 J

Velocity After Collision:  3.63 m/s
Kinetic Energy after Collision:  38.37 J
Change in Kinetic Energy:  68.37 J
Time:  90 s
```

Start of list of points:

```
[[0, 106.74], [1, 106.74],
```

Time objects collide:

```
[90, 106.74], [91, 38.37],
```

End of list of points:

```
[998, 38.37], [999, 38.37]]
```

During testing I found that the list of points only goes up to 999 even though the maximum time is correctly validated as 1000. This is because the range starts at 0 so I changed 'self.maxtime' to 'self.maxtime +1' so that the points on the graph go up to 1000 seconds meaning there are 1001 points from 0 to 1000.

```
Mass of object 1: 4.345
Velocity of object 1: 6.4564
Mass of object 2: 1.4787
Velocity of object 2: -4.678
Collision Type(elastic/inelastic):
inelastic
Total Momentum before Collision: 21.13 kg m/s
Total Kinetic Energy before Collision: 106.74 J

Velocity After Collision:  3.63 m/s
Kinetic Energy after Collision:  38.37 J
Change in Kinetic Energy:  68.37 J
Time:  90 s
```

Start of list of points:

```
[[0, 106.74], [1, 106.74],
```

Time objects collide:

```
[90, 106.74], [91, 38.37],
```

End of list of points:

```
[999, 38.37], [1000, 38.37]]
```

```
Mass of object 1: 3
Velocity of object 1: 5
Mass of object 2: 1
Velocity of object 2: -3
Collision Type(elastic/inelastic):
elastic
Total Momentum before Collision: 12.0 kg m/s
Total Kinetic Energy before Collision: 42.0 J

Velocity of Object 1 after Collision:  1.0 m/s
Velocity of Object 2 after Collision:  9.0 m/s
Kinetic Energy after Collision:  42.0 J
Change in Kinetic Energy:  0.0 J
Time:  126 s
```

Start of list of points:

```
[[0, 42.0], [1, 42.0],
```

Time objects collide:

```
[126, 42.0], [127, 42.0],
```

End of list of points:

```
[999, 42.0], [1000, 42.0]]
```

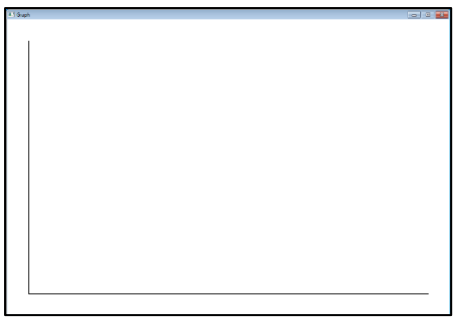
SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
2	Correct points in list of points	m1 = 4.345 v1 = 6.4564 m2 = 1.4787 v2 = -4.678 Type: Inelastic	Normal	Time taken = 90 List of cords [x,y], x starts at 106.74 then when y=time taken x changes to 38.37.	Time: 90 s	Display points on GUI not in list on python shell.
2	Correct points in list of points	m1 = 3 v1 = 5 m2 = 1 v2 = -3 Type: Elastic	Normal	Time taken = 126 List of cords [x,y], x stays at 42, y increases from 0 to 1000	Time: 126 s	Display points on GUI not in list on python shell.

Creating GUI

Development:

To begin creating the GUI for the graph I made a new python file to design the template of a graph using set variables instead of the data calculated by the rest of my program. I did this to make it easier to test the graph with certain values without interfering with other parts of my program. To create the GUI I imported pygame and created a window. I made a while loop to show the position of my mouse on the GUI to help me decide where to place certain elements of my graph. I decided to make the gap between the axis and window 50 so the resolution width = 1050 because maxtime is 1000 and axis is 50 from edge.

```
## GRAPH ##
import pygame #imports pygame
pygame.init() #initiates pygame
white = (255,255,255) #defines the colours
black = (0,0,0)
blue = (0,0,255)

Ek_before = 600 #will be highest possible value of Ek
graphDisplay = pygame.display.set_mode((1050,Ek_before+100)) #creates window of set resolution, x, y, 100 (50*2)
graphDisplay.fill(white) #makes background white
pygame.display.set_caption('Graph') #titles window
                        #points of end of line  #width
pygame.draw.line(graphDisplay, black, (50,Ek_before+50), (1000,Ek_before+50),2)#colour and position of x axis
pygame.draw.line(graphDisplay, black, (50,Ek_before+50), (50,50),2)#colour and position of y axis

while True: #shows position of mouse
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        print(event)
    pygame.display.update()
```



```
<Event(4-MouseMotion {'pos': (283, 9), 'rel': (0, -2),
'buttons': (0, 0, 0)})>
<Event(4-MouseMotion {'pos': (283, 7), 'rel': (0, -2),
'buttons': (0, 0, 0)})>
<Event(4-MouseMotion {'pos': (283, 6), 'rel': (0, -1),
'buttons': (0, 0, 0)})>
<Event(4-MouseMotion {'pos': (283, 5), 'rel': (0, -1),
'buttons': (0, 0, 0)})>
<Event(4-MouseMotion {'pos': (283, 4), 'rel': (0, -1),
'buttons': (0, 0, 0)})>
<Event(4-MouseMotion {'pos': (283, 3), 'rel': (0, -1),
'buttons': (0, 0, 0)})>
<Event(4-MouseMotion {'pos': (283, 0), 'rel': (0, -3),
'buttons': (0, 0, 0)})>
```

To label the axis of the graph I made separate loops for the x and y axis that draw multiple lines along them at intervals, at the end of these lines are the corresponding values. I also needed to

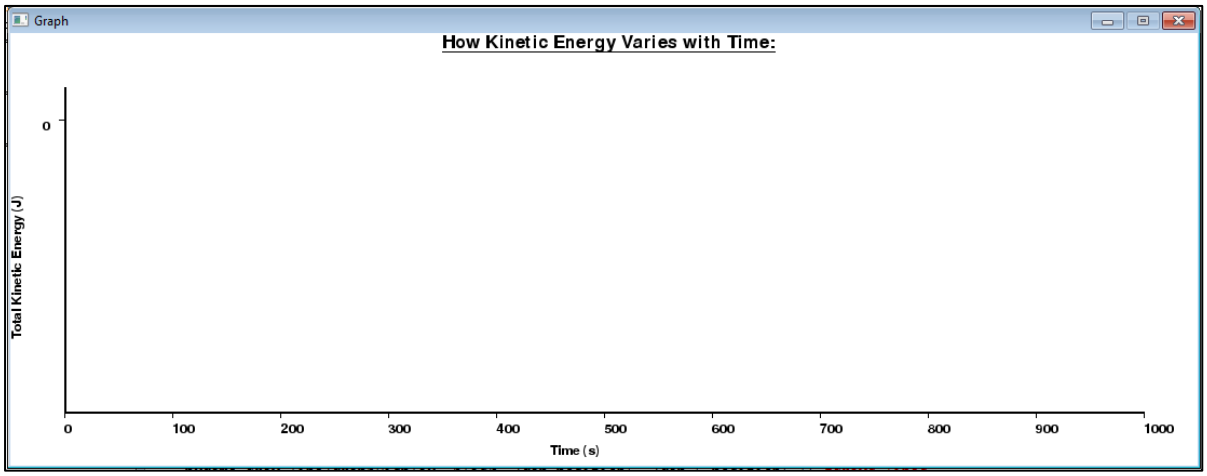
import fonts to display the text. I changed the while loop displaying the position of the mouse to a loop that allows the user to exit pygame using the close window button.

```
## GRAPH ##
import pygame
pygame.init() #initiates pygame
white = (255,255,255) #defines the colours
black = (0,0,0)
blue = (0,0,255)
title = pygame.font.Font(None, 26) #difines different types of text (Font,size)
title.set_underline(True) #underlines any text using the title font
font = pygame.font.Font(None, 18)#set to defult font

maxtime = 1000
gap = 50 #gap between the axis of the graph and the edges of the window

max_Ek = 30
y_axis = max_Ek * 10 #then *10 because scale is 1J = 10 pixels
graphDisplay = pygame.display.set_mode((maxtime+gap*2,y_axis+gap*2)) #creates window of set resolution
graphDisplay.fill(white) #makes background white
pygame.display.set_caption('Graph') #titles window
                                    #(points of ends of line,      width)
pygame.draw.line(graphDisplay, black, (gap,y_axis+50), (maxtime+gap,y_axis+gap),2)#colour and position of x axis
pygame.draw.line(graphDisplay, black, (gap,y_axis+50), (gap,gap),2)#colour and position of y axis

for i in range(0,11): #Creates points a long x axis, interval of 100 seconds
    position = i*100+gap #Converts value in seconds to position on screen
    pygame.draw.line(graphDisplay, black, (position,y_axis+gap), (position,y_axis+gap+5),1) #draws lines
    graphDisplay.blit((font.render(str(position-gap), True, (black))), (position,y_axis+gap+10)) #shows values
for i in range(0,int(max_Ek/10)+1): #Creates points a long y axis, interval of 10 Joules
    position = max_Ek-(i*100+50)+100 #Converts value in joules to position on screen
    pygame.draw.line(graphDisplay, black, (gap,position), (gap-5,position),1) #draws lines
    graphDisplay.blit((font.render(str(i*10), True, (black))), (gap-20,position)) #shows values

graphDisplay.blit((title.render("How Kinetic Energy Varies with Time:", True, (black))), (400,0)) #Displays title
graphDisplay.blit((font.render("Time (s)", True, (black))), (500,y_axis+80)) #labels x axis
text = font.render("Total Kinetic Energy (J)", True, (black)) #label for y axis
text = pygame.transform.rotate(text, 90) #rotates label by 90 degrees to line up with y axis
graphDisplay.blit(text,(0,y_axis/2))
pygame.display.update()

while True: #makes exit button work
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
```

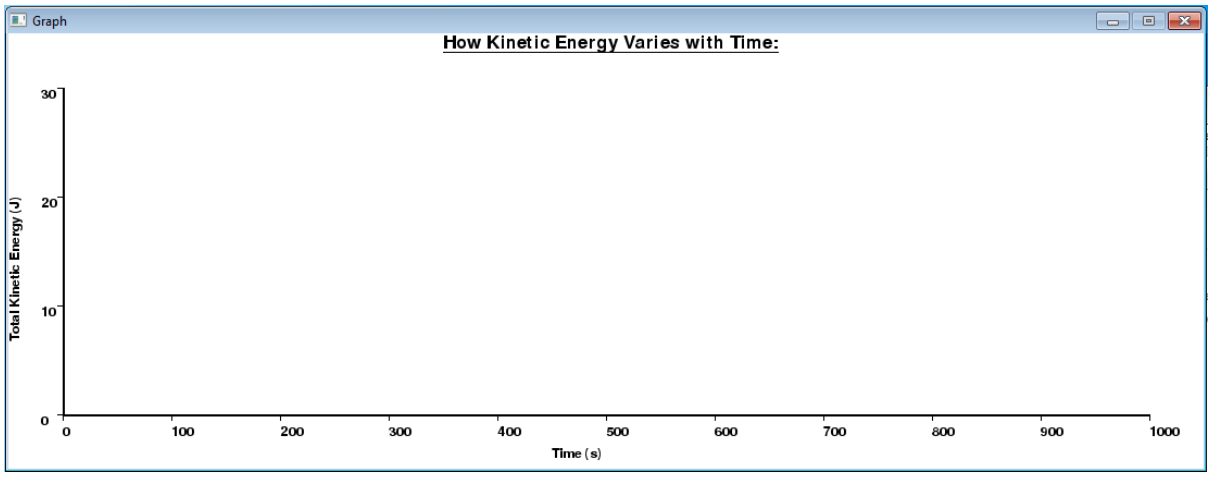
I made ‘gap’ into a variable so that the number 50 is not repeated in my code and can be changed to something else more easily.

Review:

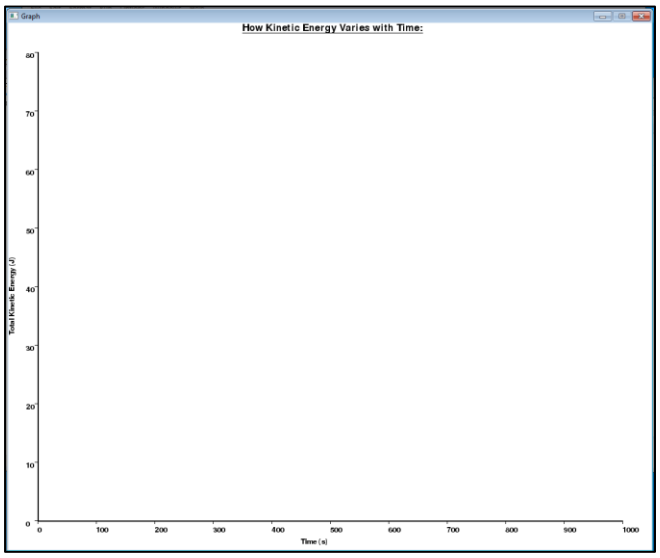
Running this program produces a graph that looks like this:



The scale on the x axis is correct but the y axis should go up to 30 in intervals of 10. To fix this I changed ‘position = max_Ek-(i*100+50)+100’ to ‘position = y_axis-(i*100+gap)+100’ because the ‘y_axis’ variable is the length of the y_axis so it takes into account the scale of the graph whereas the ‘max_Ek’ variable does not. After making these changes the program produces this graph:



This graph has the correct scale on both axis. I decided to test this section of code again before moving on by changing the value of ‘max_Ek’. I decided to make max_Ek = 80 which produced this graph:



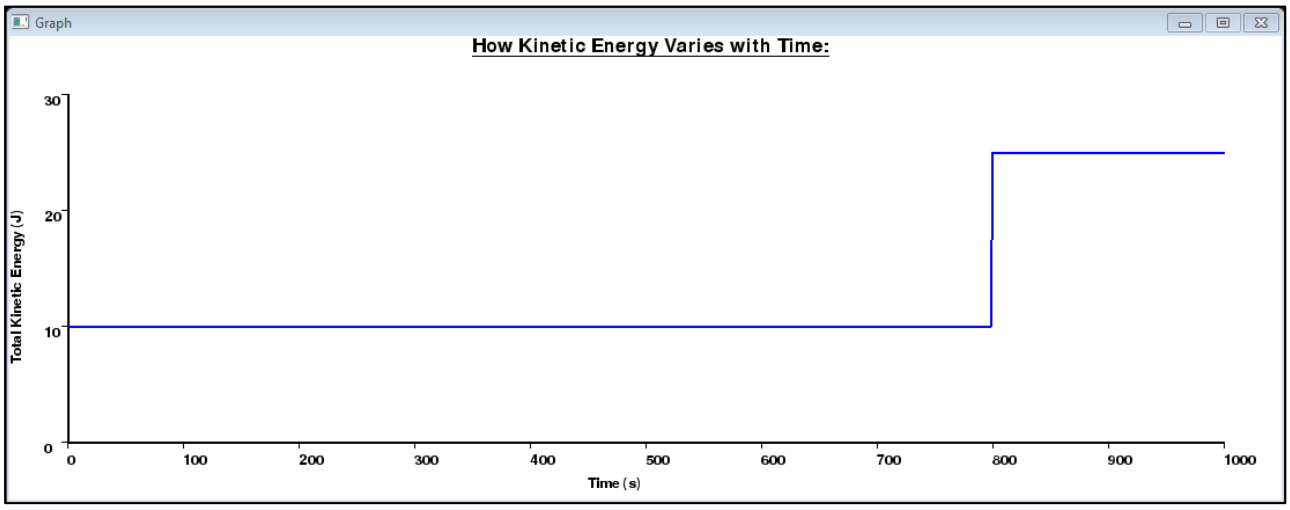
This graph’s axis is correct so I moved onto the next stage in my development.

Plotting Graph

Development:

To the plot points on the graph I needed put the graph template I created in a separate python file into the graph class in my main program. I did this by creating a new method called ‘plotGraph’ then pasted the code for the graph into it. I needed to round Ek_before up to the nearest 10 to find max_Ek which I did by importing math and using the ceil() method which returns the ceiling value of Ek_before (the smallest integer not less than Ek_before). I then created a loop that draws lines connecting all the points in the list.

