REPORT ON THE VERLET INTEGRATOR

Introduction

The aim of this report is to explain the Verlet Integrator we developed.

Our goal for this project is to have an interactable integrator that, given an initial data, in our concrete case the initial position x and y, speed vx and vy, acceleration ax and ay, a radius, a density and elapsed time, it computes their values at the end of that given time. Our intention is that it will also have a way to test the correct function of the integrator, which will print every frame and it will be able to pause it and a graphic representation. In addition, the Newton's Laws will be implemented in order to compare the results of the integrator with the final data and see the accuracy and the standard deviation.

Our frame rate will be of 60 fps. Air density will be implemented, as well as gravity. We will have a ground and the ball will be able to collide with it. Frame per frame we are going to be calculating each variable to update it.

To calculate the force in the x axis we will use the following formula:

```
fx = 0.5 * AIR_DENSITY * new_vx * new_vx * area * CD
```

```
                                                    Where:
                                                fx - force in the x axis
                                AIR_DENSITY - denstity of the air, constant
                                        new_vx - the velocity for this frame
                                          area - area of the object (sphere)
                                                CD - drag coefficient
```

From the Verlet Integrator we know the following:

In case we didn't need the speed, the formulas we would use would be:

```
new_vx = vx + new_ax * dt
```

```
                                                    Where:
                                            new_vx - the velocity for this frame
                                        vx - the velocity of the previous frame
                                    new_ax - the acceleration for this frame
                dt - the delta time (elapsed time) between the previous frame and this one
```

```
new_x = new_x * 2 - x + new_ax * dt * dt
```

```
                                                    Where:
                                            new_x - the position for this frame
                                        x - the position of the previous frame
```

```
dt - the delta time (elapsed time) between the previous frame and this one
                                    new_ax - the acceleration for this frame
```

However, we are going to be using the velocity. Then, to compute the acceleration and the speed we will use MRUA, which is used to calculate the Velocity Verlet. In this case, the acceleraion would be take as constant and we would have:

```
new_vx = vx + new_ax * dt

                                                                    Where:
                                       new_vx - the velocity for this frame
                                    vx - the velocity of the previous frame
                                    new_ax - the acceleration for this frame
dt - the delta time (elapsed time) between the previous frame and this one

  new_x = new_x * 2 - x + new_ax * dt * dt

                                                                    Where:
                                        new_x - the position for this frame
                                   x - the position of the previous frame
dt - the delta time (elapsed time) between the previous frame and this one
                                    new_ax - the acceleration for this frame
```

Then again, we take into account that the acceleration may not be the same through all of this process and, therefore, the formulas we are finally coing to use are:

```
new_ax = fx / mass

                                                                    Where:
                                    new_ax - the acceleration for this frame
                                                 fx - force in the x axis
                        mass - mass of the object, given a radius and a density

new_vx = vx + new_ax * dt

                                                                    Where:
                                       new_vx - the velocity for this frame
                                    vx - the velocity of the previous frame
                                    new_ax - the acceleration for this frame
dt - the delta time (elapsed time) between the previous frame and this one

new_x = x + vx * dt + (new_ax / 2.0) * dt * dt

                                                                    Where:
                                        new_x - the position for this frame
                                   x - the position of the previous frame
                                    vx - the velocity of the previous frame
dt - the delta time (elapsed time) between the previous frame and this one
                                    new_ax - the acceleration for this frame
```

We will use Newton's Laws to compare the results of the integrator with the "reality", as stated before. To compute the acceleration, the velocity and the position we will use the MRUA formulas:

```
new_ax = fx / mass

                                                                    Where:
                                    new_ax - the acceleration for this frame
```

```
                                                    fx - force in the x axis
                                    mass - mass of the object, given a radius and a density

                        new_vx = vx + new_ax * dt;

                                                                        Where:
                                                        new_vx - the velocity for this frame
                                                    vx - the velocity of the previous frame
                                                    new_ax - the acceleration for this frame
                                dt - the delta time (elapsed time) between the previous frame and this one

                    new_x = x + vx * dt + (new_ax / 2.0) * dt * dt

                                                                        Where:
                                                        new_x - the position for this frame
                                                    x - the position of the previous frame
                                                    vx - the velocity of the previous frame
                                dt - the delta time (elapsed time) between the previous frame and this one
                                                    new_ax - the acceleration for this frame
```

To compute the forces, the acceleration, the speed and the position for the y axis, we will use the formulas stated above as well, but taking into account the gravity, as said before.

To see further information on where we took our information on what formulas to use from, please take a look at our "Data" folder in our GitHub repository (ht HYPERLINK "https://github.com/Needlesslord/Physics2theory"tps://github.com/Needlesslord/Physics2theory) the following web pages:

https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration HYPERLINK "https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration.html".html

https://en.wikipedia.org/wiki/Verlet_integration

ht HYPERLINK "https://www.gamedev.net/articles/programming/math-and-physics/a-verlet-based-approach-for-2d-game-physics-r2714"tps://www.gamedev.net/articles/programming/math-and-physics/a-verlet-based-approach-for-2d-game-physics-r2714

To summarize, the final result of the integrator should have a welcome and small tutorial/explanation of how it works, the input of the data, then select whether the user wants to test the integrator or they want only the final results. If they choose the first option, every frame will be printed on the console, which can be paused, showing the data of the last frame, a pause signal and the results in that same position calculated with Newton's Laws and that can be unpaused again to continue the test. On the other hand, if the user chooses to only show the final data, the initial data, the final data calculated with the integrator, the final data calculated with Newton's Laws and a graphic representation will be shown. In both cases will be possible to go to the main selection again once finished.

Implementation

The Verlet Integrator is a program that simulates how an object would perform inside a game with physics. For this simulation we have decided to use a cube, since it is the most used shape in collisions because it is a shape that does not use a lot of resources. Also it is easier to apply friction to it than to a sphere.

When the program starts, it asks the user to enter some data to know from what state it has to start calculating the cube data over time. Input is done with the C++ standard input provided with iostream.

It takes into account the forces that the object receives to recalculate the acceleration of the object, its velocity, and its position. It is also prepared to calculate the aerodynamics forces depending on the medium the object is. Vertical and horizontal forces, as well as a coeffient mu for friction can be added. In addition, the user will be able to choose whether the collision will be elastic or completely inelastic.

The integrator works calculating the new object data taking into account the previous frame data. The frame rate chosen for our integrator is 60 fps, which is the common frame rate in a large amount of videogames. To change the frame rate, the only thing needed to do is to change the global value of the fps variable. This will automatically change the time step.

It is also able to calculate when an object is collisioning with another object, calculating their dimensions and the distances between them. But there is no recalculation of the objects data after collisioning. This means that the integrator does detect collisions, but does act on them. That can be used in a game to make games collide in-game, or to use objects as detectors.

The distance between two cubes is calculated taking as reference the center of each object, and subtracting half the length of an edge of both cubes to make the distances more accurate.

There are two modes in the integrator, which are called "TEST OF THE INTEGRATOR" and "FINAL DATA":

- TEST OF THE INTEGRATOR: When the object information is displayed on screen, this is shown per frame. To make it easier for the user to read the time information, apart from showing the frames passed from the beginning, also is shown the second and the frames passed on that second (each second has as much subdivisions as fps).

- FINAL DATA: This mode only shows the data inputed by the user, the data of the last frame calculated by the integrator, the data of the last frame calculated

with the Newton's Laws and allows the user to show a simulation/representation of the cube's movement.

To prove the efficiency of the integrator we've designed a function that compares our results with the ones we would obtain if we applied the traditional laws of kinematic without the "Verlet" method.

This function takes the same inputs used for the integrator and calculates position, velocity and acceleration at the last frame. These are the equations we've used:

Since we wanted to do a graphic representation/simulation of the object's movement, we tried to display a window from the console, but, after many days trying, we decided that an easier solution was to create an interface for our integrator, althought it required more time. We designed each screen, implemented module fonts to get the input and now the user, if they want to, can see how the cube would move in the conditions they gave. We decided that the input should be saved, instead of erased every time the Integrator is used, in case the user only wanted to change a digit, which can be done by deleting and rewriting. In the code, we implemented "steps" to change between screens. Each screen also has the instructions on how to move between them and it is easy to understand what you can do in each of them. In addition, we have a screen to welcome the user and a screen to tell them how the integrator works. The keys used are numbers to input the data, C to clear all, B to go to the previous screen, ENTER/RETURN to continue, W and S to move up and down and ESC to leave the app.

In order to make a graphical representation of how the motion would work we designed a new screen that would show a simple background previously done with just a line representing the surface delimitating the space so that the user is informed about the position of the ground.

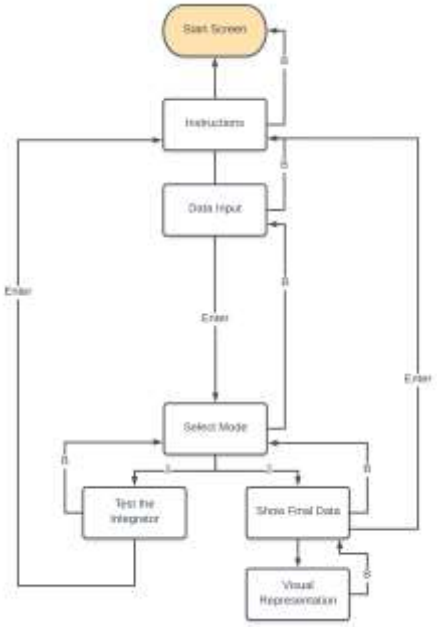
At the same time it will show a square representing the object, which acquires the propierties from the user input, and will move accordingly to the motion it should do. To achieve this we had to make a state inside the integrator that would call a function to draw the square with the correct position while the camera follows it.