```
def plotGraph(self,points):
    pygame.init() #initiates pygame
    white = (255,255,255) #defines the colours
    black = (0,0,0)
    blue = (0,0,255)
    title = pygame.font.Font(None, 26) #difines different types of text (Font,size)
    title.set_underline(True) #underlines any text using the title font
    font = pygame.font.Font(None, 18)#set to defult font
    gap = 50 #gap between the axis of the graph and the edges of the window

    max_Ek = int(math.ceil(self.Ek_before / 10.0))*10 #Rounds highest value of Ek up to the nearest 10 to determin hight of y axis
    y_axis = max_Ek * 10 #then *10 because scale is 1J = 10 pixels
    graphDisplay = pygame.display.set_mode((self.maxtime+gap*2,y_axis+gap*2)) #creates window of set resolution
    graphDisplay.fill(white) #makes background white
    pygame.display.set_caption('Graph') #titles window
                                #(points of ends of line,      width)
    pygame.draw.line(graphDisplay, black, (gap,y_axis+50), (self.maxtime+gap,y_axis+gap),2)#colour and position of x axis
    pygame.draw.line(graphDisplay, black, (gap,y_axis+50), (gap,gap),2)#colour and position of y axis

    for i in range(0,11): #Creates points a long x axis, interval of 100 seconds
        position = i*100+gap #Converts value in seconds to position on screen
        pygame.draw.line(graphDisplay, black, (position,y_axis+gap), (position,y_axis+gap+5),1) #draws lines
        graphDisplay.blit((font.render(str(position-gap), True, (black))), (position,y_axis+gap+10)) #shows values
    for i in range(0,int(max_Ek/10)+1): #Creates points a long x axis, interval of 10 Joules
        position = y_axis-(i*100+gap)+100 #Converts value in joules to position on screen
        pygame.draw.line(graphDisplay, black, (gap,position), (gap-5,position),1) #draws lines
        graphDisplay.blit((font.render(str(i*10), True, (black))), (gap-20,position)) #shows values

    graphDisplay.blit((title.render("How Kinetic Energy Varies with Time:", True, (black))), (400,0)) #Displays title
    graphDisplay.blit((font.render("Time (s)", True, (black))), (500,y_axis+80)) #labels x axis
    text = font.render("Total Kinetic Energy (J)", True, (black)) #label for y axis
    text = pygame.transform.rotate(text, 90) #rotates label by 90 degrees to line up with y axis
    graphDisplay.blit(text,(0,y_axis/2))

    for i in range(0,len(points)-1):
        posx = (self.maxtime+gap*2)-gap
        posy = (y_axis+gap*2)-gap
        pygame.draw.line(graphDisplay, blue, (posx-points[i][0],posy-points[i][1]*10), (posx-points[i+1][0],posy-points[i+1][1]*10),2)

    pygame.display.update()

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()
```




```
Total Kinetic Energy before Collision: 25.0 J

Velocity After Collision:  2.0 m/s
Kinetic Energy after Collision:  10.0 J
Change in Kinetic Energy:  15.0 J
Time:  201 s
```

This produces a graph that is flipped because the origin of the graph is different to (0,0) on the pygame window.

The graph is wrong because the time taken should be where kinetic energy changes and it should go from a high value to a lower value. This mean the graph has been plotted the wrong way round which happened because 0 for height of the screen is opposite to 0 on the graphs y axis. I fixed this by doing x_axis-(value) when plotting the points:

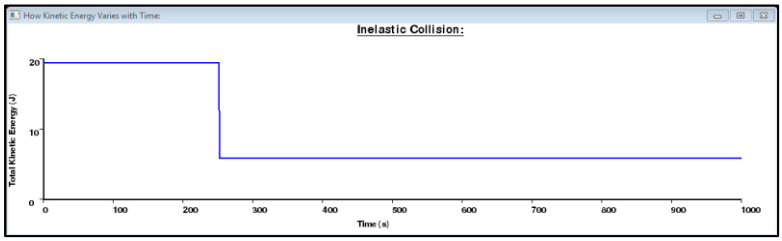
```
posx = (self.maxtime+gap*2)-gap #positioning of points on x axis depending on maximum time
posy = (y_axis+gap*2)-gap #positioning of points on y axis depending on scale
for i in range(0,len(points)-1): #loop that draws lines connecting points on graph
    pygame.draw.line(graphDisplay,blue,(x_axis-(posx-points[i][0]),posy-points[i][1]*10),(x_axis-(posx-points[i+1][0]),posy-points[i+1][1]*10),2)
```

I then made it so that the graph class uses the list of points to work out if collision is elastic or inelastic:

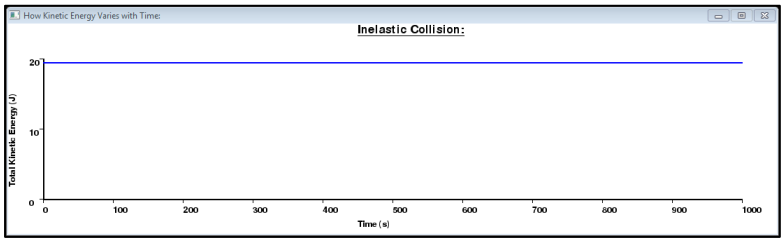
```
if points[0][1] == points[len(points)-1][1]:
    message = "Elastic Collision:"
else:
    message = "Inelastic Collision:"

graphDisplay.blit((title.render(message, True, (black))), (500,0)) #Displays title
```

Review:

To test that kinetic energy is rounded correctly and that the displayed elements of the graph are right I will use the relevant parts of my test table. To check that the graph is plotted correctly I will compare it to the list of points printed in the python shell.




```
Total Kinetic Energy before Collision: 19.5 J

Velocity After Collision:  1.29 m/s
Kinetic Energy after Collision:  5.82 J
Change in Kinetic Energy:  13.68 J
Time:  251 s
```




```
Total Kinetic Energy before Collision: 19.5 J

Velocity of Object 1 after Collision:  -0.43 m/s
Velocity of Object 2 after Collision:  3.57 m/s
Kinetic Energy after Collision:  19.49 J
Change in Kinetic Energy:  0.01 J
Time:  251 s
```

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
3	Kinetic energy rounded correctly	m1 = 4 v1 = 3 m2 = 3 v2 = -1	Normal	Ek_before = 19.5 MaxEk = 20	Ek_before = 19.5 MaxEk = 20	None
3	Graph titled correctly	Collision Type: elastic	Normal	“Elastic Collision:”	“Inelastic Collision:”	Graph is labelled incorrectly because Ek before and Ek after are rounded too early so there is a change in kinetic energy where there should not be.
3	Graph titled correctly	Collision Type: inelastic	Normal	“Inelastic Collision:”	“Inelastic Collision:”	None

Validation

Development:

The maximum kinetic energy needs to be validated because if the graph is too big it will not be visible on the users monitor. To fix this I made it so that the scale is 10:1 for any value lower than 90 and the scale is 1:1 for any value between 90 and 900. If the kinetic energy before the collision is greater than 900 a graph will not be created because a scale smaller than 1:1 would not be useful to the user because the changes in energy would not be clear. I added this code to the plotGraph method in my graph class:

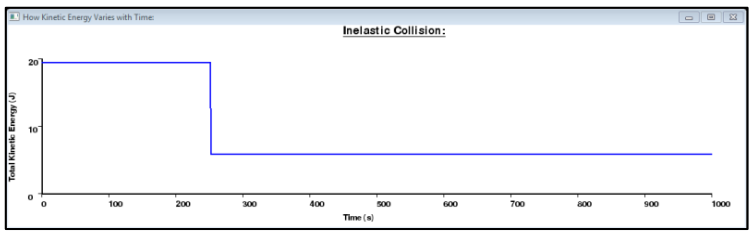
```
if self.Ek_before <= 900: #works out scale of y axis depening on maximum Ek
    scale = 100 #100 points on GUI represents 100J
    size = 1
    if self.Ek_before <= 90: #bigger scale of there is space
        scale = 10 #100 points on GUI represents 10J
        size = 10
    #<= 900 or 90 because anything greater than that is too big for screen
```

```
else: #validation - if maxEk is too big for screen graph won't be displayed
    print("Data too big to create graph")
```

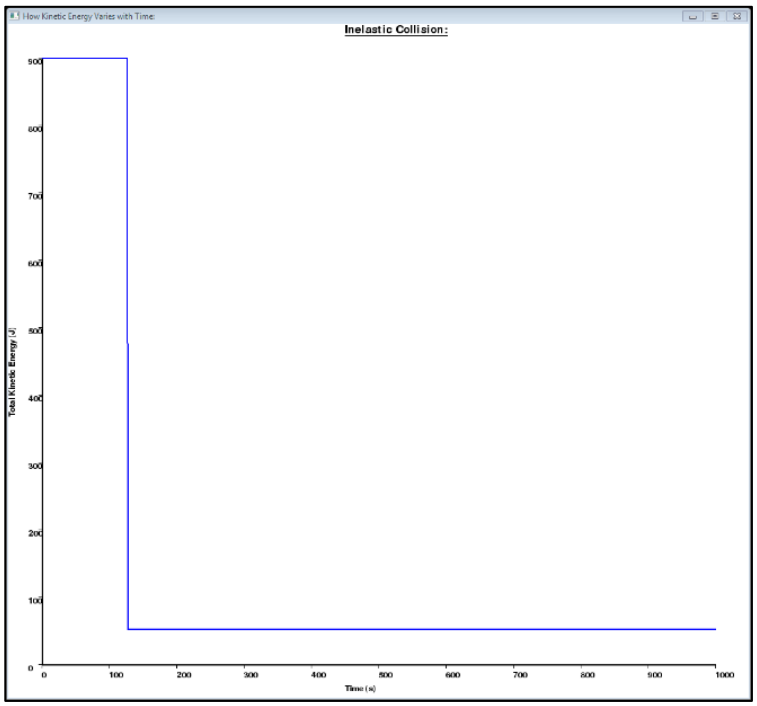
Review:

To make sure that maximum kinetic energy is validated correctly and that the right scale is used for my graph I will use the relevant sections of my test table:

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 4
Velocity of object 1: 3
Mass of object 2: 3
Velocity of object 2: -1
Collision Type(elastic/inelastic):
inelastic
--Object 1--
Mass: 4.0 kg
Initial Velocity: 3.0 m/s
Momentum:  12.0 kg m/s
Kinetic Energy:  18.0 J

--Object 2--
Mass: 3.0 kg
Initial Velocity: -1.0 m/s
Momentum:  -3.0 kg m/s
Kinetic Energy:  1.5 J

Distance between objects:  1000.0 m

Total Momentum before Collision: 9.0 kg m/s
Total Kinetic Energy before Collision: 19.5 J

Velocity After Collision:  1.29 m/s
Kinetic Energy after Collision:  5.82 J
Change in Kinetic Energy:  13.68 J
Time:  251 s
```



```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 100
Velocity of object 1: 3
Mass of object 2: 36
Velocity of object 2: -5
Collision Type(elastic/inelastic):
inelastic
--Object 1--
Mass: 100.0 kg
Initial Velocity: 3.0 m/s
Momentum:  300.0 kg m/s
Kinetic Energy:  450.0 J

--Object 2--
Mass: 36.0 kg
Initial Velocity: -5.0 m/s
Momentum:  -180.0 kg m/s
Kinetic Energy:  450.0 J

Distance between objects:  1000.0 m

Total Momentum before Collision: 120.0 kg m/s
Total Kinetic Energy before Collision: 900.0 J

Velocity After Collision:  0.88 m/s
Kinetic Energy after Collision:  52.66 J
Change in Kinetic Energy:  847.34 J
Time:  126 s
```




```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 100
Velocity of object 1: 3
Mass of object 2: 36.01
Velocity of object 2: -5
Collision Type(elastic/inelastic):
inelastic
--Object 1--
Mass: 100.0 kg
Initial Velocity: 3.0 m/s
Momentum:  300.0 kg m/s
Kinetic Energy:  450.0 J

--Object 2--
Mass: 36.01 kg
Initial Velocity: -5.0 m/s
Momentum:  -180.05 kg m/s
Kinetic Energy:  450.12 J

Distance between objects:  1000.0 m

Total Momentum before Collision: 119.95 kg m/s
Total Kinetic Energy before Collision: 900.12 J

Velocity After Collision:  0.88 m/s
Kinetic Energy after Collision:  52.66 J
Change in Kinetic Energy:  847.46 J
Time:  126 s
Data too big to create graph
```

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 7.6
Velocity of object 1: 20.14
Mass of object 2: 3.74
Velocity of object 2: -11.65
Collision Type(elastic/inelastic):
inelastic
--Object 1--
Mass: 7.6 kg
Initial Velocity: 20.14 m/s
Momentum:  153.06 kg m/s
Kinetic Energy:  1541.35 J

--Object 2--
Mass: 3.74 kg
Initial Velocity: -11.65 m/s
Momentum:  -43.57 kg m/s
Kinetic Energy:  253.8 J

Distance between objects:  1000.0 m

Total Momentum before Collision: 109.49 kg m/s
Total Kinetic Energy before Collision: 1795.15 J

Velocity After Collision:  9.66 m/s
Kinetic Energy after Collision:  529.1 J
Change in Kinetic Energy:  1266.05 J
Time:  32 s
Data too big to create graph
```

SC:	Function:	Input/Test:	Test Type:	Expected Result:	Output:	Changes Needed:
3	Maximum Ek < 900	m1 = 4 v1 = 3 m2 = 3 v2 = -1	Normal	Ek_before = 19.5 MaxEk = 20 Graph Created	Ek_before = 19.5 MaxEk = 20	None
3	Maximum Ek < 900	m1 = 7.6 v1 = 20.14 m2 = 3.74 v2 = -11.65	Error	Ek_before = 1777 Data too big to create graph	Ek_before = 1777J	None

3	Maximum Ek < 900	m1 = 100 v1 = 3 m2 = 36 v2 = -5	Boundary	Ek_before = 900 MaxEk = 900 Graph Created	Ek_before = 900J MaxEk = 900	None
3	Maximum Ek < 900	m1 = 100 v1 = 3 m2 = 36.01 v2 = -5	Boundary	Ek_before = 900.1 Data too big to create graph	Ek_before = 900.1J	None
3	Correct scale used for y axis	m1 = 100 v1 = 3 m2 = 36 v2 = -5	Normal	Ek_before = 900 Scale = 100	Ek_before = 900J Scale = 100	None
3	Correct scale used for y axis	m1 = 4 v1 = 3 m2 = 3 v2 = -1	Normal	Ek_before = 19.5 Scale = 10	Ek_before = 19.5J Scale = 10	None
3	Correct scale used for y axis	m1 = 7.6 v1 = 20.14 m2 = 3.74 v2 = -11.65	Error	Ek_before = 1777 Data too big to create graph	Ek_before = 1777J	None

Final Section 2 Version

Regression testing:

To finish development of section 2 of my program I needed to adhere to all points on my success criteria for this section and the previous section. I already tested the parts from section 2 so I do not need to test them again since I have not made more changes to them however I need to retest some parts from section 1.

While retesting the parts of my program I created in section 1 I found that some elastic collisions produced a change in kinetic energy which should be impossible because in my analysis I defined an elastic collision as “a collision where momentum and kinetic energy are conserved”.

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 4.543
Velocity of object 1: 6.4213
Mass of object 2: 3.988
Velocity of object 2: -2.976
Collision Type(elastic/inelastic):
elastic
--Object 1--
Mass: 4.543 kg
Initial Velocity: 6.4213 m/s
Momentum:  29.17 kg m/s
Kinetic Energy:  93.66 J

--Object 2--
Mass: 3.988 kg
Initial Velocity: -2.976 m/s
Momentum:  -11.87 kg m/s
Kinetic Energy:  17.66 J

Distance between objects:  1000.0 m

Total Momentum before Collision: 17.3 kg m/s
Total Kinetic Energy before Collision: 111.32 J

Velocity of Object 1 after Collision:  -2.36 m/s
Velocity of Object 2 after Collision:  7.03 m/s
Kinetic Energy after Collision:  111.2 J
Change in Kinetic Energy:  0.12 J
Time:  107 s
```

I resolved this by only rounding values calculated when they are being printed or displayed in the GUI instead of rounding them while they are still being used for further calculations. I also decided to combine the plotPoints and plotGraph methods to make the program more efficient.

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 4.543
Velocity of object 1: 6.4213
Mass of object 2: 3.988
Velocity of object 2: -2.976
Collision Type(elastic/inelastic):
elastic
--Object 1--
Mass: 4.543 kg
Initial Velocity: 6.4213 m/s
Momentum:  29.17 kg m/s
Kinetic Energy:  93.66 J

--Object 2--
Mass: 3.988 kg
Initial Velocity: -2.976 m/s
Momentum:  -11.87 kg m/s
Kinetic Energy:  17.66 J

Distance between objects:  1000.0 m

Total Momentum before Collision: 17.3 kg m/s
Total Kinetic Energy before Collision: 111.32 J

Velocity of Object 1 after Collision:  -2.36 m/s
Velocity of Object 2 after Collision:  7.03 m/s
Kinetic Energy after Collision:  111.32 J
Change in Kinetic Energy:  0.0 J
Time:  107 sseconds
```

These changes fixed the rounding error meaning that everything from section 1 and 2 now works.

Review:

Success Criteria for this section:

1. Algorithm written using object oriented programming that calculates the time taken for the collision.
2. Algorithm that works out the points to be plotted on a kinetic energy against time graph.
3. GUI that displays the graph of kinetic energy against time.
4. An obvious way to end the program.
5. The points on the success criteria of the previous section.

I labelled part of the code I added in section 2 to show where each point on the success criteria has been met:

```
class Graph(object): #Creates a graph of kinetic energy against time

    #Constructor
    def __init__(self, x1, x2, Ek_before, Ek_after, maxtime, v1, v2):

        #Attributes
        self.x1 = x1
        self.x2 = x2
        self.Ek_before = Ek_before
        self.Ek_after = Ek_after
        self.maxtime = maxtime
        self.v1 = v1
        self.v2 = v2

    #Methods
    def getTime(self): #Calculates the time taken for the objects to collide
        time_taken = 0
        while self.x2 >= self.x1 and time_taken < self.maxtime: #detects collision and limits time
            self.x1 = self.x1 + self.v1 #each sec the object moves a certain distance (it's velocity)
            self.x2 = self.x2 + self.v2
            time_taken = time_taken + 1 #increase time taken as objects move
        return time_taken

    def plotGraph(self,time): #needs time taken to know when Ek changes
        points = []
        for i in range (0,time+1):
            point = [i,self.Ek_before] #[x = time, y = Ek]
            points.append(point)
        for i in range (time+1,self.maxtime+1): #after collision
            point = [i,self.Ek_after] #x = time, y = Ek
            points.append(point)

        pygame.init() #initiates pygame
        white = (255,255,255) #defines the colours
        black = (0,0,0)
        blue = (0,0,255)
        title = pygame.font.Font(None, 26) #difines different types of text (Font,size)
        title.set_underline(True) #underlines any text using the title font
        font = pygame.font.Font(None, 18)#set to defult font
        gap = 50 #gap between the axis of the graph and the edges of the pygame window

        if self.Ek_before <= 900: #works out scale of y axis depening on maximum Ek
            scale = 100 #100 points on GUI represents 100J
            size = 1
            if self.Ek_before <= 90: #bigger scale of there is space
                scale = 10 #100 points on GUI represents 10J
                size = 10
            #<= 900 or 90 because anything greater than that is too big for screen

            max_Ek = int(math.ceil(self.Ek_before / scale))*scale #Rounds highest value of Ek up to the nearest 10/100 to determin hight of y axis
            y_axis = max_Ek * size #length of y_axis depends on scale determinded earlier
            x_axis = self.maxtime+gap*2 #total width of pygame window
            graphDisplay = pygame.display.set_mode((x_axis,y_axis+gap*2)) #creates window of set resolution
            graphDisplay.fill(white) #makes background white
            pygame.display.set_caption('How Kinetic Energy Varies with Time:') #titles window
                                       #(points of ends of line (x,y),           width)
            pygame.draw.line(graphDisplay, black, (gap,y_axis+50), (self.maxtime+gap,y_axis+gap),2)#colour and position of x axis
            pygame.draw.line(graphDisplay, black, (gap,y_axis+50), (gap,gap),2)#colour and position of y axis


            for i in range(0,11): #Creates points a long x axis, interval of 100 seconds
                position = i*100+gap #Converts value in seconds to position on screen
                pygame.draw.line(graphDisplay, black, (position,y_axis+gap), (position,y_axis+gap+5),1) #draws lines
                graphDisplay.blit((font.render(str(position-gap), True, (black))), (position,y_axis+gap+10)) #shows values
            for i in range(0,int(max_Ek/scale)+1): #Creates points a long x axis, interval of 10 Joules
                position = y_axis-(i*100+gap)+100 #Converts value in joules to position on screen
                pygame.draw.line(graphDisplay, black, (gap,position), (gap-5,position),1) #draws lines
                graphDisplay.blit((font.render(str(i*scale), True, (black))), (gap-20,position)) #shows values

            if points[0][1] == points[len(points)-1][1]: #works out collision type to title graph
                message = "Elastic Collision:"
            else:
                message = "Inelastic Collision:"

            graphDisplay.blit((title.render(message, True, (black))), (500,0)) #Displays title
            graphDisplay.blit((font.render("Time (s)", True, (black))), (500,y_axis+80)) #labels x axis
            text = font.render("Total Kinetic Energy (J)", True, (black)) #label for y axis
            text = pygame.transform.rotate(text, 90) #rotates label by 90 degrees to line up with y axis
            graphDisplay.blit(text,(0,y_axis/2))

            posx = (self.maxtime+gap*2)-gap #positions points plotted to line up with axis
            posy = (y_axis+gap*2)-gap
            for i in range(0,len(points)-1): #plots points by drawing multiple tiny lines
                pygame.draw.line(graphDisplay,blue,(x_axis-(posx-points[i][0]),posy-points[i][1]*size),(x_axis-(posx-points[i+1][0]),posy-points[i+1][1]*size),2)

            pygame.display.update()

            while True: #allows user to exit program
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        pygame.quit()
                        quit()
        else: #validation - if maxEk is too big for screen graph won't be displayed
            print("Data too big to create graph")
```

Review Summary:

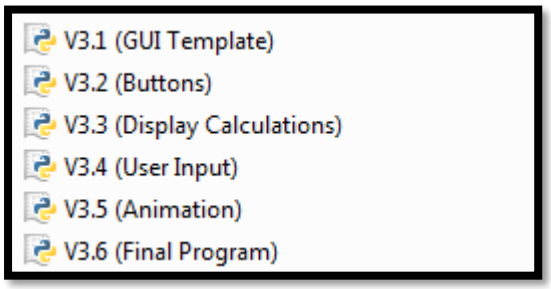
For the most part my development for this section closely followed my design with very few changes. For the main algorithms in my program the process is the same expect for the addition of some new variables and minor changes. I also combined some of the methods I planned to create in the design section into one method. I changed some variable names to better suit their purpose and added some different test data to my test table when I discovered other aspects of my program that needed testing. After testing my final version for section 2 against the success criteria to ensure that everything works and making sure it includes the usability features I mentioned in my design, I am now ready to move onto developing section 3 of my project.

Section 3 Development:

GUI Template

Development:

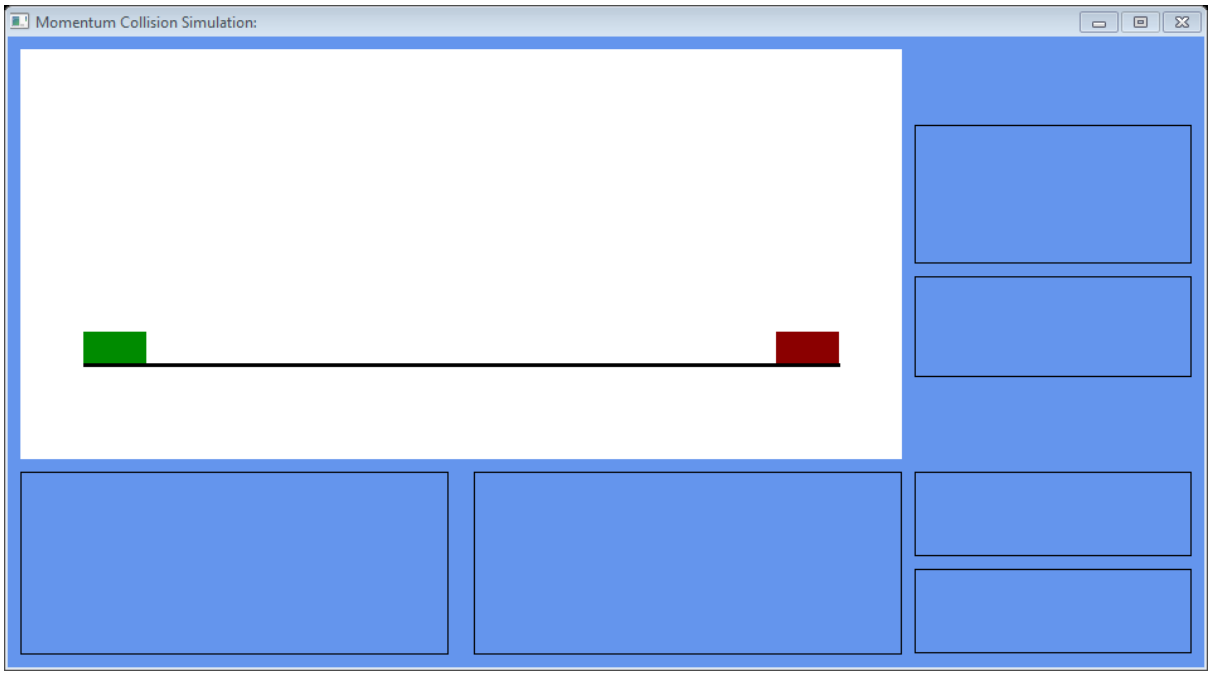
In my design for this section I planned the different points on my success criteria that I will cover and broke them down into 6 different parts. To begin my development I created a folder for section 3 and within this a python file for each of these sub versions so that I know the order tasks need to be completed and to get an idea of how far along I am in the development process to ensure I don't run out of time:



To begin my development for this section I copied my final code from section 2 into V3.1 then looked at my design section to see how I planned to layout the GUI. I decided to create a template of my GUI first because this will make the rest of my development easier as I will already know the position objects will need to be in. I put the main code for the GUI in the main program instead of in a class because only one window is needed for my project.

In my program, I defined the fonts and colours that I will use which I did some further research to find. I created a loop to show the position of my mouse on the window, which I used to find the coordinates, needed for the objects. Defining the colours as variables instead of using the colour code each time means that I can easily change the colours of certain elements in my development after user feedback.

```
pygame.init() #initiates pygame
white = (255,255,255) #defines the colours using RGB colours codes
black = (0,0,0)
light_blue = (100,149,237)
dark_blue = (61,89,171)
green = (0,139,0)
red = (139,0,0)

big_font = pygame.font.Font(None, 18)#difines different types of text (Font,size)
big_font2 = pygame.font.Font(None, 26)
big_font2.set_underline(True)
small_font = pygame.font.Font(None, 18)
small_font2 = pygame.font.Font(None, 26)
small_font2.set_underline(True)

guiDisplay = pygame.display.set_mode((950,500)) #creates window of set resolution
guiDisplay.fill(light_blue) #makes background blue
pygame.draw.rect(guiDisplay, white, [10, 10, 700, 325])#draws box that will contain animation
pygame.draw.line(guiDisplay, black, (60,260), (660,260),3) #draws platform for objects
pygame.draw.rect(guiDisplay, green, [60, 234, 50, 25]) #draws object 1 on platform
pygame.draw.rect(guiDisplay, red, [610, 234, 50, 25]) #draws object 2 on platform

pygame.draw.rect(guiDisplay, black, [10, 345, 340, 145], 1) #draws boxes to seperate areas of GUI
pygame.draw.rect(guiDisplay, black, [370, 345, 340, 145], 1)
pygame.draw.rect(guiDisplay, black, [720, 345, 220, 67.5], 1)
pygame.draw.rect(guiDisplay, black, [720, 422.5, 220, 67.5], 1)
pygame.draw.rect(guiDisplay, black, [720, 70, 220, 110], 1)
pygame.draw.rect(guiDisplay, black, [720, 190, 220, 80], 1)
pygame.display.set_caption('Momentum Collision Simulation:') #titles window
while True: #shows position of mouse
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        print(event)
    pygame.display.update()
```



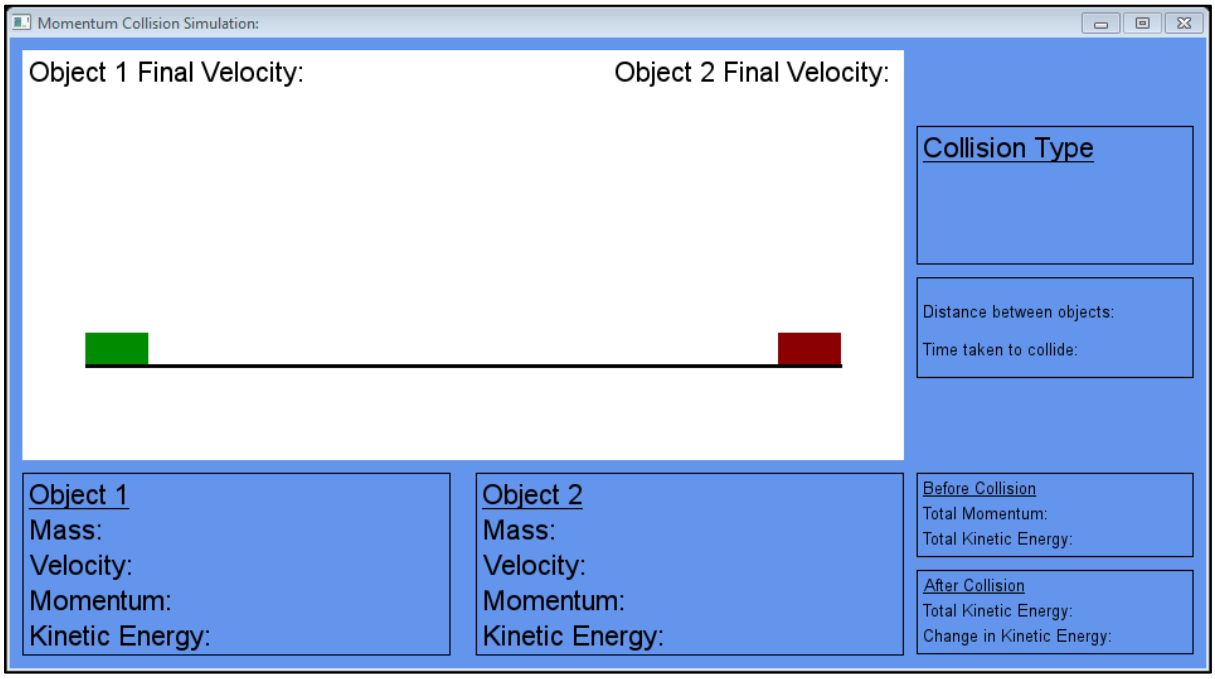
I then displayed the text on the GUI and used a loop for text that is repeated in multiple boxes. I found that the default font for pygame made certain letters look too close together which was difficult to read so I changed the font to Arial which is much clearer and changed the text sizes to fit the boxes.

Changed Code:

```
big_font = pygame.font.Font("C:\Windows\Fonts\Arial.ttf", 22)#difines different types of text (Font,size)
big_font2 = pygame.font.Font("C:\Windows\Fonts\Arial.ttf", 22)
big_font2.set_underline(True) #underlines font
small_font = pygame.font.Font("C:\Windows\Fonts\Arial.ttf", 13)
small_font2 = pygame.font.Font("C:\Windows\Fonts\Arial.ttf", 13)
small_font2.set_underline(True) #underlines font
```

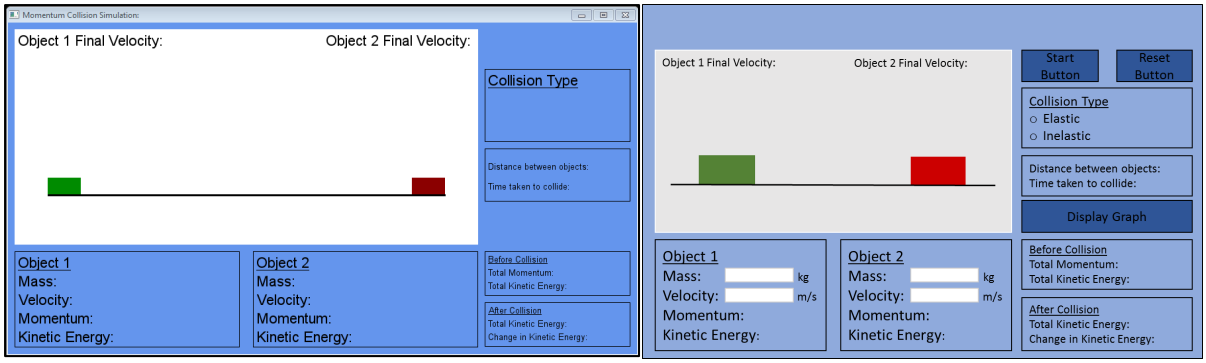
Added Code:

```
guiDisplay.blit((big_font2.render("Collision Type", True, (black))), (725,75))#labels display boxes
guiDisplay.blit((small_font.render("Distance between objects:", True, (black))), (725,210))
guiDisplay.blit((small_font.render("Time taken to collide:", True, (black))), (725,240))
guiDisplay.blit((small_font2.render("Before Collision", True, (black))), (725,350))
guiDisplay.blit((small_font.render("Total Momentum:", True, (black))), (725,370))
guiDisplay.blit((small_font.render("Total Kinetic Energy:", True, (black))), (725,390))
guiDisplay.blit((small_font2.render("After Collision", True, (black))), (725,427.5))
guiDisplay.blit((small_font.render("Total Kinetic Energy:", True, (black))), (725,447.5))
guiDisplay.blit((small_font.render("Change in Kinetic Energy:", True, (black))), (725,467.5))
guiDisplay.blit((big_font.render("Object 1 Final Velocity:", True, (black))), (15,15))
guiDisplay.blit((big_font.render("Object 2 Final Velocity:", True, (black))), (480,15))
guiDisplay.blit((big_font2.render("Object 1", True, (black))), (15,350))
guiDisplay.blit((big_font2.render("Object 2", True, (black))), (375,350))

messages = ["Mass:", "Velocity:","Momentum:","Kinetic Energy:"]
x = 0
for i in range (0,len(messages)): #used loop for labels that are repeated in multiple boxes
    x = x + 28 #28 is gap between lines of text
    guiDisplay.blit((big_font.render(messages[i], True, (black))), (15,350+x))
    guiDisplay.blit((big_font.render(messages[i], True, (black))), (375,350+x))

pygame.display.set_caption('Momentum Collision Simulation:') #titles window
```



Review:

I could have used functions or classes for text or creating rectangles, which would avoid some repetition however; this would take more lines of code so I decided it was not worth it. There are no values to test in this section so the test table is irrelevant but I will compare the GUI I created so far to the template I created in my design section:



Everything that needs to be included at this stage in my development is included. I made the distance between the objects and their size different to my design because they are meant to be 1000m apart so to make the scale simple (1:2) I made their separation 500. The objects therefore needed to be smaller to fit in the window. The colours are also slightly different but this does not need to be changed. I will need to move the 'Object 2 Final Velocity:' text over to the left to make space for the calculated value to be displayed.

Buttons

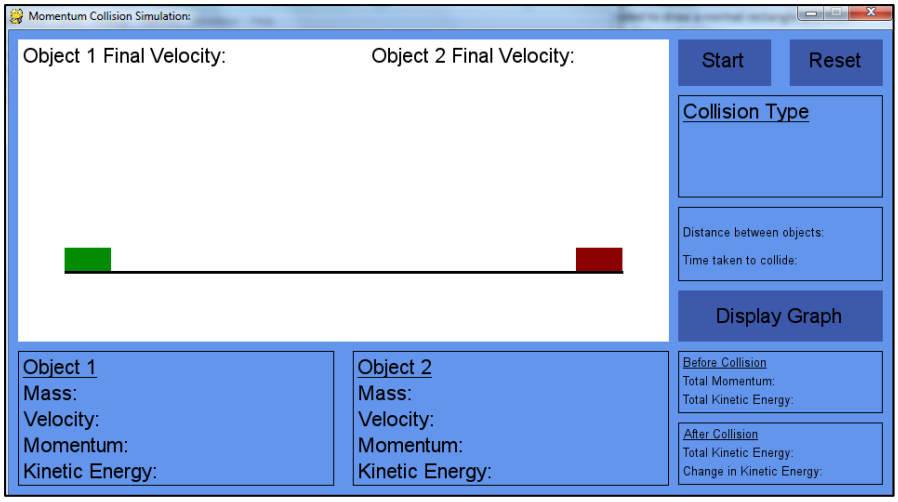
Development:

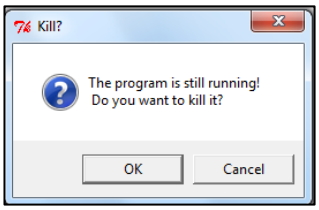
In my design section, I planned to include three buttons: start, reset and display graph. Since my GUI will have multiple buttons I decided to create a button function to avoid repetition in my code. In pygame there are no built in functions for buttons so I needed to draw a normal rectangle and then detect if the mouse is within the box and detect if the user clicks in that area.

```
def button(x,y,w,h,c,action=None): #(x,y,width,height,colour,action)
    mouse = pygame.mouse.get_pos() #gives position of mouse
    click = pygame.mouse.get_pressed() #when mouse clicked
    if x+w > mouse[0] > x and y+h > mouse[1] > y: #if users mouse within area of button
        pygame.draw.rect(guiDisplay, c,(x,y,w,h)) #draws button

        if click[0] == 1 and action != None: #if user click button
            action() #runs function of button when clicked
    else:
        pygame.draw.rect(guiDisplay, c,(x,y,w,h)) #draws button
```

```
button(720,280,220,55,dark_blue,quit) #creates buttons using function
button(720,10,100,50,dark_blue,quit)
button(840,10,100,50,dark_blue,quit)
guiDisplay.blit((big_font.render("Display Graph", True, (black))), (760,295)) #labels buttons
guiDisplay.blit((big_font.render("Start", True, (black))), (745,20))
guiDisplay.blit((big_font.render("Reset", True, (black))), (860,20))
```

Since I have not yet written the functions that will be run when the buttons are pressed, I set the action to 'quit' for now so that if the button is pressed that window will close. This allows me to test that the buttons detect clicks even though I have not written their functions yet.





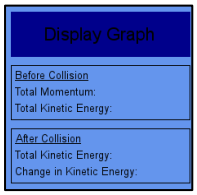
I made the buttons a different colour to the background so that they stand out as buttons. To make it even clearer that these are buttons I changed the function so that if the user's mouse is on the box it will become a darker blue. To do this I defined a new colour and added a new parameter to the function for the second colour. In addition, since I already created the display graph function in section 2 I added this action to the button so the graph is displayed when the button is clicked.

```
button(720,280,220,55,blue,dark_blue,graph.plotGraph(time_taken))
```

When running this I found that the GUI for the graph is displayed straight away and never the main GUI because the plotGraph function ran when the button was created. To fix this I made a separate parameter in the button function to take the parameter that the action function will need when the button is clicked.

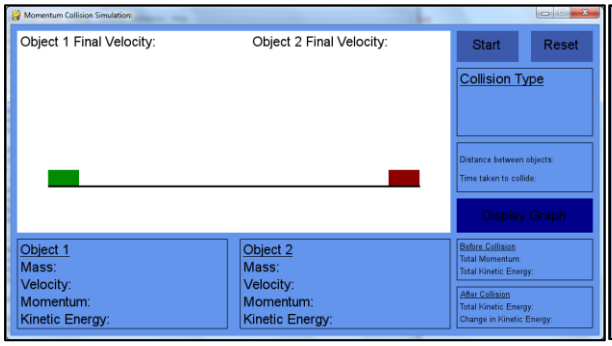
```
button(720,280,220,55,blue,dark_blue,graph.plotGraph,time_taken)
```

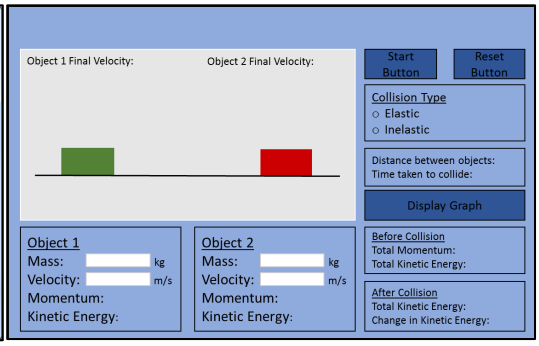
```
def button(x,y,w,h,c,c2,action,parameter=None): #(x,y,width,height,colour,second colour,action,parameter for action)
    mouse = pygame.mouse.get_pos() #gives position of mouse
    click = pygame.mouse.get_pressed() #when mouse clicked
    if x+w > mouse[0] > x and y+h > mouse[1] > y: #if users mouse within area of button
        pygame.draw.rect(guiDisplay, c2,(x,y,w,h))#makes button second colour

        if click[0] == 1: #if user click button
            action(parameter) #runs function of button when clicked
    else:
        pygame.draw.rect(guiDisplay, c,(x,y,w,h)) #draws button
```



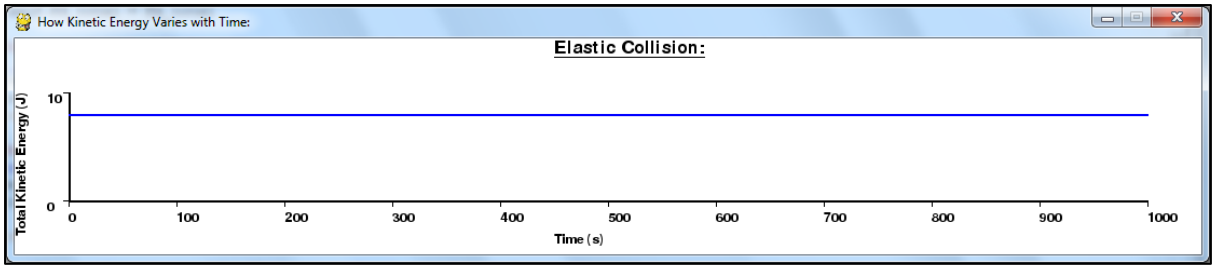
Review:

There are no values to test in this section so the test table is irrelevant but I will compare the GUI I created so far to the template I created in my design section:





Everything that needs to be included at this stage in my development is included. The colours are slightly different but this does not need to be changed. The 'display graph' button appears darker because my mouse was on it when I took the screenshot. When this button is clicked the window changes to this:



This means that the buttons function is working correctly and the function of the other buttons I will add when I have created them. I noticed that sometimes the buttons take a few clicks before they work which I will try to fix later in my development.

Displaying Calculations

Development:

Currently the results of my algorithms calculations are displayed in the python shell but they should be displayed in the GUI. To do this I must first get the start button to work so that when pressed the user input (currently still in python shell) will be used to make calculations which will then be displayed in the GUI. To do this I created a function called start then put all of the main programs from section two inside. I changed the action of the start button to start so that the start function is run when the button is clicked.

```
def start():
    x1 = 0.0 #initial positon of object 1
    x2 = 1000.0 #initial position of object 2
    maxtime = 1000 #maximum time of collision to prevent program running forever
    blue = (61,89,171)
    dark_blue = (0,0,139)

    Object1 = Momentum_Ek(m1,v1) #Created first object using Momentum_Ek class which calculates momentum and initial kinetic energy
    p1 = Object1.getMomentum() #p used to represent momentum because it resebles the greek symbol 'rho' which is used to represent it in physics
    Ek1 = Object1.getKinetic_Energy() #returns kinetic energy of first object before collision

    Object2 = Momentum_Ek(m2,v2)#Created second object using Momentum_Ek class which calculates momentum and initial kinetic energy
    p2 = Object2.getMomentum() #p used to represent momentum because it resebles the greek symbol 'rho' which is used to represent it in physics
    Ek2 = Object2.getKinetic_Energy()#returns kinetic energy of second object before collision

    print("--Object 1--")
```

```
button(720,10,100,50,blue,dark_blue,start)
```

When I ran this code an error occurred because one of my variables had the same name as one of the classes which wasn't an issue before but doesn't work when the code is inside a function.

```
    Collision = Collision(m1, v1, m2, v2, p_total, Ek_total) #Creates object usi
ng Collision class that calculates the final velocities + Change_Ek
UnboundLocalError: local variable 'Collision' referenced before assignment
>>>
```

```
collision = Collision(m1, v1, m2, v2, p_total, Ek_total)
```

I fixed this by making the variable start with a lower case letter. The user input is now only processed when the start button is clicked but the output is still displayed in the python shell. In order to display the results in the GUI I defined a new colour 'grey' to match with the rest of the text but to show that these are calculations then put this code inside the start function:

```
#displays calculations on GUI
guiDisplay.blit((small_font.render(str(x2)+"m", True, (grey))), (880,210))
guiDisplay.blit((small_font.render(str(time_taken)+"s", True, (grey))), (850,240))
guiDisplay.blit((small_font.render(str(final_Ek)+" J", True, (grey))), (850,447.5))
guiDisplay.blit((small_font.render(str(change_Ek)+" J", True, (grey))), (880,467.5))
guiDisplay.blit((small_font.render(str(round(p_total,2))+" kgm/s", True, (grey))), (830,370)) #some values still need to be rounded
guiDisplay.blit((small_font.render(str(round(Ek_total,2))+" J", True, (grey))), (850,390))
guiDisplay.blit((big_font.render(str(round(final_v1,2))+" m/s", True, (grey))), (240,15))
guiDisplay.blit((big_font.render(str(round(final_v2,2))+" m/s", True, (grey))), (615,15))
```

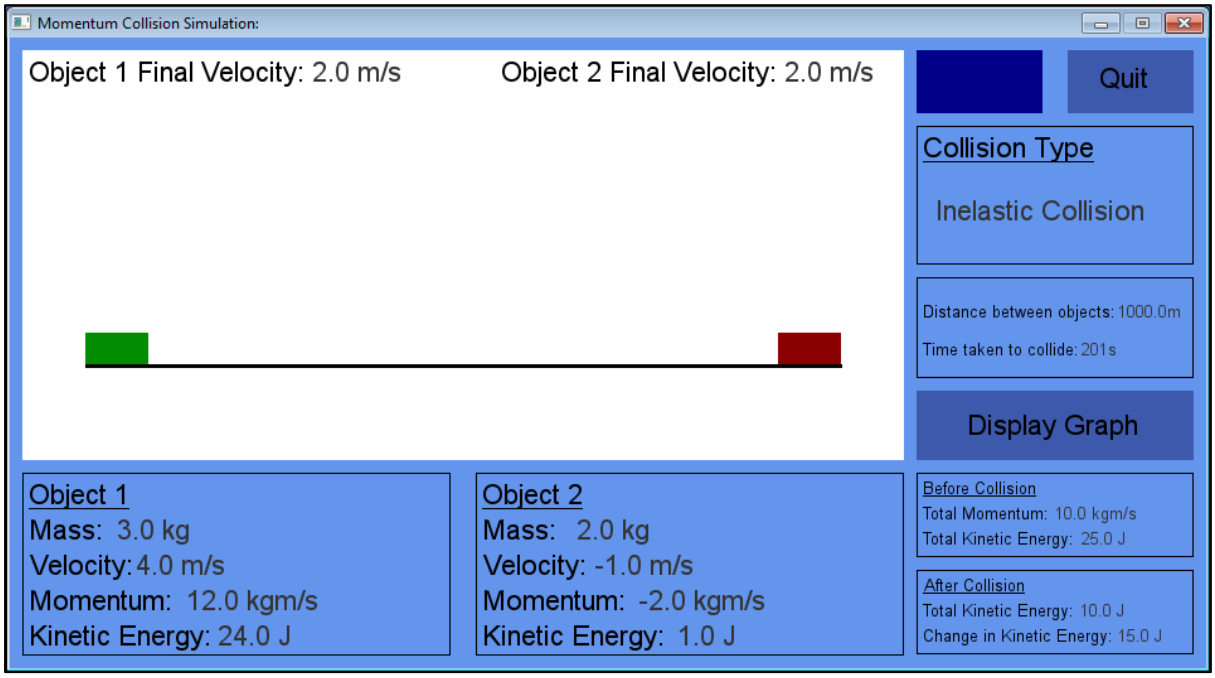
When I ran this code the results would display temporarily then disappear. This was because the code creating the template of the GUI was inside the main loop meaning the text appeared when the start button was clicked and then the background of the GUI filled in again covering the results. I fixed this by only keeping the buttons in the main loop so that they always work and moving the rest of the code out of it.

```
## MAIN LOOP ##
while True: #finds position of mouse
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    button(720,10,100,50,blue,dark_blue,start)#creates buttons using function
    button(840,10,100,50,blue,dark_blue,quit)
    guiDisplay.blit((big_font.render("Start", True, (black))), (745,20)) #labels buttons
    guiDisplay.blit((big_font.render("Reset", True, (black))), (860,20))

    pygame.display.update()
```

Review:

I found that only having the buttons in the main loop and moving the rest of the code meant that the buttons worked faster and no longer required multiple clicks. To test that the correct calculations are displayed in the right places I compared the output in the python shell for section two to the GUI displayed in section three when the same numbers are inputted.