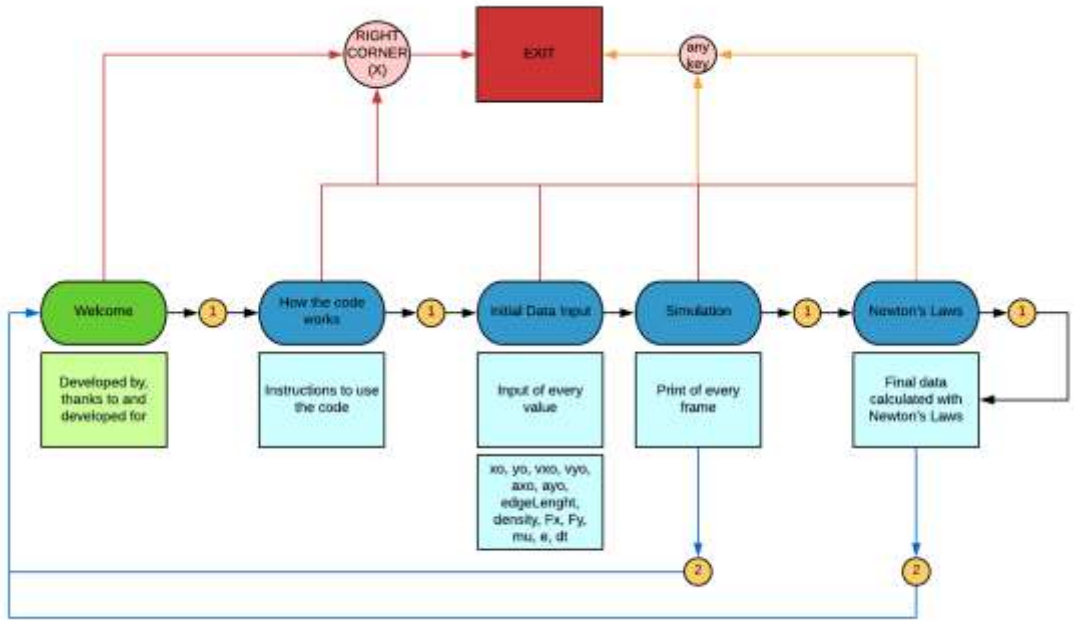
To get to this screen, once the user reaches the Final Data screen he must press the key '1' and it will pop. During the representation the user can press 'B' to go back to the Final Data screen.

We ended up trying to implement more things than expected. Our first intention was to do a simple integrator controlled with the console and with a graphic representation. But, since the simulation ended up being a challenge, we decided to build an interface, which visually is more appealing to the user. In

addition, there are 3 different forces that the user can control and the collision can be made elastic or completely inelastic.

However, we unexpectedly found a crash in the code and couldn't implement neither the interface nor the graphic representation. Therefore, we ended up in the console again, making it interactuable. Below you will be able to see the flowchart of both the hypothetical interface and our final design for the console, respectively.





To summarize, we have an interactuable console where you have a welcome, a small tutorial and then the input of the initial data. Afterwards, all the frames of the simulation are printed and, when finished, you can choose whether to calculate it also with Newton or to restart the simulation.

Results

The tests we are going to run are the following:

(1) That the code does not crash in any way during the simulation or the different steps.

(2) N number of simulations where each variable is set to 0.

(3) All data set to 0.

(4) 5 different simulations with random numbers.

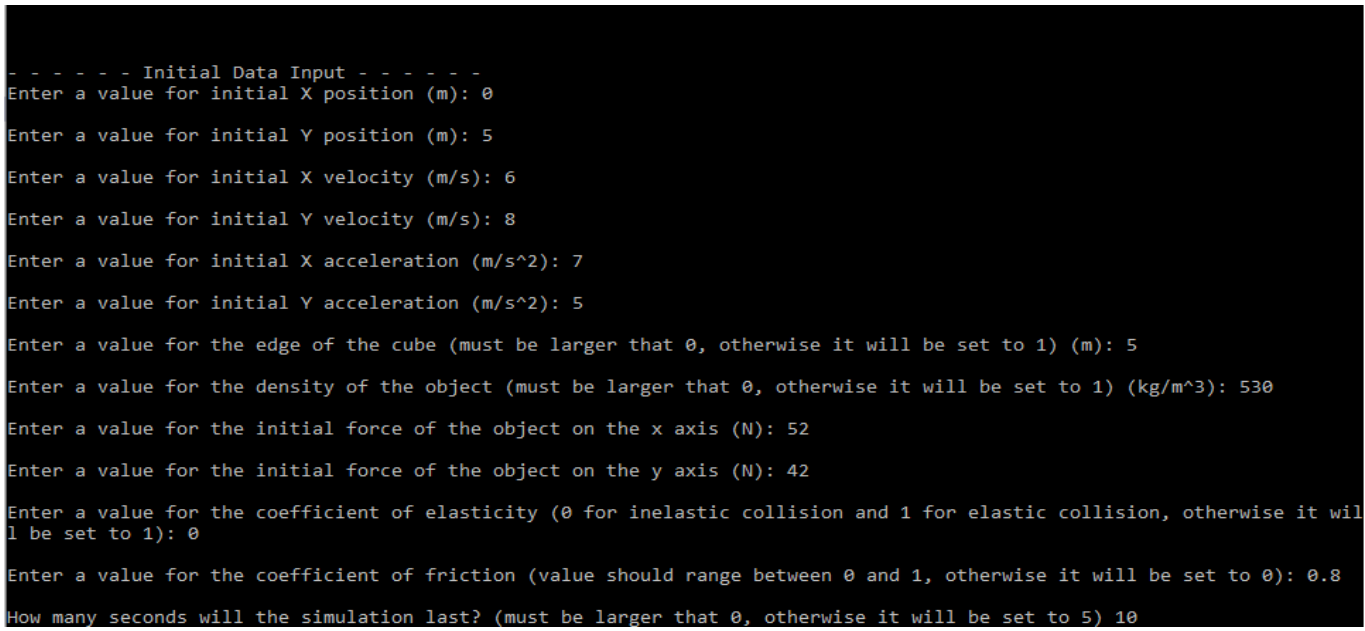
(5) Calculate the standard error deviation with the data calculated with the integrator and the Newton's laws.

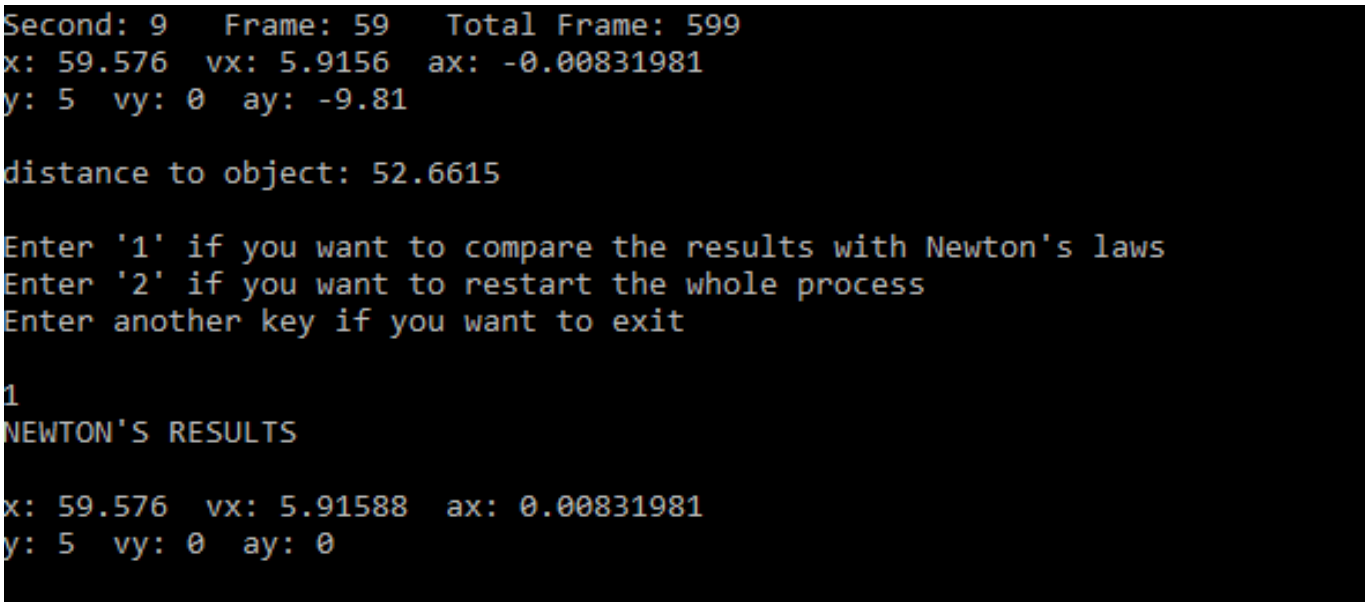
(1) We have been testing the integrator and has not crashed. It flows well between instructions/steps and the data inputed is recognized well.

(2) The results make sense and any "nan" or any other weird value is shown. In addition, the collisions work well, since it is not printing "Colliding!" in every frame. Each frame is shown, and they match the time they are in. The procession also makes sense, and none steps (frames) are missed. Below all the initial data and final data, both calculated with the integrator and Newton's Laws will be shown. In addition, in (5) you will be able to see the standard error deviation.

xo = 0, all the other data is random

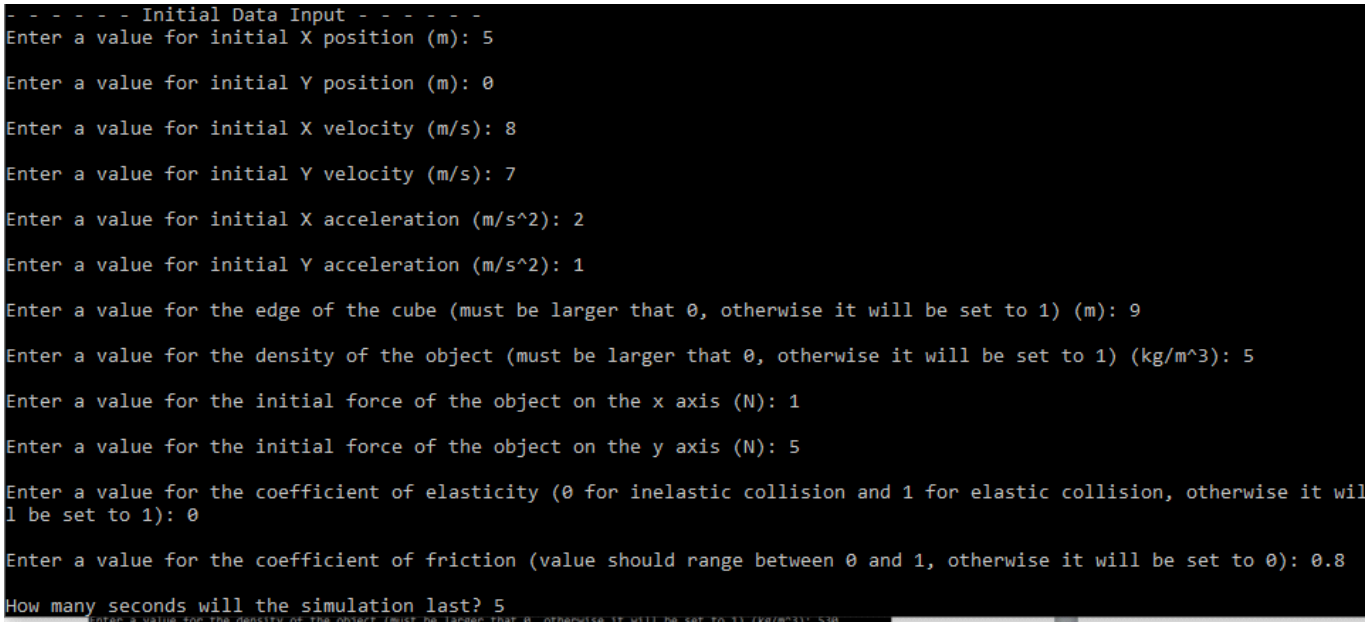
The simulation has run well. Below we can see the results:

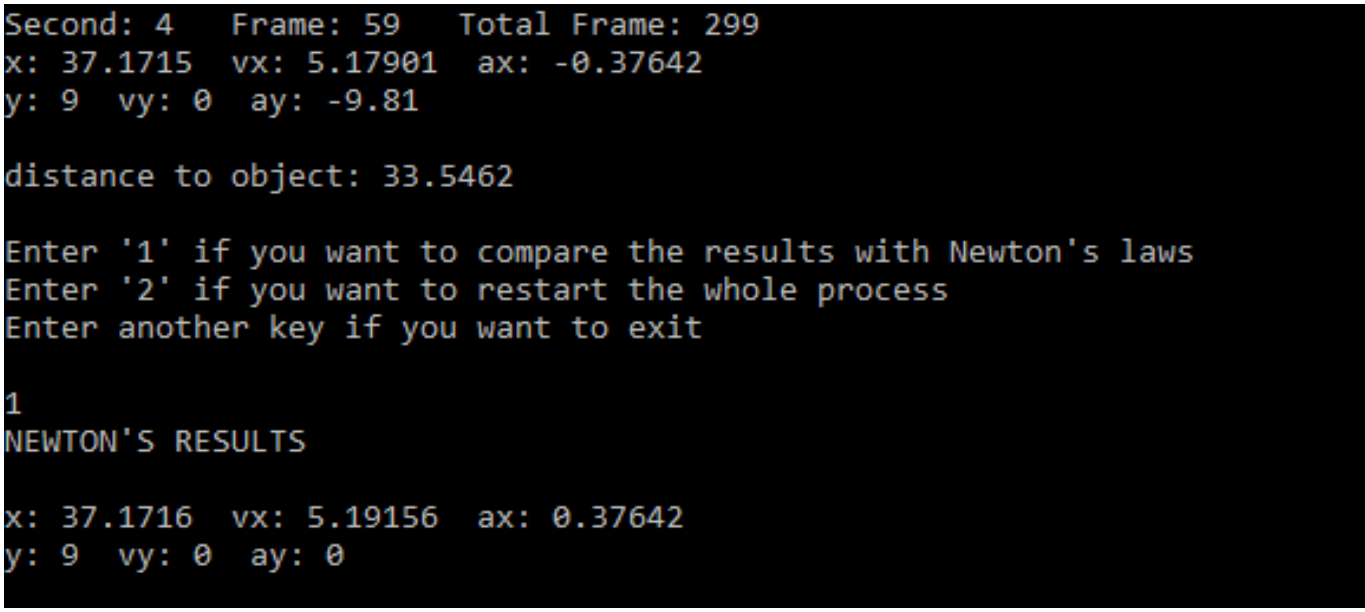




yo = 0, all the other data is random

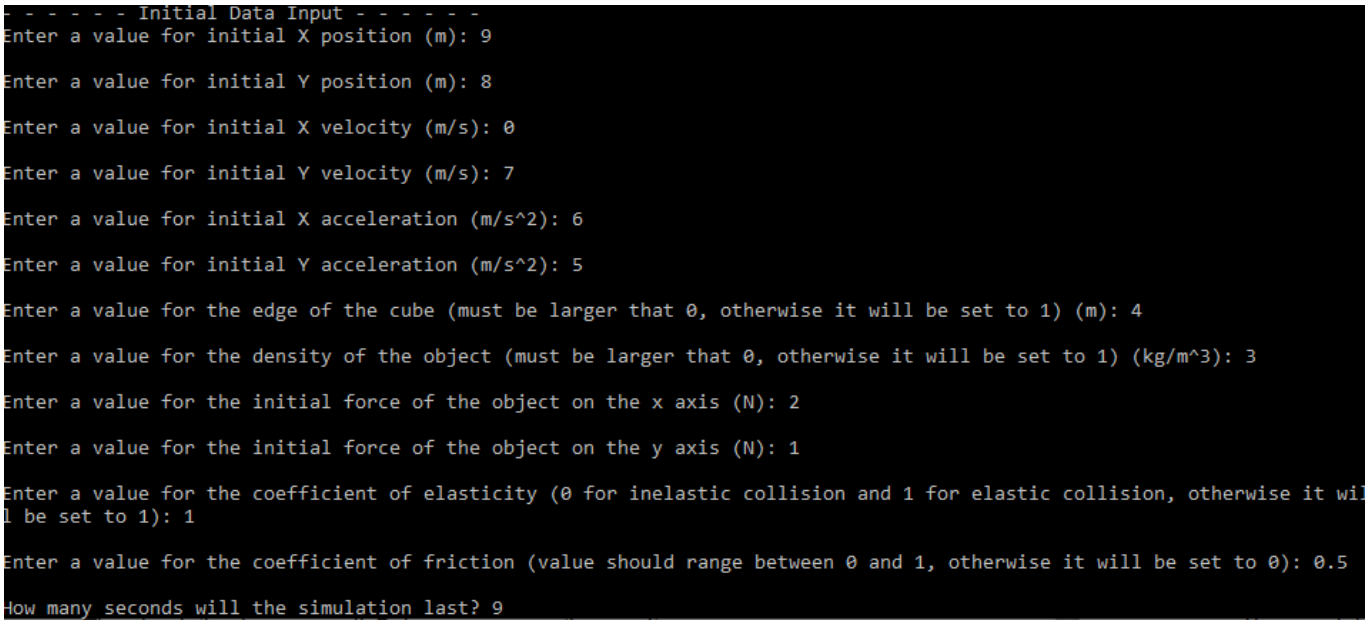
The simulation has run well. Below we can see the results:

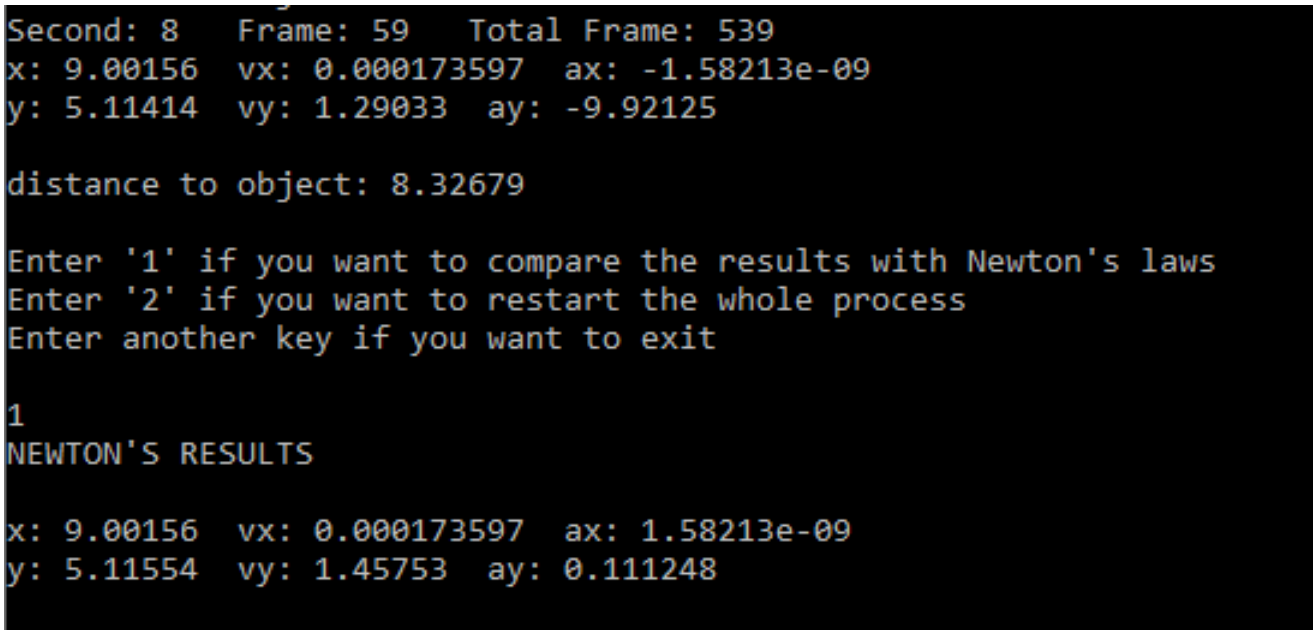




vxo = 0, all the other data is random

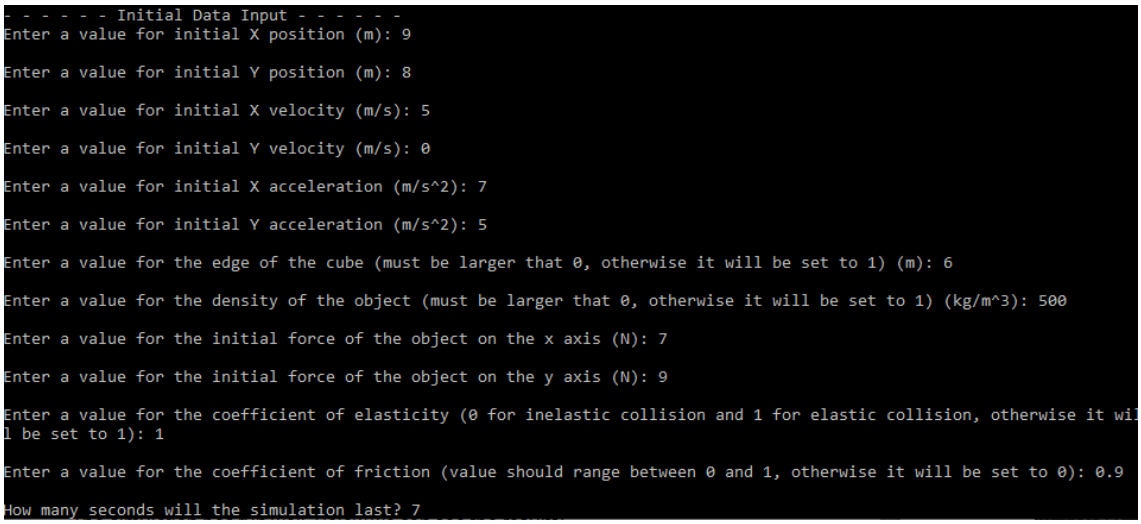
The simulation has run well. Below we can see the results:

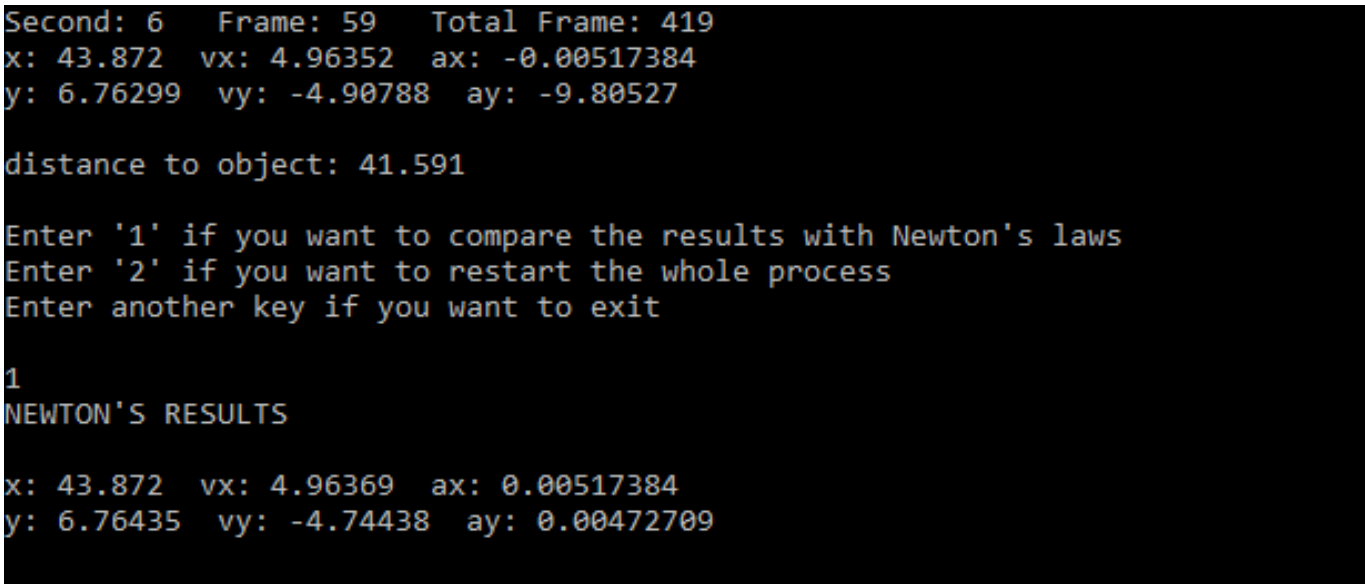




vyo = 0, all the other data is random

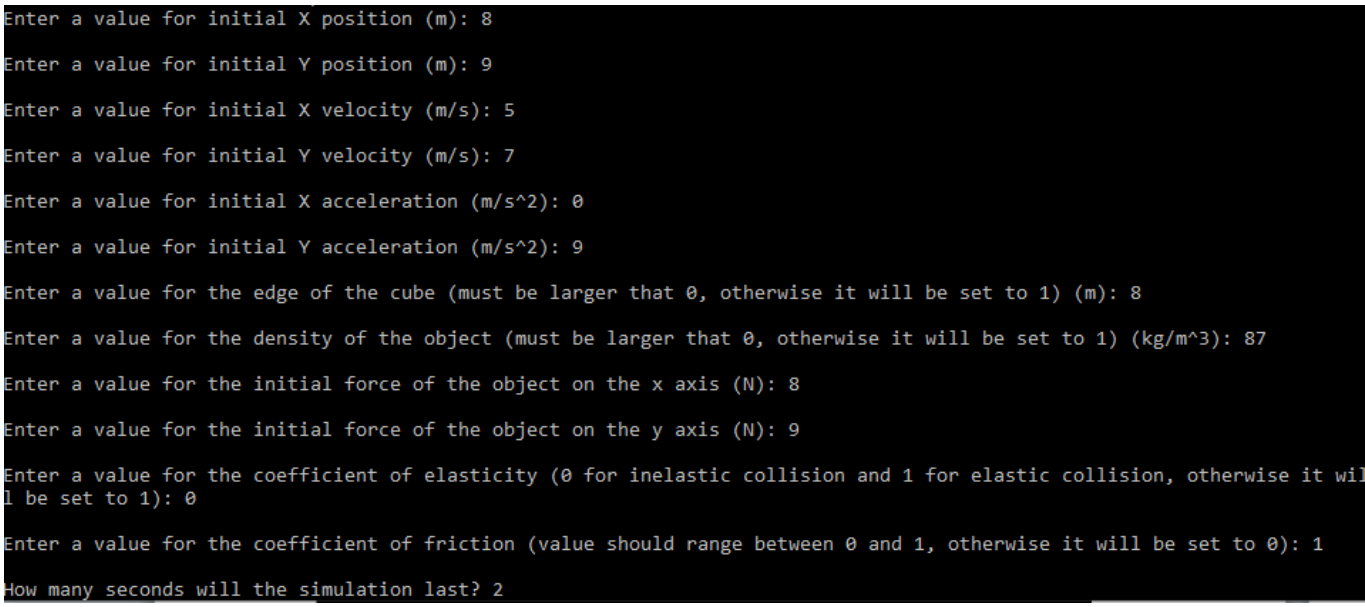
The simulation has run well. Below we can see the results:

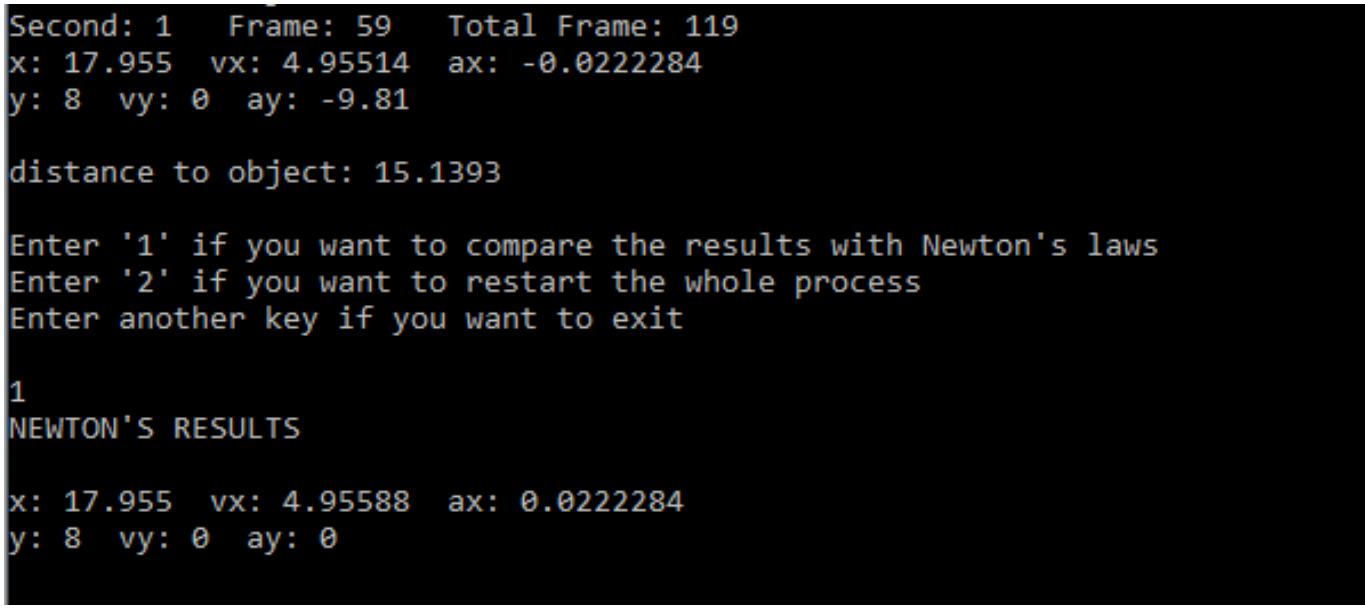




axo = 0, all the other data is random

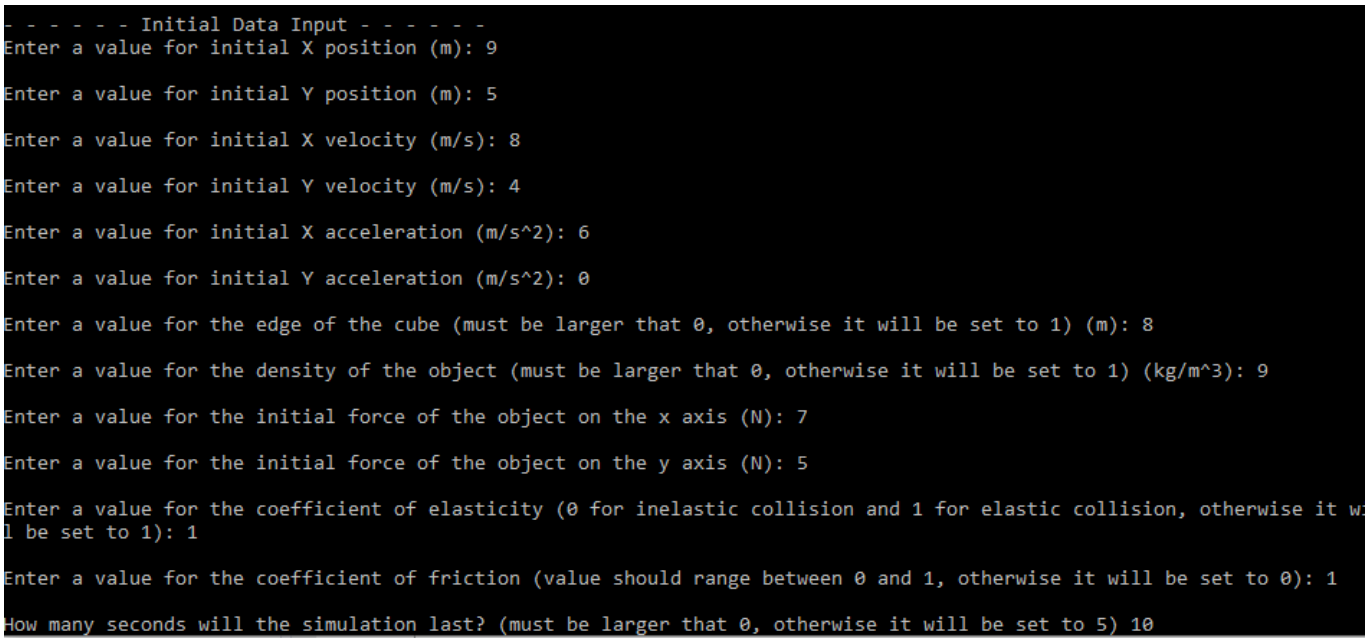
The simulation has run well. Below we can see the results:

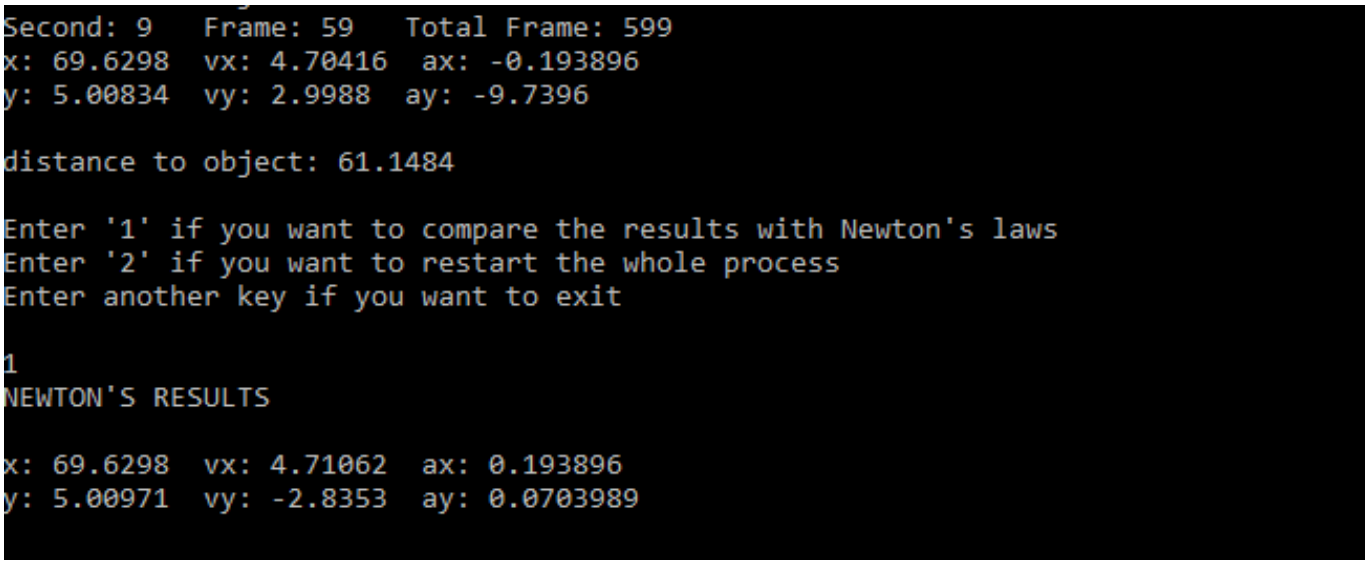




ayo = 0, all the other data is random

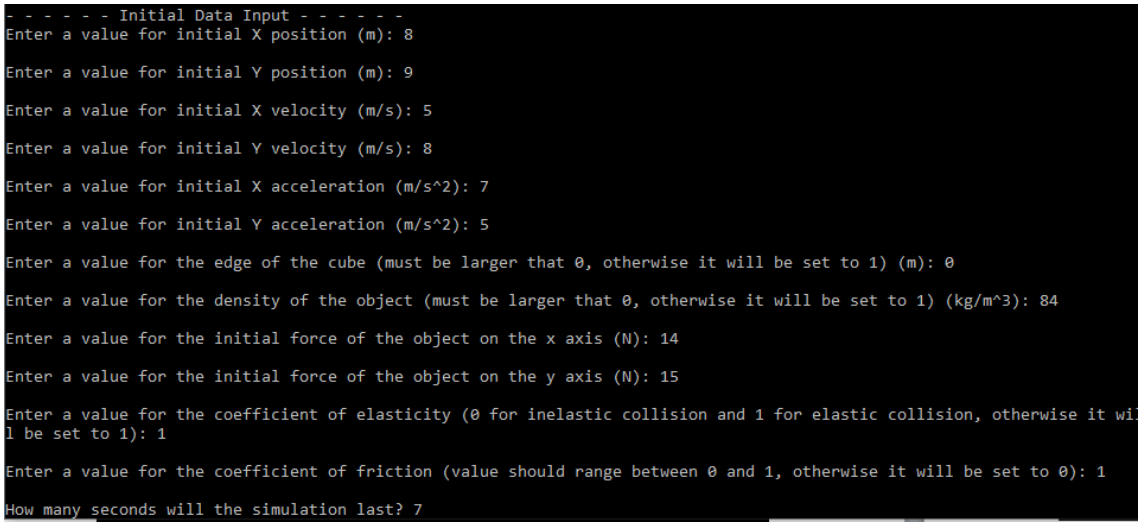
The simulation has run well. Below we can see the results:

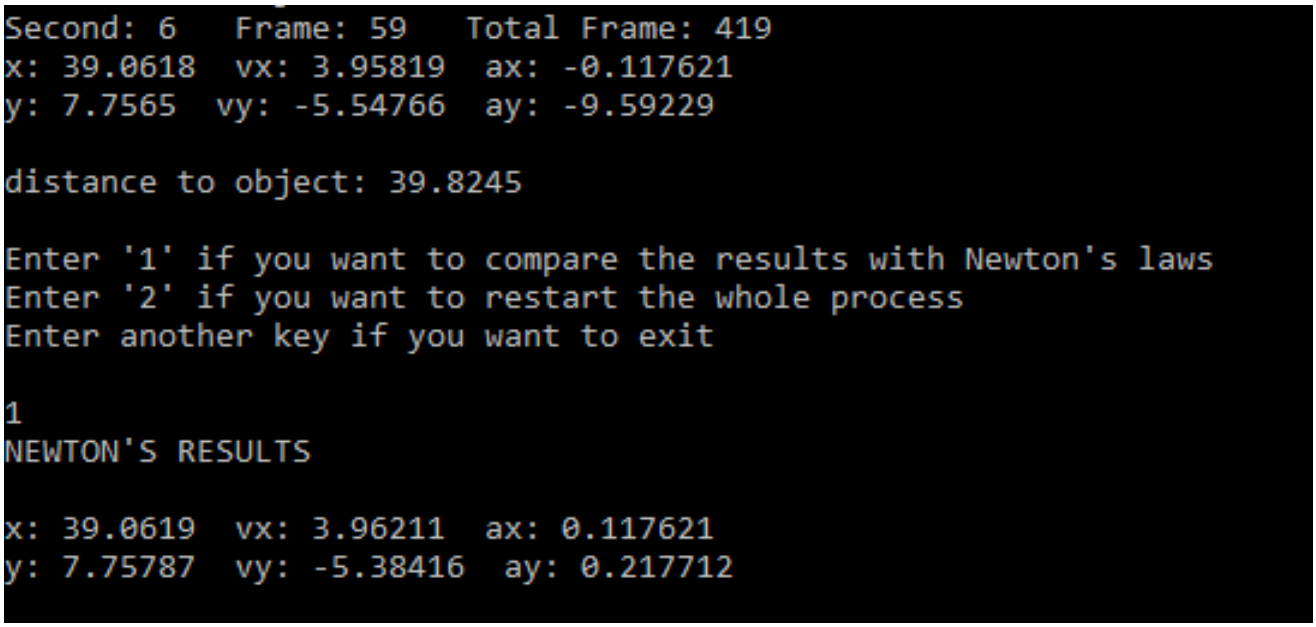




edge_lenght = 0, all the other data is random

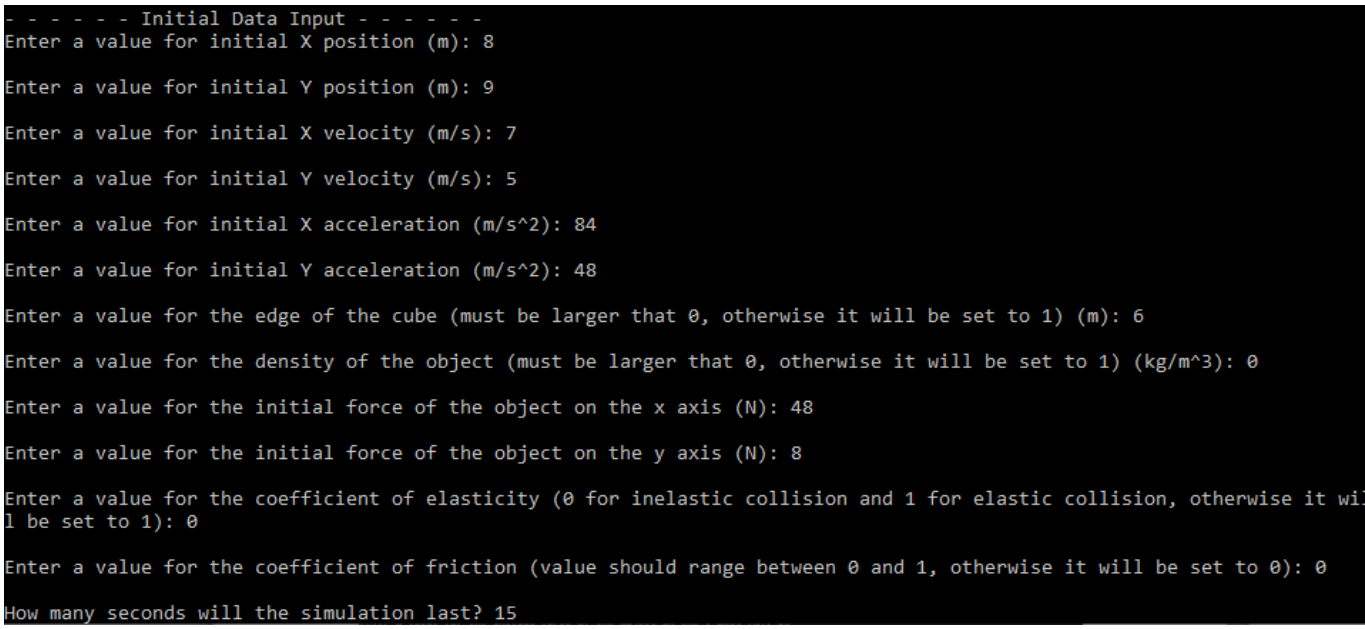
The simulation has run well. Below we can see the results:

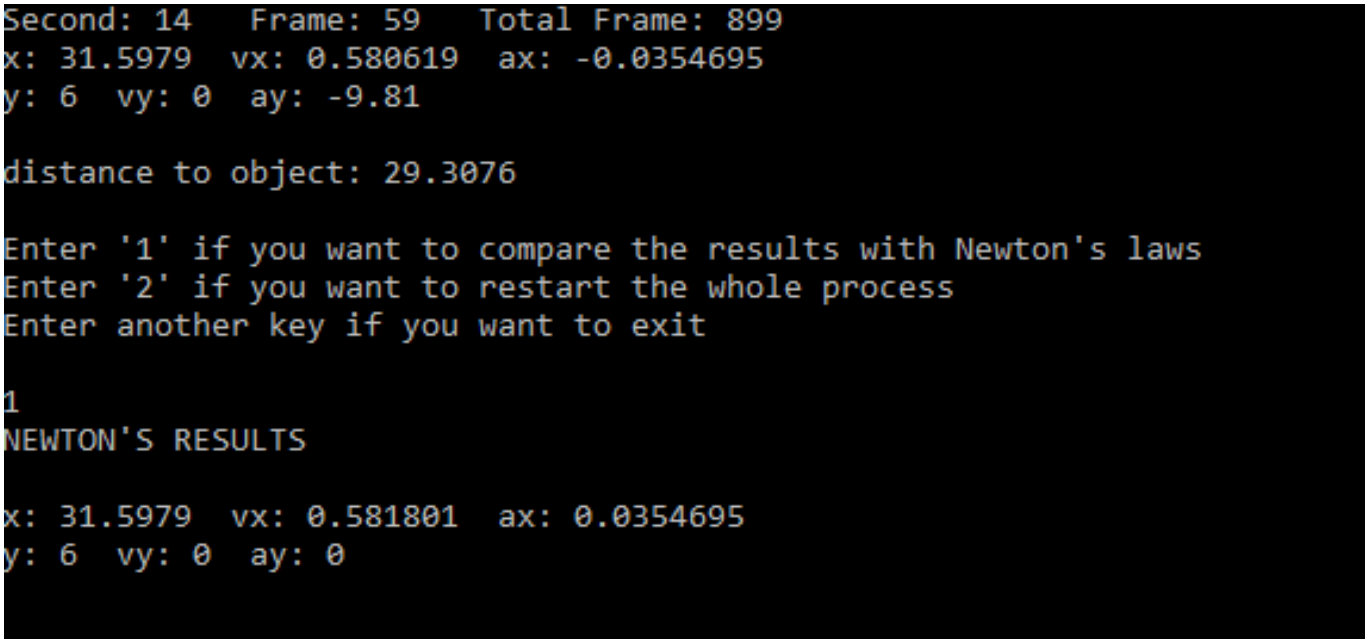




density = 0, all the other data is random

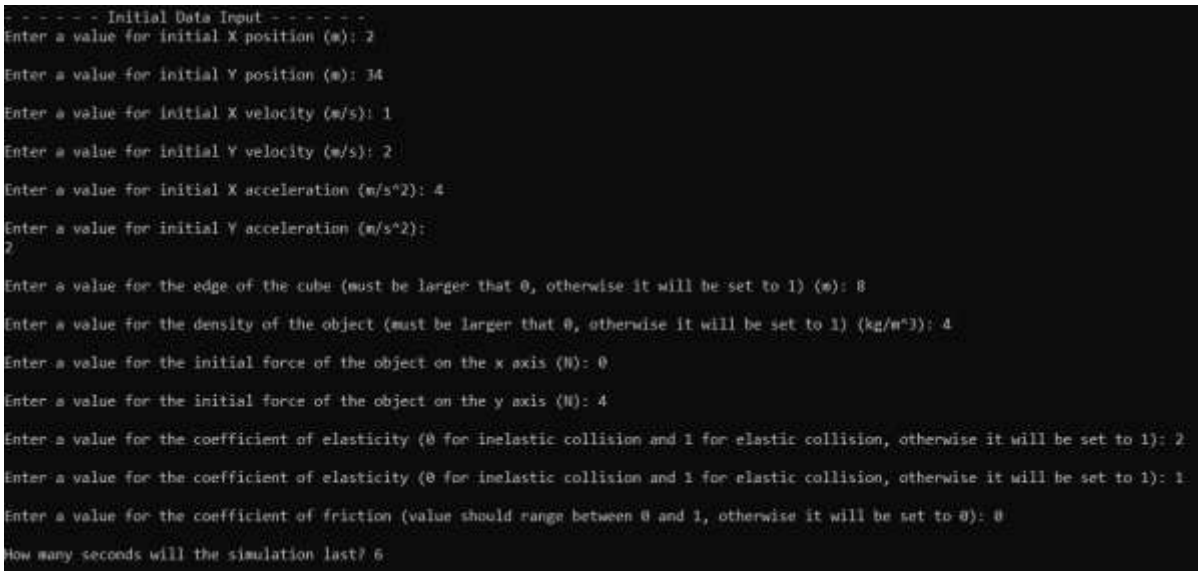
The simulation has run well. Below we can see the results:

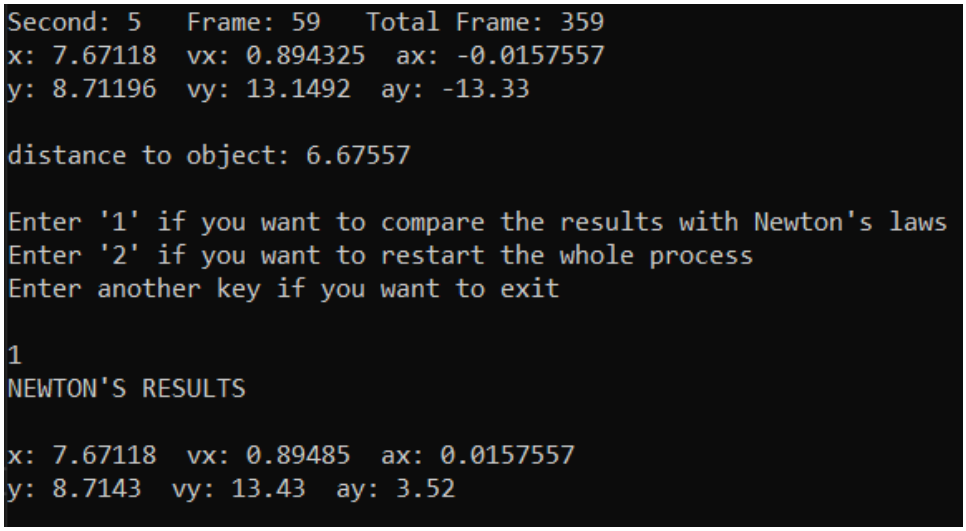




Fx = 0, all the other data is random

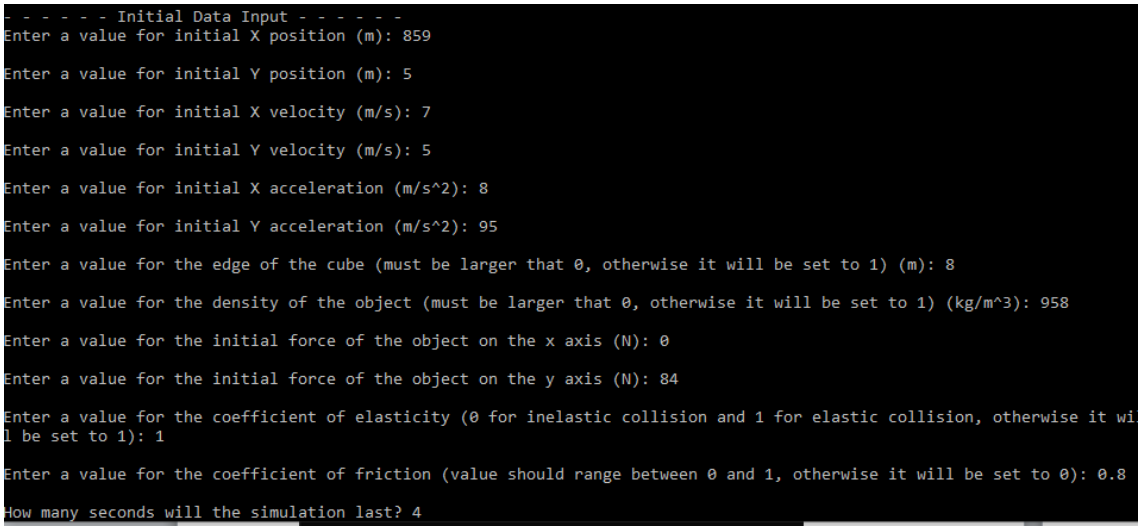
The simulation has run well. Below we can see the results:

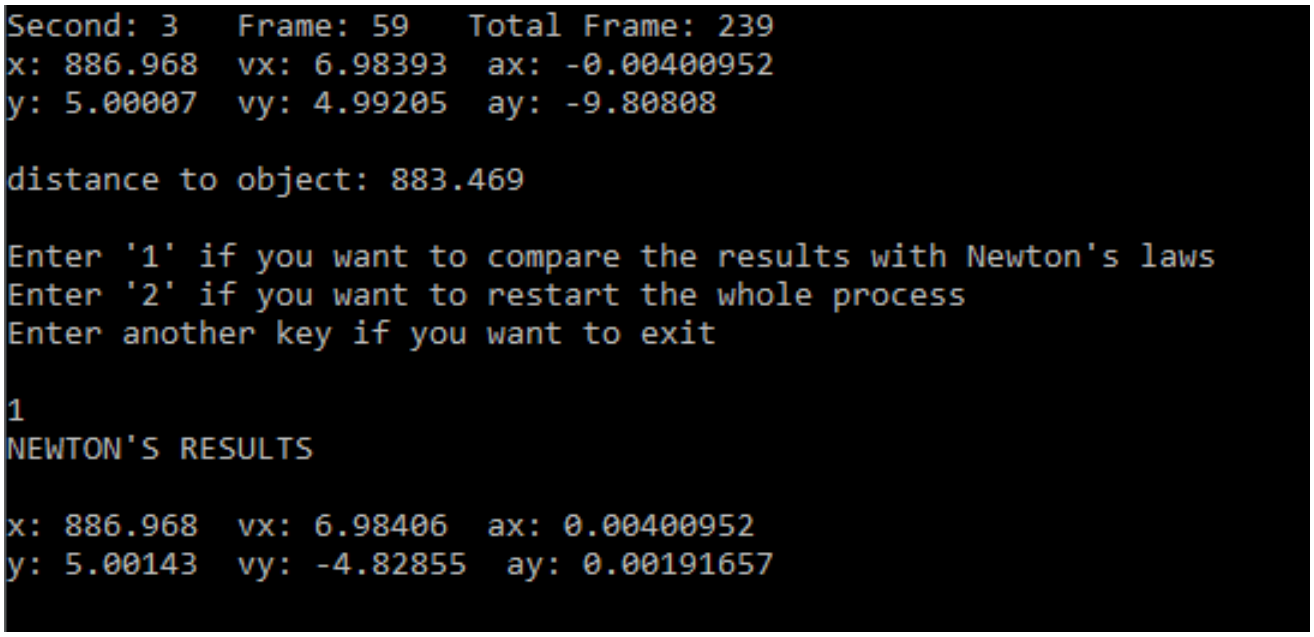




Fy = 0, all the other data is random

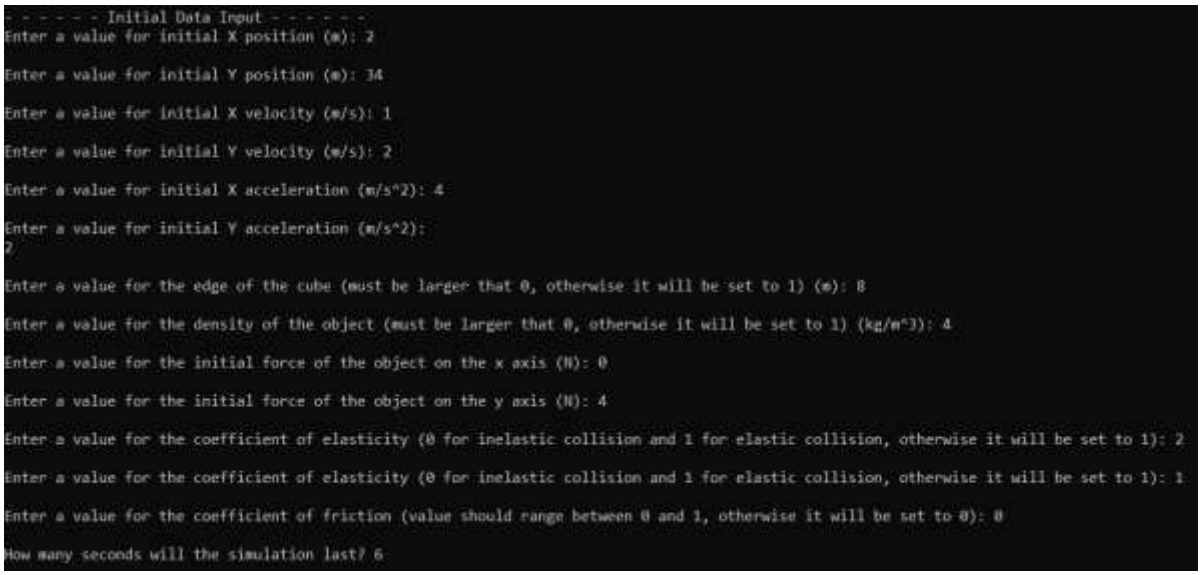
The simulation has run well. Below we can see the results:

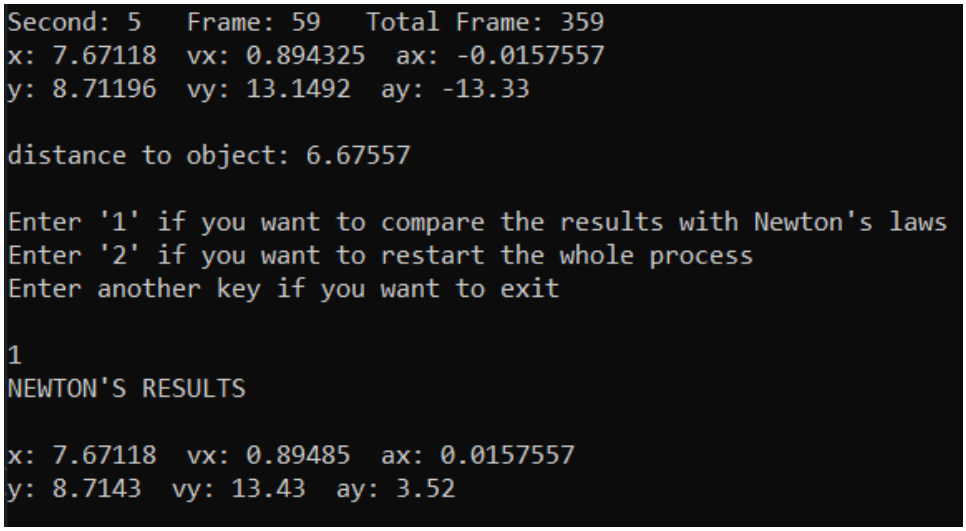




e = 0, all the other data is random

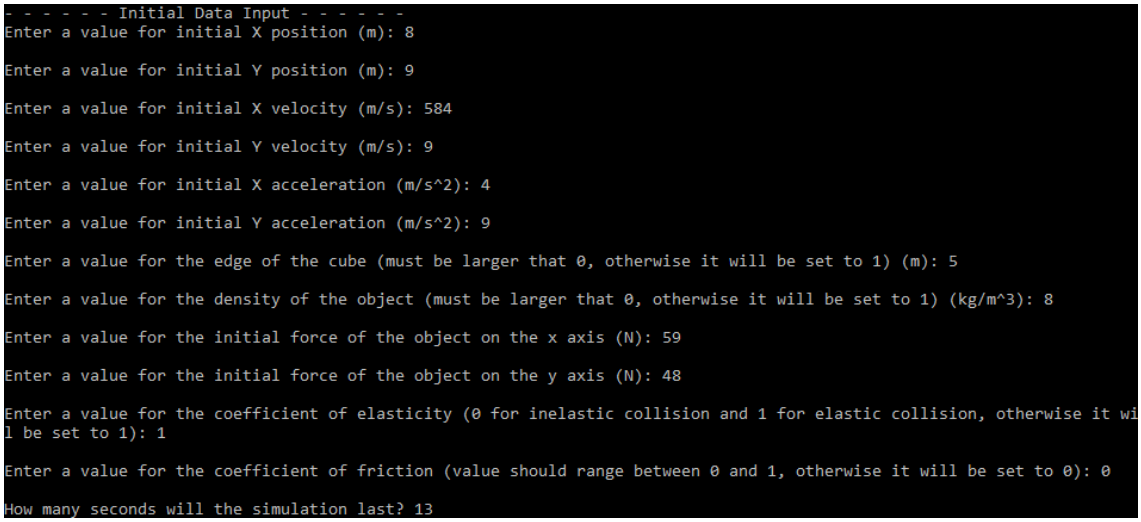
The simulation has run well. Below we can see the results:

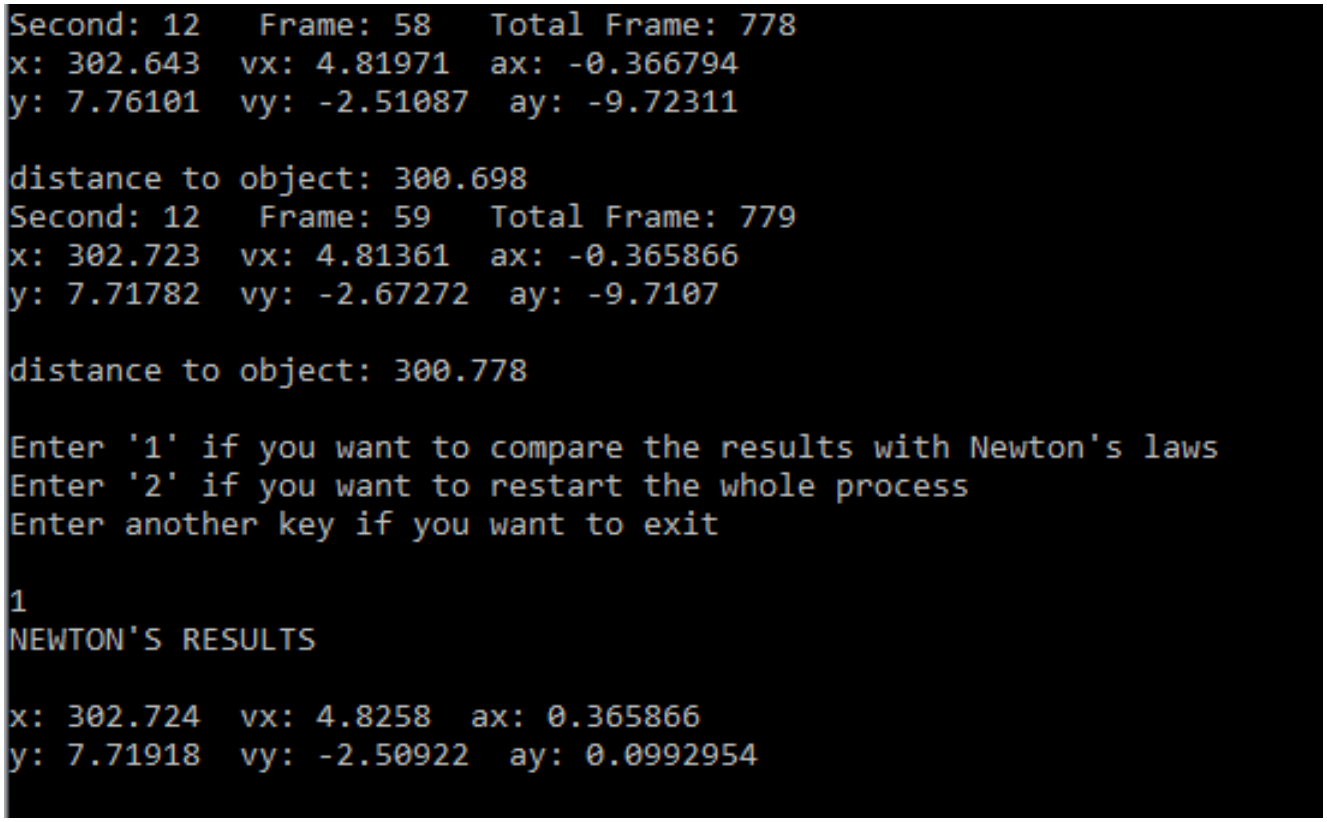




mu = 0, all the other data is random

The simulation has run well. Below we can see the results:





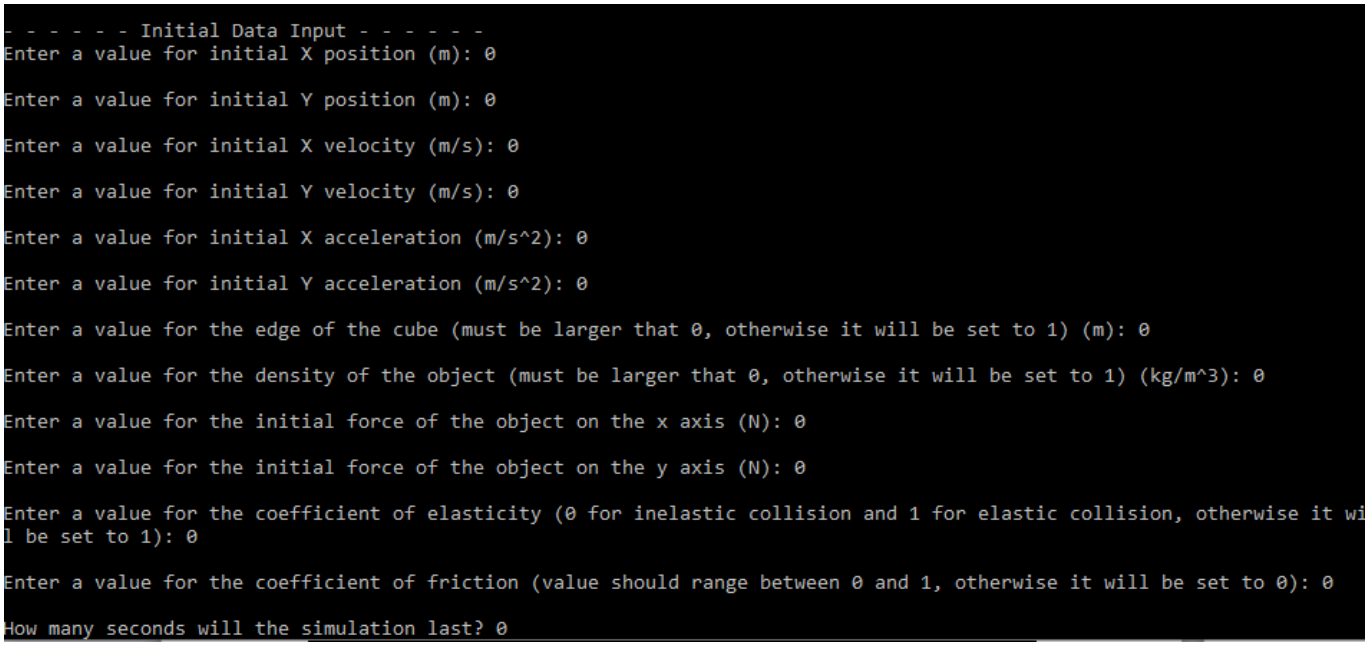
dt = 0, all the other data is random

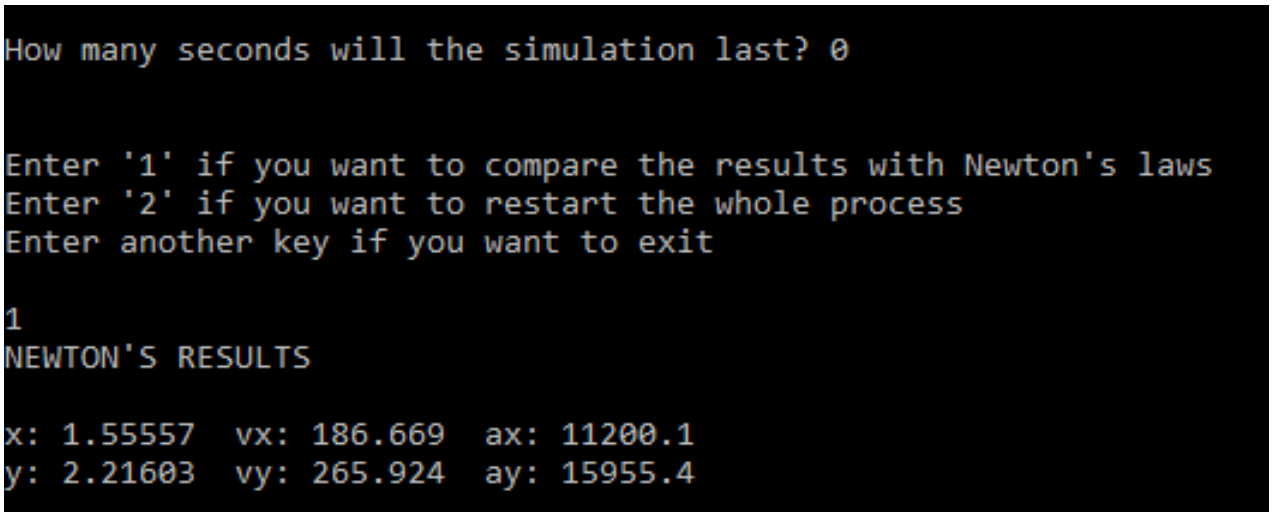
The simulation has run well. Below we can see the results:

```
- - - - - - Initial Data Input - - - - - -
Enter a value for initial X position (m): 9

Enter a value for initial Y position (m): 88

Enter a value for initial X velocity (m/s): 54

Enter a value for initial Y velocity (m/s): 85

Enter a value for initial X acceleration (m/s^2): 88

Enter a value for initial Y acceleration (m/s^2): 5

Enter a value for the edge of the cube (must be larger that 0, otherwise it will be set to 1) (m): 9

Enter a value for the density of the object (must be larger that 0, otherwise it will be set to 1) (kg/m^3): 54

Enter a value for the initial force of the object on the x axis (N): 8

Enter a value for the initial force of the object on the y axis (N): 54

Enter a value for the coefficient of elasticity (0 for inelastic collision and 1 for elastic collision, otherwise it wi
l be set to 1): 1

Enter a value for the coefficient of friction (value should range between 0 and 1, otherwise it will be set to 0): 85

Wrong value introduced.

Enter a value for the coefficient of friction (value should range between 0 and 1, otherwise it will be set to 0): 0

How many seconds will the simulation last? 0
```

```
Second: 4   Frame: 58   Total Frame: 298
x: 239.809  vx: 40.0242  ax: -2.08019
y: 328.887  vy: 18.6567  ay: -10.2695

distance to object: 398.528
Second: 4   Frame: 59   Total Frame: 299
x: 240.476  vx: 39.9896  ax: -2.07659
y: 329.197  vy: 18.4857  ay: -10.2612

distance to object: 399.167

Enter '1' if you want to compare the results with Newton's laws
Enter '2' if you want to restart the whole process
Enter another key if you want to exit

1
NEWTON'S RESULTS

x: 240.477  vx: 40.0588  ax: 2.07659
y: 329.198  vy: 18.6643  ay: 0.451207
```

(3) and (4) As stated before, the final data is correct and matches the results calculated with Newton's Laws.





```
- - - - - - Initial Data Input - - - - - -
Enter a value for initial X position (m): 89

Enter a value for initial Y position (m): 84

Enter a value for initial X velocity (m/s): 85

Enter a value for initial Y velocity (m/s): 84

Enter a value for initial X acceleration (m/s^2): 8

Enter a value for initial Y acceleration (m/s^2): 859

Enter a value for the edge of the cube (must be larger that 0, otherwise it will be set to 1) (m): 4

Enter a value for the density of the object (must be larger that 0, otherwise it will be set to 1) (kg/m^3): 95

Enter a value for the initial force of the object on the x axis (N): 84

Enter a value for the initial force of the object on the y axis (N): 8

Enter a value for the coefficient of elasticity (0 for inelastic collision and 1 for elastic collision, otherwise it wi
l be set to 1): 1

Enter a value for the coefficient of friction (value should range between 0 and 1, otherwise it will be set to 0): 1

How many seconds will the simulation last? 11
```

```
Second: 10   Frame: 59   Total Frame: 659
x: 653.316  vx: 33.3028  ax: -1.84213
y: 225.939  vy: -39.9052  ay: -7.18575

distance to object: 689.375

Enter '1' if you want to compare the results with Newton's laws
Enter '2' if you want to restart the whole process
Enter another key if you want to exit

1
NEWTON'S RESULTS

x: 653.317  vx: 33.3642  ax: 1.84213
y: 225.941  vy: -39.7417  ay: 2.62425
```

(5) The standard error deviation is as follows. We have used https://www.easycalculation.com/es/statistics/standard-error-calculator.php to calculate it. The deviation in each of the variables has been between 0,02 and 0,0005, which indicates that the integrator works inside the expected parameters, and that the offset is minimal. Therefore, after all the tests, we can assure that the code of the integrator is trustworthy.

Comparison with other integrators and Newton's Laws

As a result of the Newton comparison we've determined that Integrators, especially the "Verlet" method, are way more functional in this kind of exercises, since every operation has a very small imprecision and that they are more useful than calculating frame per frame with Newton's Laws.

Moreover, if we look at the "Euler" method (consists on applying velocity and gravity on every frame), we can see that as we use larger units or reduce the frame rate the imprecision is worse and builds up on every frame calculation.

In the following links you will be able to find the information on this topic that we used to come to a conclusion:

https://youtu.be/3HjO_RGIjCU?t=129

https://www.gamedev.net/forums/topic/422388-verlet-or-euler-integration/

Conclusions

Our Verlet Integrator has shown great results and works perfectly in the console. The tests show that the results calculating the forces with Newton's Laws and with the integrator differ almost nothing. In addition, the simulation works smoothly and the frame rate is controlled. Therefore, we encourage everyone to try our integrator since we have proven that the results are trustworthy.

by:
Tomás Carreras
Enric-G. Durán
Marc Garcia
Núria Lamonja (Team Leader)
Alex Lopez
Raul Morente
Albert Robles

GitHub Repository:
https://github.com/Needlesslord/Physics2theory

Thanks to:
David de la Torre

Project done for:
CITM-TTC