```
----- MOMENTUM COLLISION SIMULATION -----
Mass of object 1: 3
Velocity of object 1: 4
Mass of object 2: 2
Velocity of object 2: -1
Collision Type(elastic/inelastic):
inelastic
--Object 1--
Mass: 3
Initial Velocity: 4
Momentum:  12
Kinetic Energy:  24.0

--Object 2--
Mass: 2
Initial Velocity: -1
Momentum:  -2
Kinetic Energy:  1.0

Total Momentum before Collision: 10
Total Kinetic Energy before Collision: 25.0

Velocity After Collision:  2.0
Kinetic Energy after Collision:  10.0
Change in Kinetic Energy:  15.0


Press the enter key to exit.
```



I found that the correct outputs are displayed in the GUI so I will now move onto making the object move in order to simulate the collision-taking place.

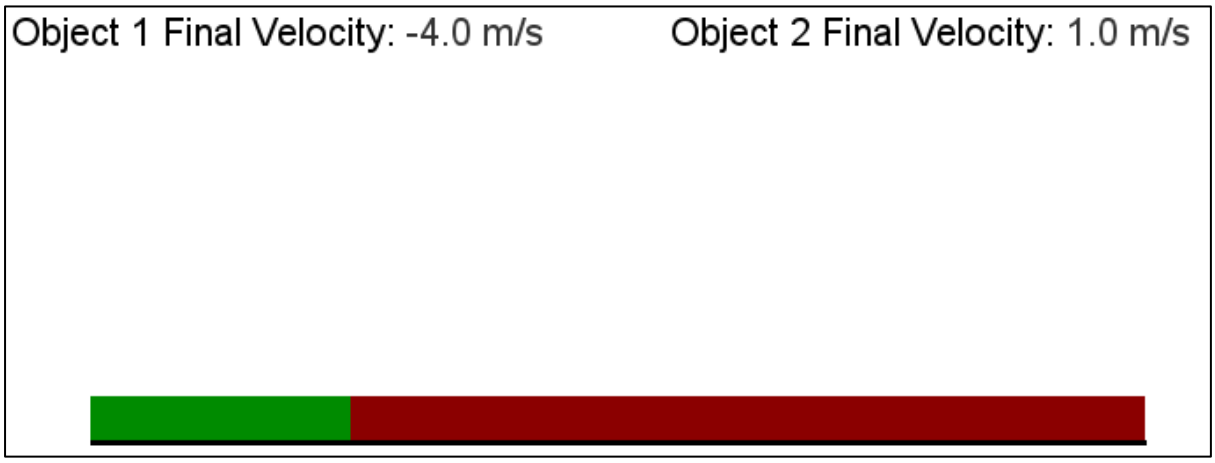
Animation

Development:

To make the objects on the GUI move I created a new animation class like I planned in my design which creates the objects then when the start button is pressed makes them collide with each other. To ensure the animation is slow enough for users to view I imported time and used the sleep method.

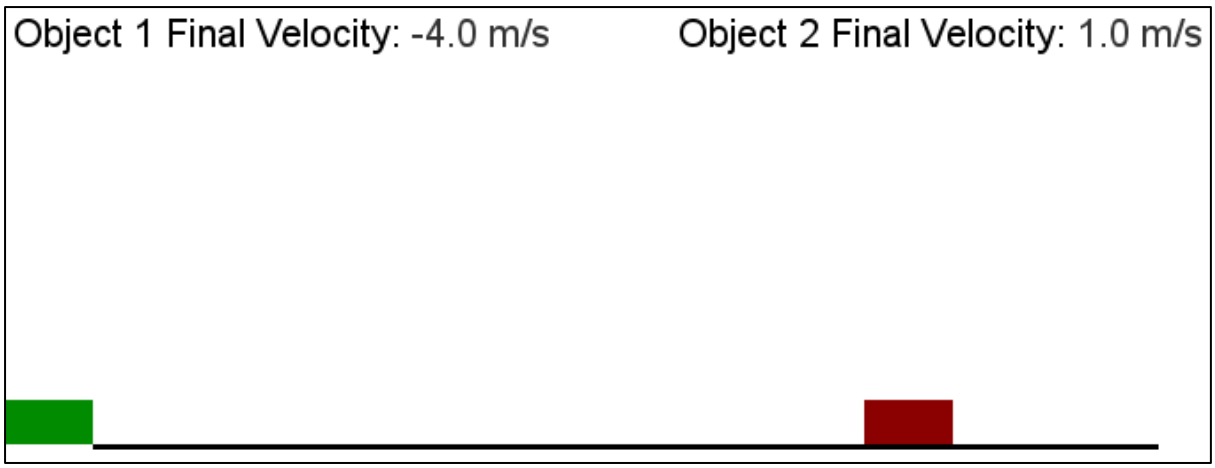
```
## Animation Class ##
class Animation(object): #Creates the objects and makes them collide

    #Constructor
    def __init__(self, v1, v2, x2):

        #Attributes
        self.v1 = v1
        self.v2 = v2
        self.x2 = x2

    #Methods
    def createObjects(self):
        pygame.draw.rect(guiDisplay, green, [60, 234, 50, 25]) #draws object 1 on platform
        pygame.draw.rect(guiDisplay, red, [610, 234, 50, 25]) #draws object 2 on platform

    def moveObjects(self,time_taken):
        distance1 = v1/2 #distance objects move per second = velocity divided by two because scale on GUI is 1:2
        distance2 = v2/2
        for i in range (0,time_taken): #moves objects at the speed they are before they collide
            time.sleep(0.01) #makes animation slow enough for users to view
            pygame.draw.rect(guiDisplay, green, [60+distance1, 234, 50, 25])#draws object 1 at its new position
            distance1 = distance1 + (v1/2)
            pygame.draw.rect(guiDisplay, red, [610+distance2, 234, 50, 25])#draws object 2 at its new position
            distance2 = distance2 - (v2/2)
            pygame.display.update()
```

When running this code I found that object 2 moved backwards so I changed “distance2 = distance2 - (v2/2)” to “distance2 = distance2 + (v2/2)” because v2 is already always negative or 0. I also found that the objects left a trail when they moved because the loop keeps drawing objects in different positions.



I fixed this issue by drawing a white rectangle behind each object after each movement. I then added another loop to the ‘moveObjects’ method so that the objects keep moving after the collision at their new speed.

```
def moveObjects(self,time_taken,final_v1,final_v2):
    distance1 = v1/2 #distance objects move per second = velocity divided by two because scale on GUI is 1:2
    distance2 = v2/2
    for i in range (0,time_taken): #moves objects at the speed they are before they collide
        time.sleep(0.01) #makes animation slow enough for users to view
        pygame.draw.rect(guiDisplay, white, [60, 234, 600, 25]) #makes backround white after each movement so no trail
        pygame.draw.rect(guiDisplay, green, [60+distance1, 234, 50, 25]) #draws object 1 at its new position
        distance1 = distance1 + (v1/2)
        pygame.draw.rect(guiDisplay, red, [610+distance2, 234, 50, 25]) #draws object 2 at its new position
        distance2 = distance2 + (v2/2)
        pygame.display.update()
    for i in range (time_taken,maxtime): #moves objects at calculated speed after collision
        time.sleep(0.01) #makes animation slow enough for users to view
        pygame.draw.rect(guiDisplay, white, [60, 234, 600, 25]) #makes backround white after each movement so no trail
        pygame.draw.rect(guiDisplay, green, [60+distance1, 234, 50, 25]) #draws object 1 at its new position
        distance1 = distance1 + (final_v1/2)
        pygame.draw.rect(guiDisplay, red, [610+distance2, 234, 50, 25]) #draws object 2 at its new position
        distance2 = distance2 + (final_v2/2)
        pygame.display.update()
```

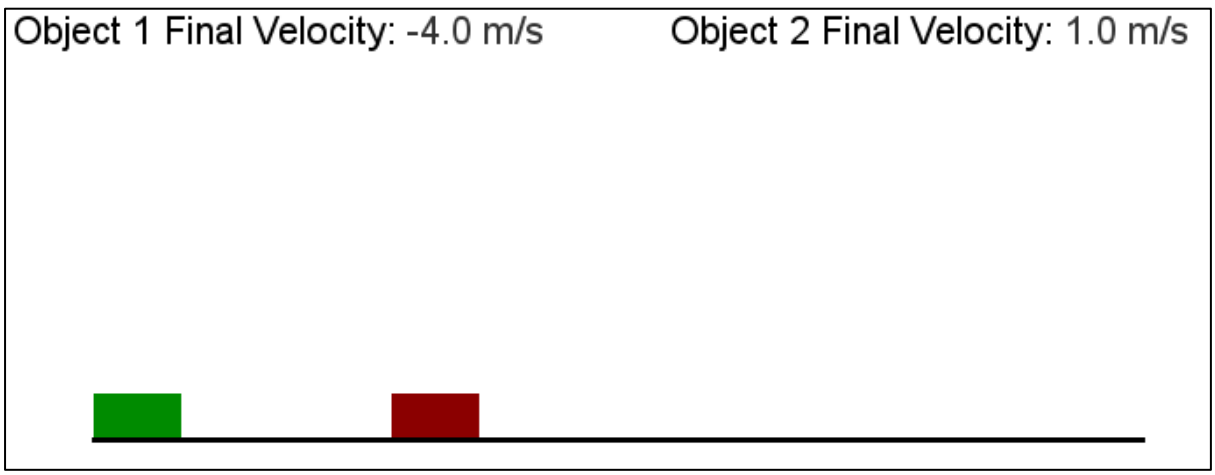


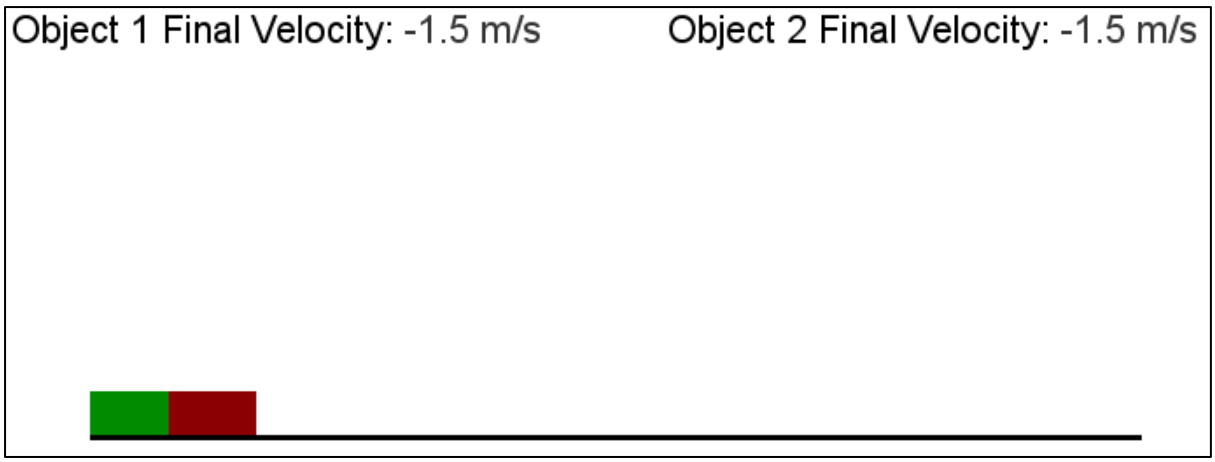
When testing this code I found that the objects do change speed after colliding however, the objects move off screen when they reach the edge and the animation continues on a loop. To fix this I stopped using the 'moveObjects' method in the loop in the start function and changed the second loop in the method to a while loop that only continues when the objects are within certain boundaries.

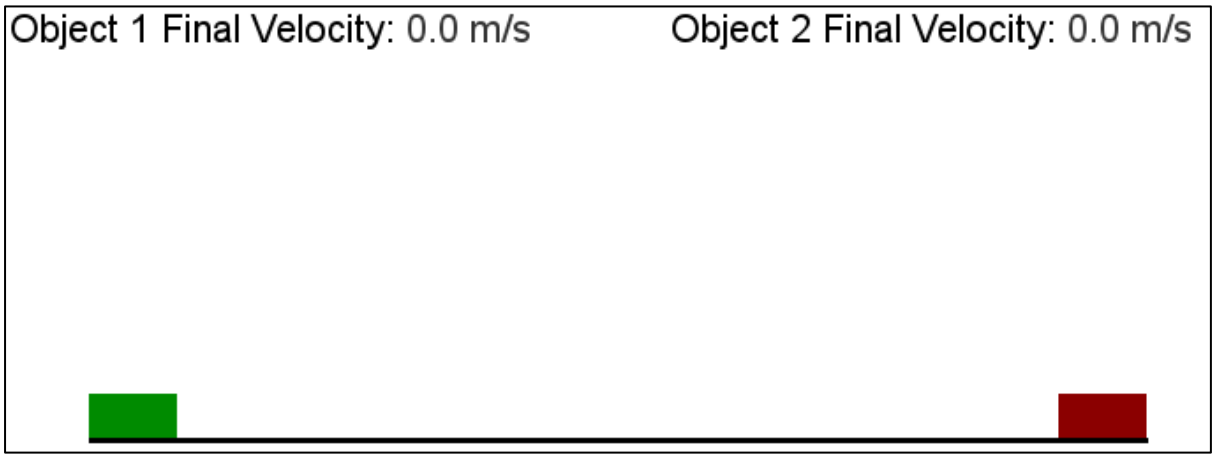
```
def moveObjects(self,time_taken,final_v1,final_v2):
    distance1 = v1/2 #distance objects move per second = velocity divided by two because scale on GUI is 1:2
    distance2 = v2/2
    for i in range (0,time_taken): #moves objects at the speed they are before they collide
        time.sleep(0.01) #makes animation slow enough for users to view
        pygame.draw.rect(guiDisplay, white, [60, 234, 600, 25]) #makes backround white after each movement so no trail
        pygame.draw.rect(guiDisplay, green, [60+distance1, 234, 50, 25]) #draws object 1 at its new position
        distance1 = distance1 + (v1/2)
        pygame.draw.rect(guiDisplay, red, [610+distance2, 234, 50, 25]) #draws object 2 at its new position
        distance2 = distance2 + (v2/2)
        pygame.display.update()
    while (60+distance1)>60 and (610+distance2)<610: #moves objects at calculated speed after collision until they reach boundary
        time.sleep(0.01) #makes animation slow enough for users to view
        pygame.draw.rect(guiDisplay, white, [60, 234, 600, 25]) #makes backround white after each movement so no trail
        pygame.draw.rect(guiDisplay, green, [60+distance1, 234, 50, 25]) #draws object 1 at its new position
        distance1 = distance1 + (final_v1/2)
        pygame.draw.rect(guiDisplay, red, [610+distance2, 234, 50, 25]) #draws object 2 at its new position
        distance2 = distance2 + (final_v2/2)
        pygame.display.update()
```

Review:

The objects begin moving when the start button is pressed then stop when one of the objects reaches the edge. In elastic collisions they move apart after colliding whereas in inelastic collision they move together at the same speed.







If the velocity of both objects are zero they will remain stationary.

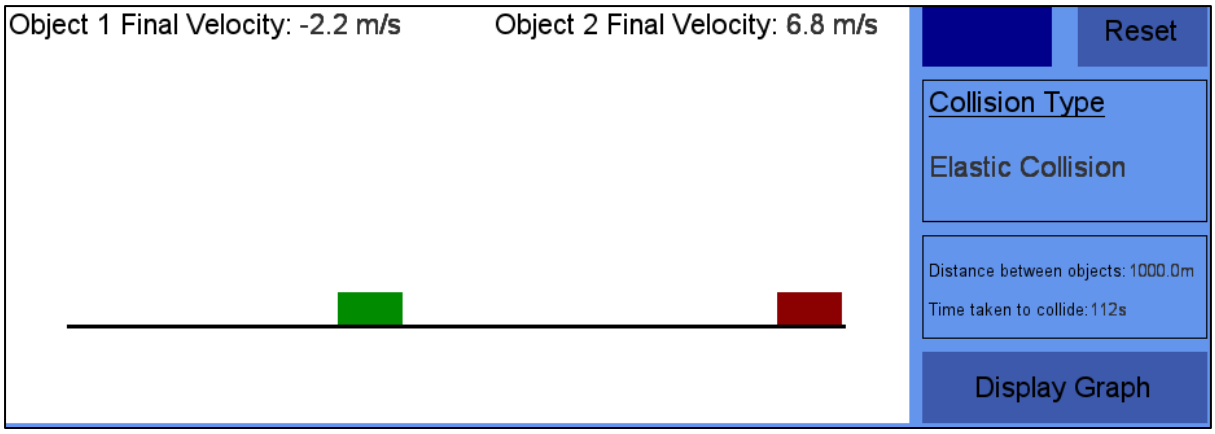
Reset

Development:

The user may want to view the animation multiple times so I made the action of the reset button 'start' so that the simulation repeats each time it is pressed.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
    button(720,280,220,55,blue,dark_blue,graph.plotGraph,time_taken) #once calculations have been made and displayed graph button starts working
    guiDisplay.blit((big_font.render("Display Graph", True, (black))), (760,295))
    button(840,10,100,50,blue,dark_blue,start) #reset button that repeats animation of objects
    guiDisplay.blit((big_font.render("Reset", True, (black))), (860,20))
    pygame.display.update()
```

Review:



When the reset button is pressed the objects move back to their starting position then collide again. The reset button can be pressed multiple times it will only work at the end of each animation when the objects have stopped moving so the animation cannot be reset midway through.

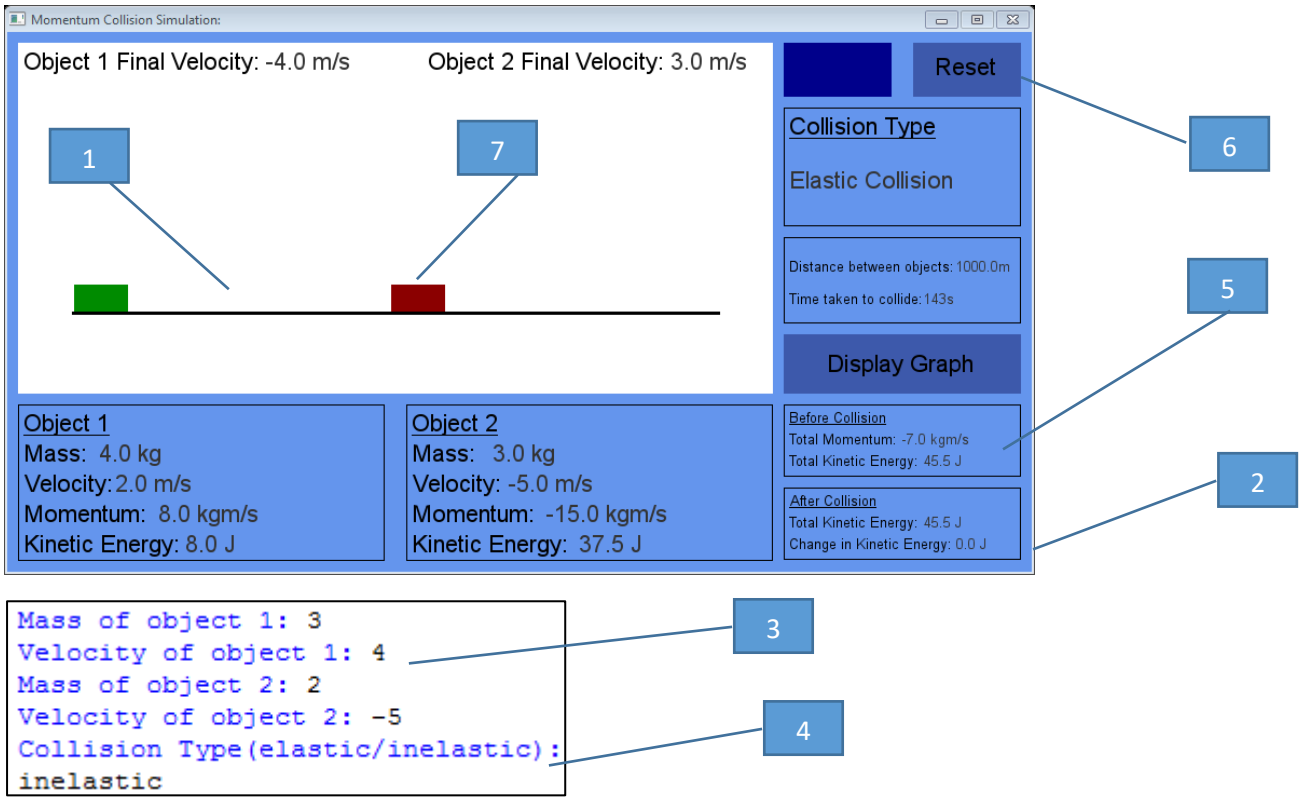
Final Review

Section 3 Review:

Success Criteria for this section:

1. Main window containing graphical user interface using Pygame 1.9.2a0 showing two objects (icons) on a flat surface.
2. Simple, easy to navigate graphical user interface.
3. Entry widgets for the velocity and mass of both objects.
4. Series of two radio buttons to allow users to choose between elastic and inelastic collisions.
5. Widgets on GUI displaying the final velocity of both objects, the total momentum and the change in the kinetic energy with their appropriate unit.
6. GUI including working start, reset and clear buttons.
7. Animation displayed on the GUI demonstrating the two objects colliding.
8. The points on the success criteria of the previous sections.

I labelled the GUI to show where each point on the success criteria has been met:



No further validation was needed in section 3 because the values displayed on the GUI where all calculated and validated in previous sections. I will need to make sure that this validation still works by regression testing.

Regression Testing:

To ensure that all points on my success criteria are met I retested everything from each section using the testing methods stated in my design for each section. I did this because during development of section 3 I may have made changes that could affect previous sections of code. I found that everything still worked as it should so no further development will be needed.

Review Summary:

For the most part my development for this section closely followed my design with very few changes. For the main algorithms in my program, the process is the same expect for the addition of some new variables and minor changes. Since pygame has no built in functions for entry widgets or radio buttons I decided it would be easier to use the python shell for inputting data then displaying it on the GUI. This means there is no need for a clear button since there are no entry widgets to be cleared. Instead, I added a display graph button which shows the GUI of the graph I created in section 2 when clicked. After finishing the final version of my program and testing it against the success criteria to ensure that everything works and all necessary features I mentioned in my design are included I am now ready to move onto evaluating my project.

Evaluation:

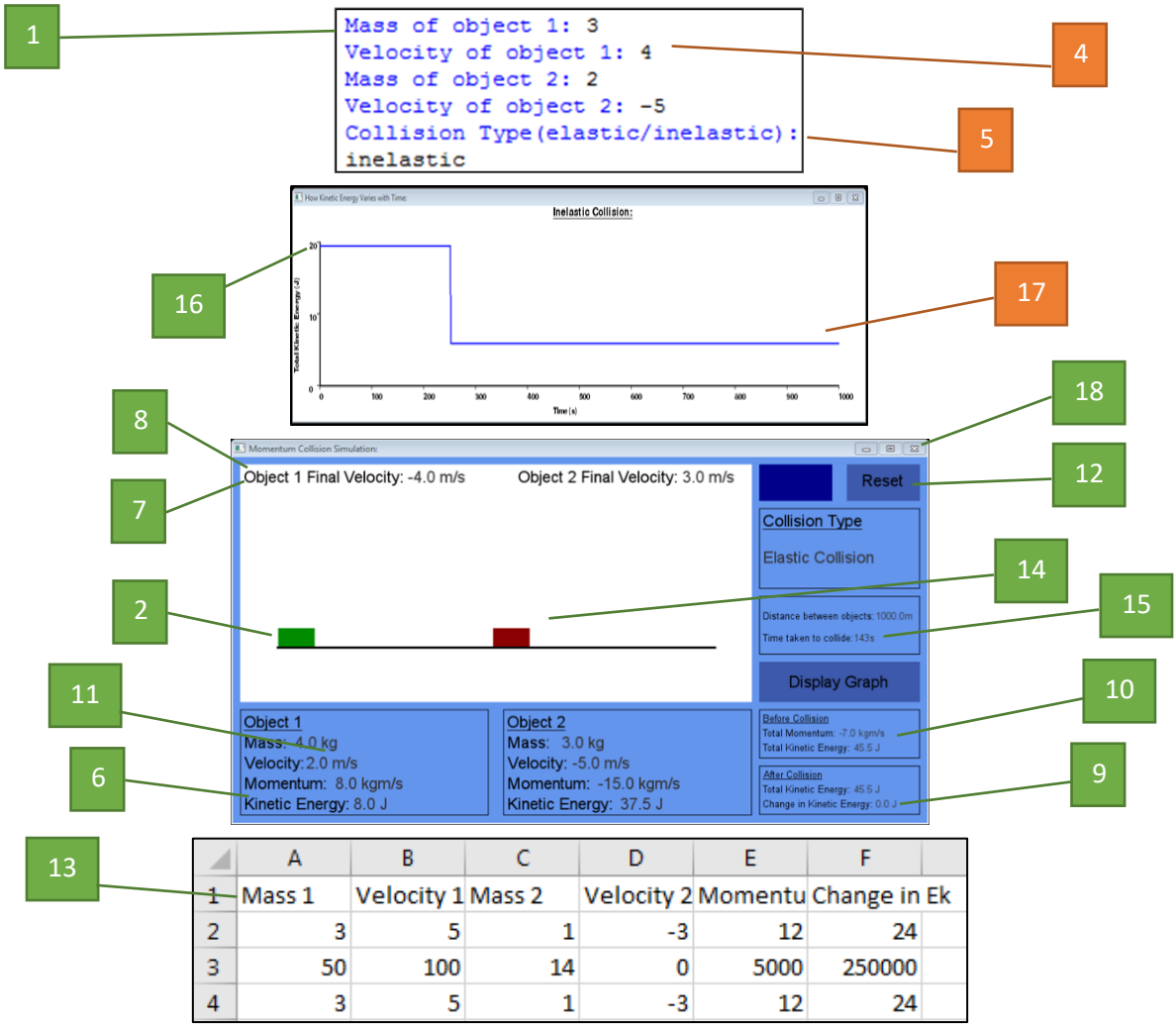
Success Criteria

Test Plan:

No.	Criteria:	Testing:
1	Program must run successfully using Python 3.3.2 without errors occurring.	Tested by checking robustness of program. Enough validation to ensure no errors occurred.
2	Main window containing graphical user interface using Pygame 1.9.2a0 showing two objects (icons) on a flat surface.	Tested by viewing GUI.
3	Simple, easy to navigate graphical user interface.	Tested by user feedback.
4	Entry widgets for the velocity and mass of both objects.	Tested by viewing GUI and entering data and comparing it to output.
5	Series of two radio buttons to allow users to choose between elastic and inelastic collisions.	Tested by viewing GUI and selecting different options then comparing to outcome.
6	Algorithms written using object oriented programming that work out the correct momentum and kinetic energy.	Tested by entering values and comparing them to results of other momentum/energy calculators.
7	Algorithm working out the final velocities after an elastic collision.	Tested by entering values and comparing them to results of other elastic collision calculators.
8	Algorithm working out the final velocity after an inelastic collision.	Tested by entering values and comparing them to results of other inelastic collision calculators.
9	Final velocity, total momentum and change in kinetic energy rounded to two decimal places.	Tested by entering values and comparing them to results of other collision calculators.
10	Widgets on GUI displaying the final velocity of both objects, the total momentum and the change in the kinetic energy with their appropriate unit.	Tested by viewing GUI.
11	SI units (kg, m/s, kgm/s and J) used for velocity, mass, momentum and energy.	Tested by viewing GUI.
12	GUI including working start, reset and clear buttons.	Tested by clicking buttons on GUI.
13	Database that stores the velocities and masses and the calculated total momentum and change in kinetic energy for an inelastic collision.	Tested by opening database and viewing data stored inside.
14	Animation displayed on the GUI demonstrating the two objects colliding.	Tested by viewing GUI.
15	Algorithm written using object oriented programming that calculates the time taken for the collision.	Tested by viewing code and GUI.
16	Algorithm that works out the points to be plotted on a kinetic energy against time graph.	Tested by selecting ‘Display Graph’ button on GUI.
17	GUI that displays the graph of kinetic energy against time.	Tested by selecting ‘Display Graph’ button on GUI.
18	An obvious way to end the program.	Tested by running program.

Test Evidence:

Some points tested by viewing GUI-



	A	B	C	D	E	F
1	Mass 1	Velocity 1	Mass 2	Velocity 2	Momentu	Change in Ek
2	3	5	1	-3	12	24
3	50	100	14	0	5000	250000
4	3	5	1	-3	12	24

Some points require further post development testing-

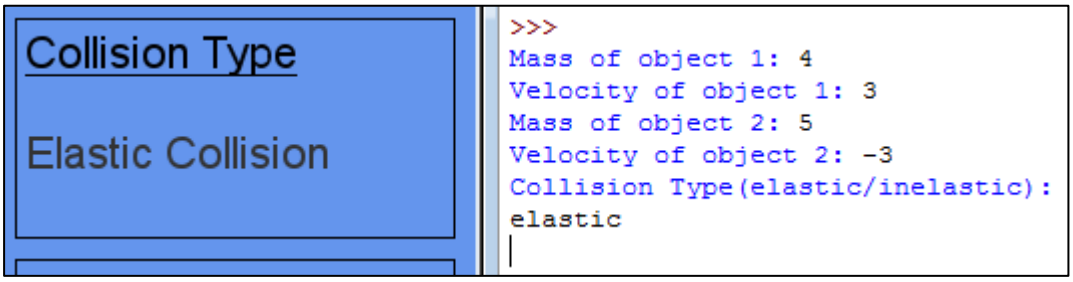
1. Program must run successfully using Python 3.3.2 without errors occurring:

```
def Validation(message,value): #Function that validates the user inputs,
                               #function recieves the message to be displayed and the type of data being inputted
    while True:
        try: #Allows user to try to input a value after the displayed message
            userInput = float(input(message)) #Input must be a float because velocity and mass don't need to be whole numbers
        except ValueError: #If the value entered is not a float a ValueError will occur so a statement will be printed and the loop repeated
            print("This is not a number\n")
            continue
        if value == 'm' and userInput <= 0: #If the value being inputted is a mass it needs to be greater than
            print("Mass must be greater than 0\n") #needs to be greater than 0 because mass can't be less than 0
            continue
        if value == 'v1' and userInput < 0: #If the value being inputted is v1 it needs to be greater or equal to 0
            print("Velocity of object 1 must be greater than or equal to 0\n")#So always moving in one direction (velocity is vector)
            continue
        if value == 'v2' and userInput > 0: #If the value being inputted is v2 it needs to be smaller or equal to 0
            print("Velocity of object 2 must be less than or equal to 0\n") #So moving towards other object so collision occurs
            continue
        else:
            return userInput
            break
```

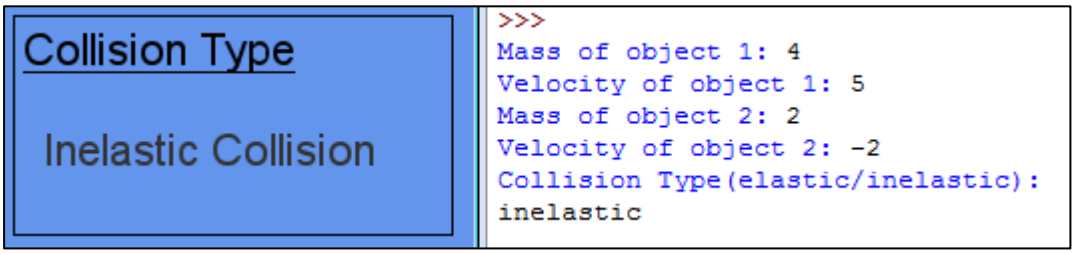
```
Mass of object 1: -2
Mass must be greater than 0

Mass of object 1: 3
Velocity of object 1: 4
Mass of object 2: 2
Velocity of object 2: 5
Velocity of object 2 must be less than or equal to 0

Velocity of object 2: -5
Collision Type(elastic/inelastic):
```

No errors occur because validation is successful.

5. Series of two radio buttons to allow users to choose between elastic and inelastic collisions:

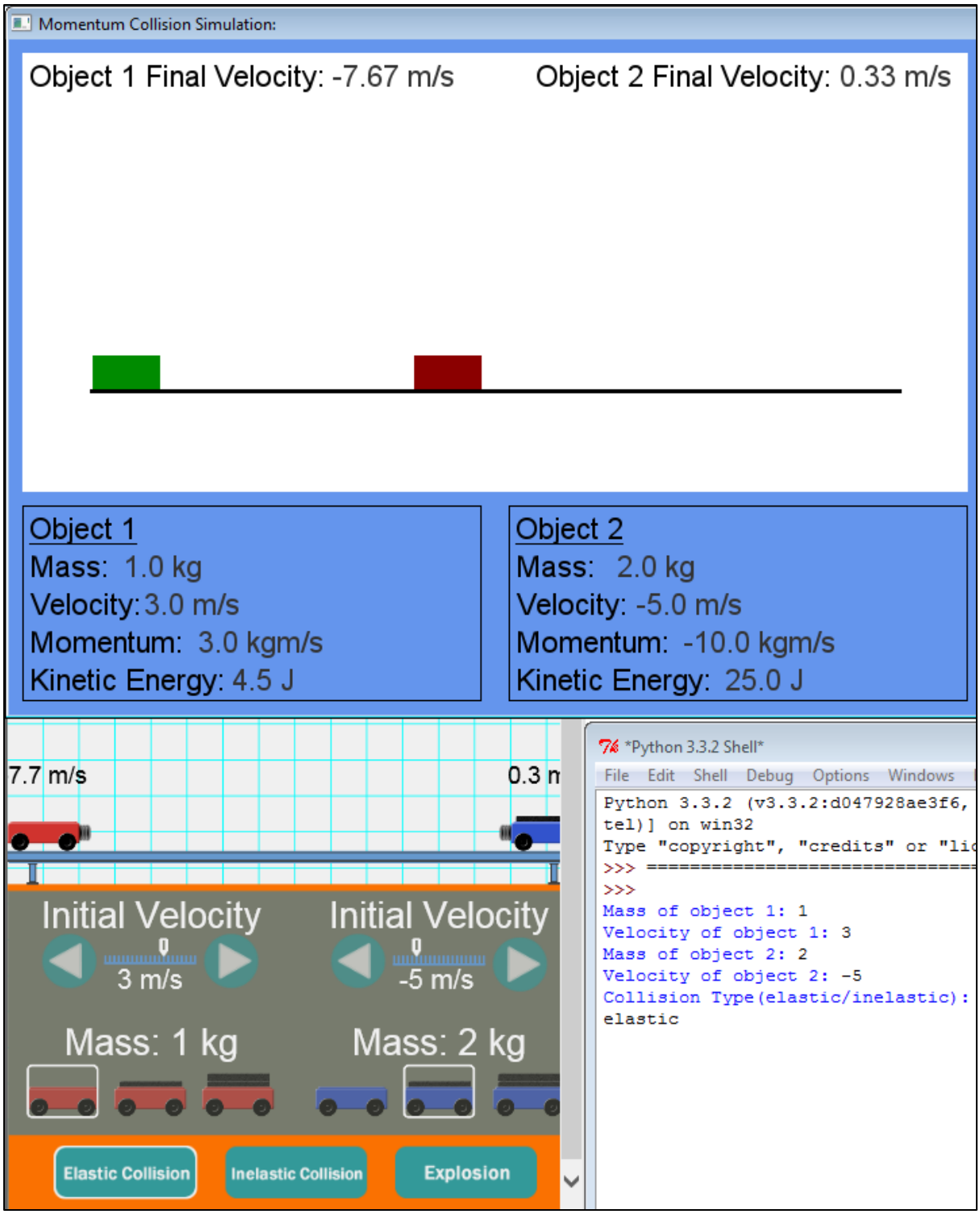


When elastic is entered, collision type is displayed as elastic.



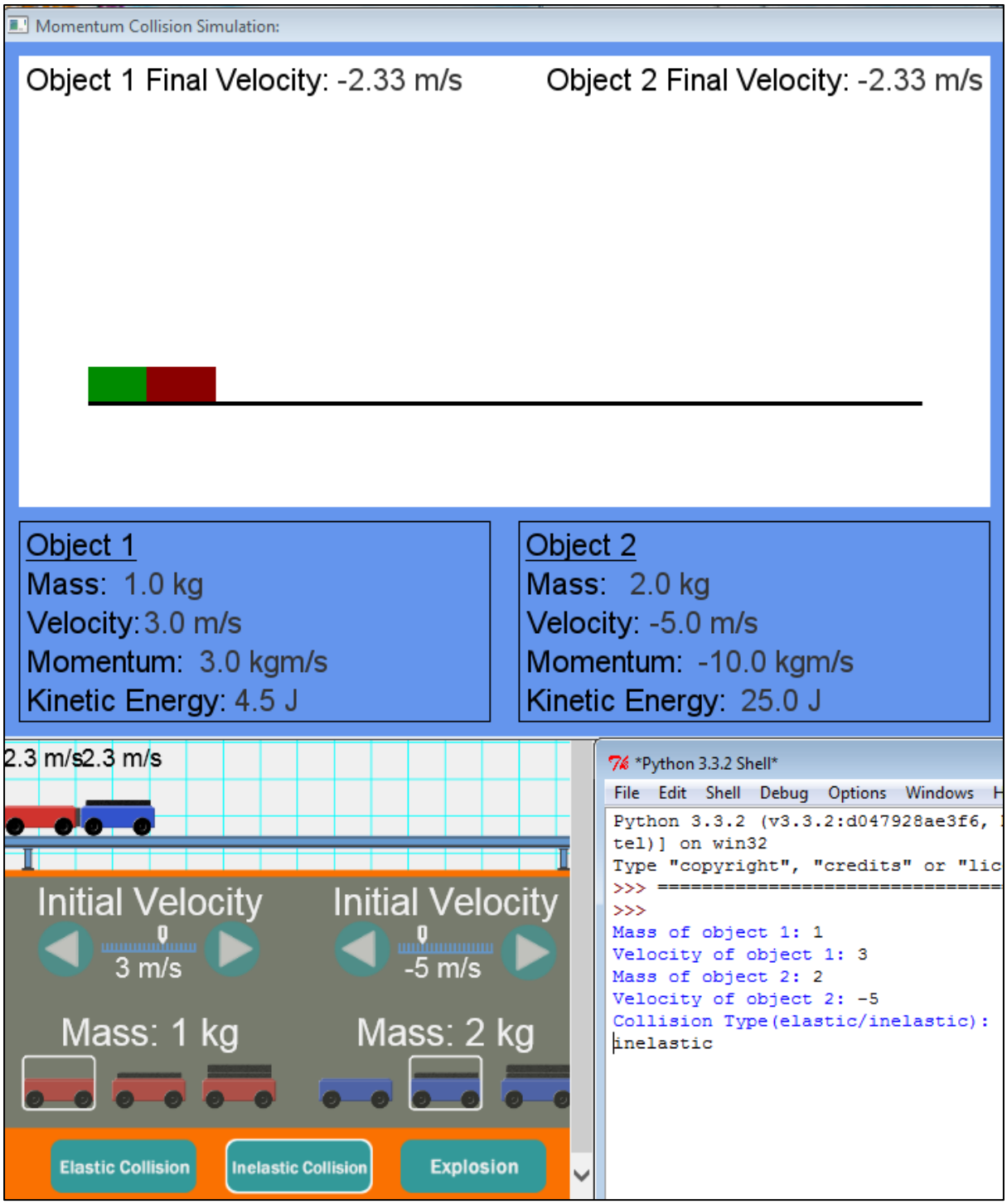
When 'inelastic' is entered, collision type is displayed as inelastic.

7. Algorithm working out the final velocities after an elastic collision:



Values entered for elastic collision match values displayed on GUI. Values on GUI are the same as values calculated by another collision simulator so calculations must be correct.

8. Algorithm working out the final velocity after an inelastic collision:



Values entered for inelastic collision match values displayed on GUI. Values on GUI are the same as values calculated by another collision simulator so calculations must be correct.

15. Algorithm written using object oriented programming:

```
## Mommentum & Kinetic Energy Class ##
class Momentum_Ek(object): #Calculates the momentum and kinetic energy of a particular object using its
                           #mass and velocity

    #Constructor
    def __init__(self, mass, velocity):

        #Attributes
        self.mass = mass
        self.velocity = velocity
```

```
## Collision Class ##
class Collision(object): #Calculaates the final velocity, final kinetic energy and change in Ek of objects
                         #after elastic or inelastic collision

    #Constructor
    def __init__(self, m1, v1, m2, v2, p_total, Ek_total):

        #Attributes
        self.m1 = m1
        self.m2 = m2
        self.v1 = v1
        self.v2 = v2
        self.p_total = p_total
        self.Ek_total = Ek_total
```

```
## Graph Class ##
class Graph(object): #Creates a graph of kinetic energy against time

    #Constructor
    def __init__(self, x1, x2, Ek_before, Ek_after, maxtime, v1, v2):

        #Attributes
        self.x1 = x1
        self.x2 = x2
        self.Ek_before = Ek_before
        self.Ek_after = Ek_after
        self.maxtime = maxtime
        self.v1 = v1
        self.v2 = v2
```

```
## Animation Class ##
class Animation(object): #Creates the objects and makes them collide

    #Constructor
    def __init__(self, v1, v2, x2):

        #Attributes
        self.v1 = v1
        self.v2 = v2
        self.x2 = x2
```

Multiple classes used for program so written in OOP.

Testing Evaluation:

No.	Criteria:	Status:
1	Program must run successfully using Python 3.3.2 without errors occurring.	Fully met.
2	Main window containing graphical user interface using Pygame 1.9.2a0 showing two objects (icons) on a flat surface.	Fully met.
3	Simple, easy to navigate graphical user interface.	?
4	Entry widgets for the velocity and mass of both objects.	Partially met – instead user enters values in python shell.
5	Series of two radio buttons to allow users to choose between elastic and inelastic collisions.	Not met – instead user choses by typing in python shell.
6	Algorithms written using object oriented programming that work out the correct momentum and kinetic energy.	Fully met.
7	Algorithm working out the final velocities after an elastic collision.	Fully met.
8	Algorithm working out the final velocity after an inelastic collision.	Fully met.
9	Final velocity, total momentum and change in kinetic energy rounded to two decimal places.	Fully met.
10	Widgets on GUI displaying the final velocity of both objects, the total momentum and the change in the kinetic energy with their appropriate unit.	Fully met.
11	SI units (kg, m/s, kgm/s and J) used for velocity, mass, momentum and energy.	Fully met.
12	GUI including working start, reset and clear buttons.	Fully met.
13	Database that stores the velocities and masses and the calculated total momentum and change in kinetic energy for an inelastic collision.	Fully met.
14	Animation displayed on the GUI demonstrating the two objects colliding.	Fully met.
15	Algorithm written using object oriented programming that calculates the time taken for the collision.	Fully met.
16	Algorithm that works out the points to be plotted on a kinetic energy against time graph.	Fully met.
17	GUI that displays the graph of kinetic energy against time.	Partially met – can only create graphs for values within certain boundaries.
18	An obvious way to end the program.	Fully met.

Review:

The test evidence for the points on my success criteria show in each case whether the point has been fully met, partially met or not met by pointing to the area on the GUI/database/python shell corresponding to that part of the criteria on my test evaluation table. The green boxes have been fully met whereas the orange boxes show partially or not met points.

I decided not to include the entry widgets and radio buttons on the GUI because pygame does not have built in functions for these elements and it would be too time consuming to create them myself since including them would not make much difference to the functionality of the program. The graph only works for certain values because some would be too large to fit on a normal screen using a scale that's clear enough to the user. To evaluate point 3 on my success criteria I will assess the usability features.

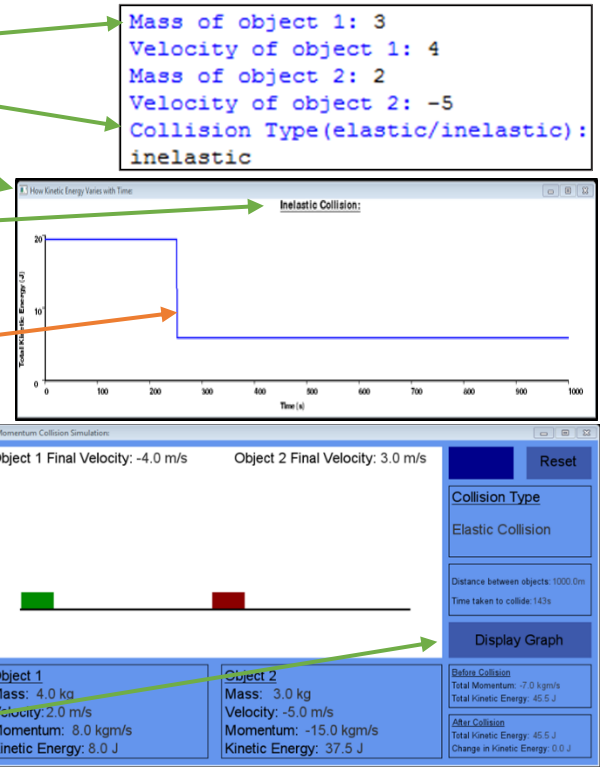
Usability

Testing:

Summary of Usability Features mentioned in Design section:

- Navigating the program is not overly complex.
- There is good communication with the user.
- Give the graph a title and label the axis of the graph with the value it shows and its unit.
- Make sure that all text on the graph is in a clear font of a reasonable size.
- Make the line plotted on the graph thick enough for people to see but small enough so that it is clear which point on the graph it corresponds to.
- Make sure that all text on the GUI is in a clear font of a reasonable size so that people with vision difficulties can read them and screen readers for blind people can detect the text.
- Make the colour of text stand out against the background colour so that it is easy to read allowing users to understand what is going on in the GUI.
- Make the buttons big enough that they are easy to click with the mouse.

Evidence of Usability Features:



Robustness:

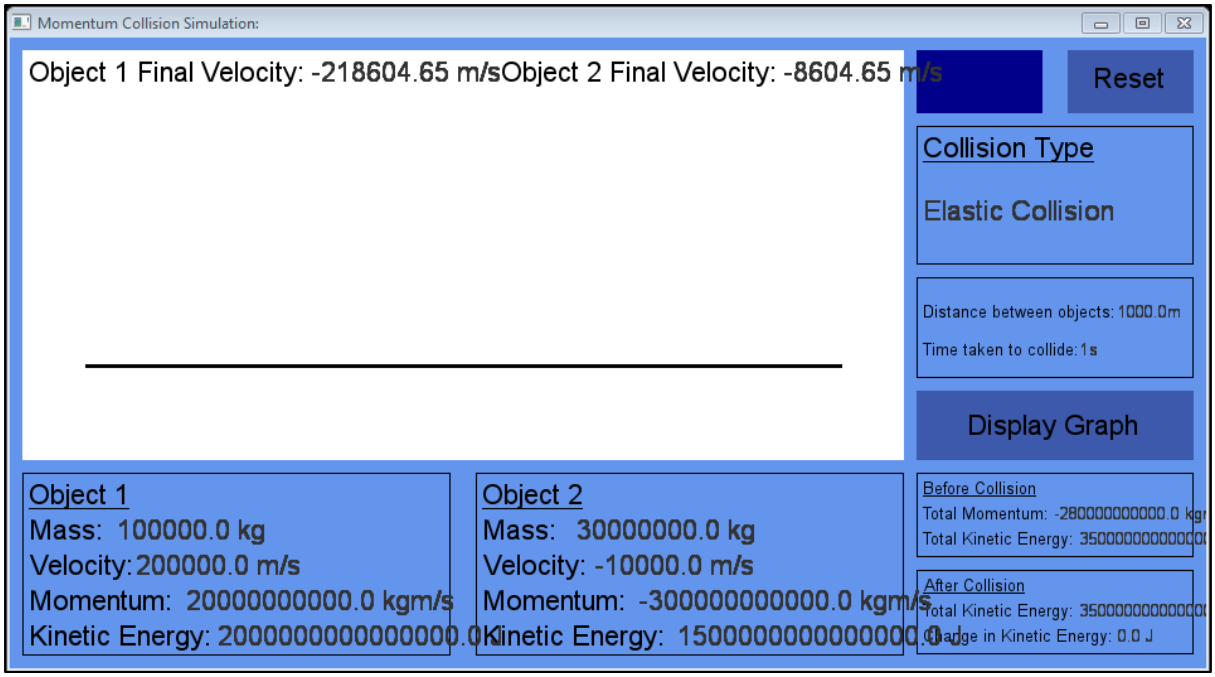
I made sure to include lots of validation to prevent errors from occurring in my algorithms for example the velocity of both objects can only be entered as opposite directions so that they collide and mass must be greater than zero since a value lower than this is not possible. Because of this, no errors occur that stop the program from running.

```
def Validation(message,value): #Function that validates the user inputs,
                               #function recieves the message to be displayed and the type of data being inputted
    while True:
        try: #Allows user to try to input a value after the displayed message
            userInput = float(input(message)) #Input must be a float because velocity and mass don't need to be whole numbers
        except ValueError: #If the value entered is not a float a ValueError will occur so a statement will be printed and the loop repeated
            print("This is not a number\n")
            continue
        if value == 'm' and userInput <= 0: #If the value being inputted is a mass it needs to be greater than
            print("Mass must be greater than 0\n") #needs to be greater than 0 because mass can't be less than 0
            continue
        if value == 'v1' and userInput < 0: #If the value being inputted is v1 it needs to be greater or equal to 0
            print("Velocity of object 1 must be greater than or equal to 0\n")#So always moving in one direction (velocity is vector)
            continue
        if value == 'v2' and userInput > 0: #If the value being inputted is v2 it needs to be smaller or equal to 0
            print("Velocity of object 2 must be less than or equal to 0\n") #So moving towards other object so collision occurs
            continue
        else:
            return userInput
            break
```

```
Mass of object 1: -2
Mass must be greater than 0

Mass of object 1: 3
Velocity of object 1: 4
Mass of object 2: 2
Velocity of object 2: 5
Velocity of object 2 must be less than or equal to 0

Velocity of object 2: -5
Collision Type(elastic/inelastic):
```

To prevent errors on the GUI I included further validation for example the ‘Display Graph’ option is only available when the graph plotted will not take up more than the whole screen. Unfortunately, an error does occur when the calculated values are too high so they take up too much space on the GUI and go over the set boundaries. This does not stop the user interface from working however; it can cover up important information and does not display an animation.

```
if self.Ek_before <= 900: #works out scale of y axis depening on maximum Ek
    scale = 100 #100 points on GUI represents 100J
    size = 1
    if self.Ek_before <= 90: #bigger scale of there is space
        scale = 10 #100 points on GUI represents 10J
        size = 10
    #<= 900 or 90 because anything greater than that is too big for screen
```



Review:

The evidence for my usability features show in each case whether the point has been met by pointing to the area on my final program corresponding to that feature in the table. Green arrows show successful features, orange arrows show partially successful features. The third point is orange because it is not always clear which value the points on the graph correspond to since the axis is not very precise. The fourth point is orange because the text is not always a clear, reasonable size since sometimes the numbers are too long and go over the boundaries.

Since I am the person that created this project, I will need to test it with other users to ensure my usability features work effectively. In order to do this I will need to receive user feedback from my stakeholder and students that will likely use this program.

User Feedback

Users:

Overall, the user feedback I received from students was positive since it is an easy program to follow however; they suggested that I could make the input more efficient by allowing the user to type i/e instead of needing to type inelastic/elastic. Another suggestion was that I make the velocity of the second object automatically opposite to the first object so that it does not need to be validated. This would also make the program more efficient although the purpose of this project is to help educate students so forcing them to enter one of the velocities as negative helps them to understand vector quantities. Some users complained that the animation was too fast to view for certain inputs which I could fix during further development.

Stakeholder:

I let my stakeholder Mr Hutchinson test my program and he seemed overall pleased with the result. He mentioned that the "big, clear, projectable images are very useful since students need to see the collision happen as much as see data varying" and that it is "good for demonstrating vectors to students" however he would like to see "vector arrows if possible showing magnitude and direction of each object to further their understanding". He also said that he liked how "simple and clear the program is compared to many others" but did not like that you need to restart the program to enter new data. I then asked Mr Hutchinson if there was any other features he would like to see added to the program and he suggested a "multiple choice quiz to test students' knowledge of momentum and vectors".

Limitations

My final solution has limitations such as:

- There is no option for explosion collisions because my simulation is aimed at students studying GCSE physics and explosion collisions are not on their specification.
- It only shows collisions on a flat surface because GCSE students do not need to learn about angular collisions.
- It only shows collisions of objects traveling at a constant speed (not accelerating) because the simulation is meant to demonstrate the idea of momentum and this will be easier for students to understand if the speeds are constant.
- The values inputted and outputted only work in SI units because these are the standard units and GCSE students should be able to convert other units into SI units so there is no point in adding a unit converter to my program.
- The animation ends after a certain amount of time because otherwise the objects would continue moving forever since they travel at a constant speed so wont decelerate.
- The simulation does not include sound because I feel this does not add anything to the project that makes it more suitable to the stakeholders needs since it is not educational and will likely just distract users from the varying data.
- Users cannot change the speed of the simulation or pause it.
- Users cannot change the amount my program rounds its calculations by.
- The GUI does not display any arrows to represent the vectors.
- The graph of total kinetic energy against time can only be displayed for certain values.

Further Development

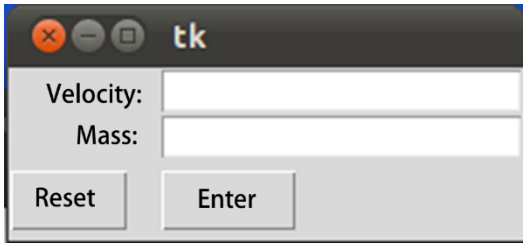
Success Criteria:

Points that were not fully met:

1. Entry widgets for the velocity and mass of both objects.
2. Series of two radio buttons to allow users to choose between elastic and inelastic collisions.
3. GUI that displays the graph of kinetic energy against time.

Addressing points in further development:

1. This point was only partially met because pygame does not include a built in function for entry widgets so the user input is done via the python shell. I could fix this issue in further development by using a different program for the GUI with more built in functions such as tkinter or by programming an entry widget function myself given more time.

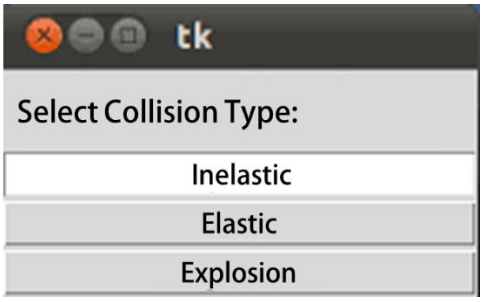


Entry widget example using tkinter

Algorithm for entry widget using pygame, repeat for each number (0-9)

```
while widget selected:
    for event in pygame.event.get():
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.0:
                display.blit("0")
            if event.key == pygame.1:
                display.blit("1")
            if event.key == pygame.2:
                display.blit("2")...
```

2. This point was not met because pygame does not include a built in function for radio buttons so the user input is done via the python shell. I could again fix this issue in further development by using a different program for the GUI with more built in functions or by programming a radio button function myself given enough time.

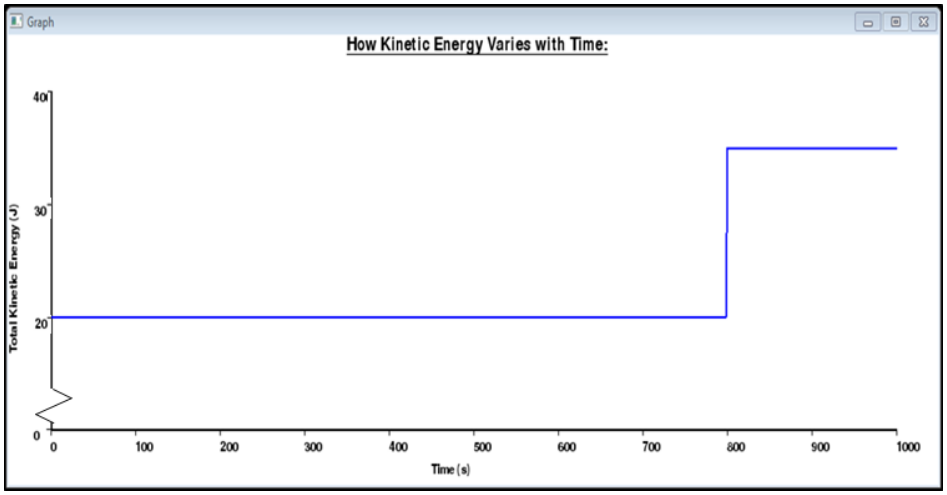


Radio button example using tkinter.

Algorithm for radio button, using variables already in program.

```
Button1 = drawRect(100,10)
Button2 = drawRect(200,20)
displayText(font,"Inelastic",100,10)
displayText(font,"Elastic",200,20)
if mouse click within Button1:
      type = "Inelastic"
else:
      type = "Elastic"
```

3. This point was only partially met because some graphs where too big to fit on the screen but if I changed them to a smaller scale the user would not have been able to see the important details on the graph. To address this issue in further development I could change the graph algorithm to make these graphs on a smaller scale but provide a way to zoom in on the GUI so the details can be visible. An alternative option would be to change the algorithm so that graphs that are too big have an axis that does not start at zero so it fits on the screen more easily.



Graph with break on y-axis so that it does not need to be as long. Means more can fit on the monitor while being clear.

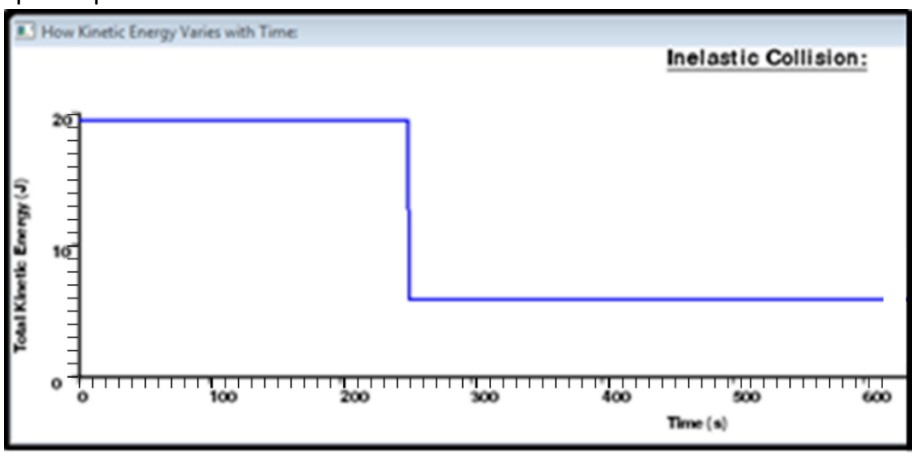
Usability:

Features that were not fully successful:

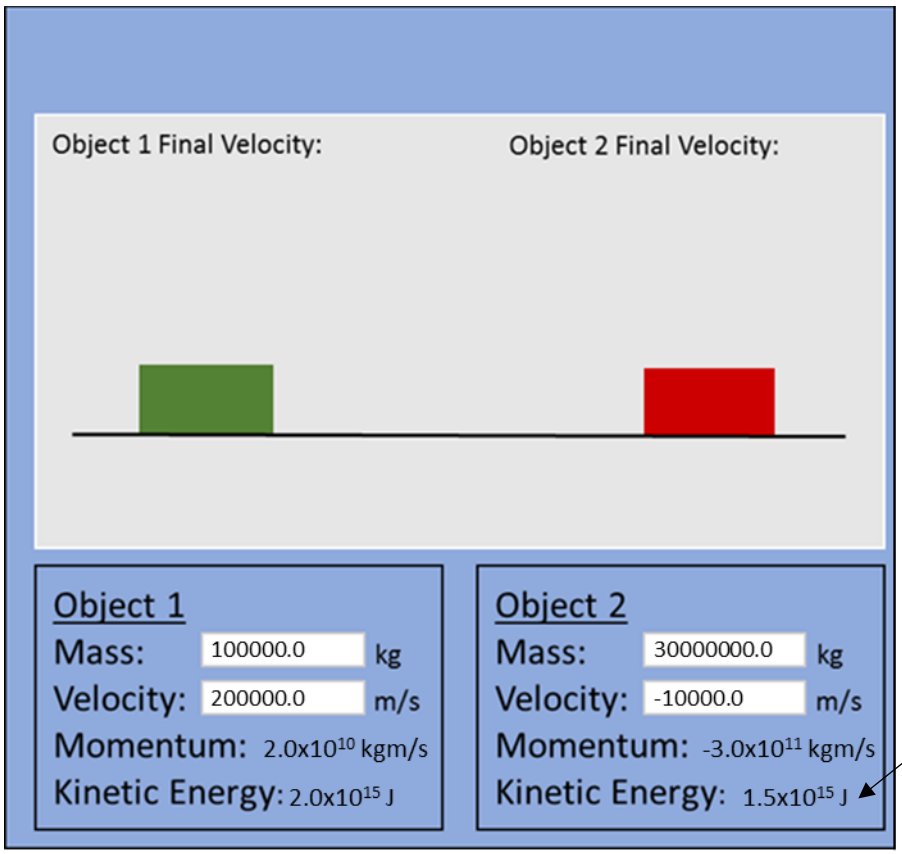
1. Make the line plotted on the graph thick enough for people to see but small enough so that it is clear which point on the graph it corresponds to.
2. Make sure that all text on the GUI is in a clear font of a reasonable size so that people with vision difficulties can read them and screen readers for blind people can detect the text.
3. Enable user to enter multiple sets of data and see the results.

Addressing features in further development:

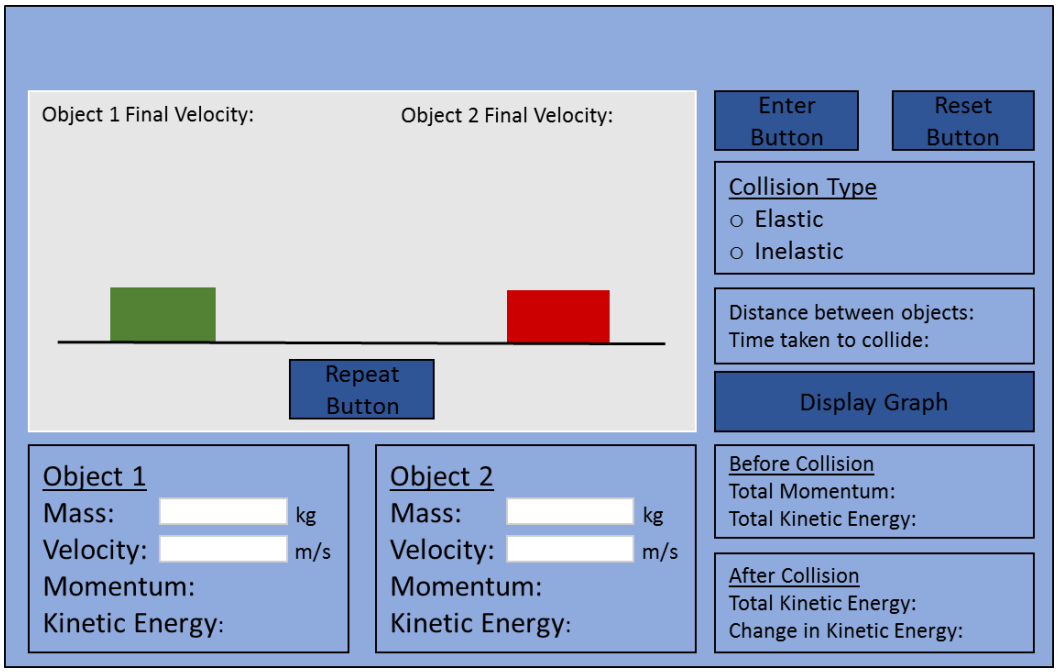
1. This point was only partially successful because the graphs axis are not very precise so it is not always clear which value a point is if it's in between two marked points on the axis. To fix this issue I could draw more lines between the numbered points on the axis at an equal space apart like on a ruler to make it clearer.



2. This point was only partially successful because the text is not always a clear, reasonable size since sometimes the numbers are too long and go over the boundaries. This could be addressed in further development by implementing more validation so the number cannot reach a value high enough to be long enough to go over the boundary. The issue with this resolution is that the user may need to enter high values so instead, I could make the GUI display larger numbers in smaller text or I could put high values in index notation since they are already rounded.



3. This point was not met because the program has to be restarted each time you want to input new data. To fix this issue I could make the 'reset' button reset the values instead of the animation and include a new 'repeat' button to repeat the animation.



Limitations:

These potential improvements/ changes to my solution have limitations such as:

1. Some graphs may still be too big to fit on the screen.
2. Text might need to be so small for some numbers to fit that it is unreadable
3. Users may not understand index notations.
4. Users cannot change the speed of the simulation or pause it.
5. The GUI does not display any arrows to represent the vectors.

The program could be developed to deal with these limitations by:

1. Still including validation to ensure the display graph buttons does not draw a graph if the graph would take up more space than the whole screen.
2. I could write a function that allows user to zoom in on the GUI to read the numbers or write it in index notation which is always the same length.
3. Make it so that if the user hovers their mouse over a number written in index notation it explains how it works and what number this represents.
4. Include a slider in the GUI allowing the user to speed up and slow down the animation.
5. Include an arrow above each object after the start button is pressed representing its vector.

Maintenance

Multiple maintenance issues could occur during further development such as:

- Making changes or developments to one part of my program could unintentionally have an impact on other parts of it stopping it from working properly.
- Corrective changes because it may be hard to find the correct place to do the changes. It could be difficult to recognize parts of the code if I do not work on them for a long time.
- Further development can be very time consuming because of all the other maintenance issues meaning you have to take extra steps to ensure the development does not have an impact on other parts of your program.
- Maintaining the database so that it doesn't take up too much space and keeps working as intended.

Dealing with maintenance issues:

- Impact Analysis – In order to keep previously developed parts of my program working I would need to find out the effects a proposed modification would have on the rest of my code. Impact analysis is the action of assessing the probable effects of a change with the plan of reducing sudden side effects.
- Commenting code and using appropriate variable names will make it much easier to develop my program in the future since it will be easy to understand and follow.
- Documenting all further development in detail like I have done with my previous development so it is easy to keep track of changes that have been made.
- Good time management and organisation to ensure development is completed within a certain amount of time.
- Checking on the database after certain amounts of time to ensure it is maintained by removing repeated data and checking that data is still being stored correctly.

Conclusion

In my analysis, I stated that the purpose of my project is to simulate collisions between two objects on a flat surface to show people how momentum effects different types of collisions. I planned to create a momentum collision simulator written in Python 3.3.2 that allows a user to input the mass and velocity of different objects then receive an output for the total momentum, change in kinetic energy and the speed after the collision. I decided to make the target group GCSE Physics students since they could use the simulation to better their understanding of momentum collisions. Because of this, I chose Mr Hutchinson (a Physics teacher) to be my stakeholder.

In order to ensure I achieved these goals I created a success criteria to test my final program against, interviewed my stakeholder to get user requirements and planned ahead by designing my project before developing it, researching things such as momentum and the GCSE physics specification and by using computational methods such as problem recognition, abstraction and decomposition.

I believe I have fulfilled the objectives of my program since testing proved that all the major points on my criteria have been met, user feedback was mostly positive and my stakeholder was overall pleased with the outcome of my project. Through completing this task I have gained valuable skills such as time management, organisation/planning, object oriented programming and designing GUIs which will help me to maintain this program and when working on future projects.

My final product differs from my original plan in a few ways for example; the layout of the GUI is slightly different from the layout in the design section, which was an intentional change to make the

window look better. Another change is that entry is done via the shell instead of entry widgets; this is an example that I would fix during further development.

If I were to redo my programming task, I would still choose the same project however, there are some things I would do differently to remove many of the current limitations. I would include a momentum quiz like my stakeholder suggested to test students' understanding; I would display high numbers in index notation so they fit in the window properly, I would allow the user to change the speed of the collision so it is easier to view and I would show the vectors of the objects.

In conclusion, my program effectively fulfils all the original aims of my project however; I will continue to work on it in order to keep it maintained and develop it further to make it more suitable for my stakeholder, to improve my programming skills and to implement my ideas for further development.

Bibliography:

People

Rouse.M (2008). – Used for meaning and definition of object oriented programming

Books

Parsons, R. (2011). GCSE Physics The Revision Guide, GCP

-Used for GCSE Physics example questions and workings on momentum and collisions

Websites

Source:	**Use:**
https://filestore.aqa.org.uk/resources/physics/specifications/AQA-8463-SP-2016.PDF	Specification for AQA GCSE physics.
https://dictionary.cambridge.org/dictionary/english/simulation	Definition of a simulation.
http://convertalot.com/elastic_collision_calculator.html	Elastic collision formulas and example of collision simulation.
http://www.physicsclassroom.com/Physics-Interactives/Momentum-and-Collisions/Collision-Carts/Collision-Carts-Interactive	Example of momentum collision simulation.
https://phet.colorado.edu/sims/collision-lab/collision-lab_en.html	Example of momentum collision simulation.
https://searchmicroservices.techtarget.com/definition	Definition of object oriented programming and inheritance.