

Assignment 1 ACCB017

by Jacob Evans

Submission date: 29-Nov-2024 10:30PM (UTC+0000)

Submission ID: 244172391

File name: 50439_Jacob_Evans_Assignment_1_ACCB017_628791_703873890.docx (1.12M)

Word count: 2525

Character count: 13453



ACCB017 - GAMES ARCHITECTURE AND MECHANICS

Assignment 1

We are demonstrating the importance of numerical
integration within game architecture and game engines.

Jacob Evans
2311160@student.uwtsd.ac.uk

Contents

Part 1	2
Euler's Method	2
Runge-Kutta Method	3
Heun's Method	4
Part 2	5
The Simulation Code	5
Results and Comparison	8
Conclusion	11
Appendices	11
References	12

Part 1

Numerical integration, in mathematics, is the study of finding an “approximate value”. Despite these methods providing only an approximate value, certain methods can be so accurate that the value is as close as the value can get [1]. An example of such a method is the “Trapezoid Rule”, which is used to calculate an approximate area under a curved graph.

These numerical integration methods are incredibly useful and important when creating a realistic game engine. Using numerical integration methods such as Euler (explicit, implicit or semi-implicit), Verlet and Runge-Kutta can create smooth fluid dynamics and particle systems (sand, water, smoke). This allows developers/programmers to implement these methods to see how each particle reacts and give an approximate of how the particle will continue to react (predict). If the physics within the game works differently to how we would expect it to work (i.e water not taking shape of its container), then immersion is broken. Deciding which method to use on aspects of a game (particle-based, grid based) can reduce the instability and computer resource cost.

Euler’s Method

Euler’s method is described as a “step-by-step” approach to numerical integration. The idea behind Euler’s is to “join” smaller segments together to form an approximate value of the whole segment [2]. This method is also known as the “Marching Method”. Euler’s calculates the next value by using the rate of change to move forward in small steps. This rate of change is implemented in Part 2 as “deltaTime”, which is a time interval of 0.01s. It uses the previous value to then find the newest value.

This method is used primarily in first derivative equations such as “Velocity = Acceleration x Time”, which is one of the methods used in part 2. It’s a simple and efficient use of numerical integration, as it is easy to understand and requires minimal resources. This makes the formula simple to use and read while also reducing strain on the engine. In turn this reduces execution time, making it more useful on simple physics engines.

This method also has its flaws/disadvantages. Due to the method primarily using first derivative, the approximation is less accurate. The acceleration in part 2 is a constant, therefore the approximation is quite accurate, but this changes when acceleration is no longer a constant. The smaller rate of change also affects the accuracy of Euler’s method. Having larger intervals causes the method to become less accurate while increasing intervals (deltaTime in part 2) can cause the data to become more accurate, at the cost of slowing down the engine/simulation.



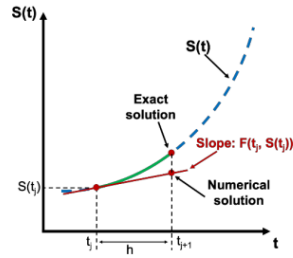


FIGURE 3 FROM RESOURCE [3]

2

From figure 3 above, we can visually see the inaccuracy that Euler's can create. Due to how quickly Euler's approximates data, these errors or inaccuracies also increase quickly and more drastically.

An example of Euler's method being fine to use is when a game character jumps while sprinting. If the objective is to keep the characters momentum/velocity the same when in the air and landing, then Euler's would be fine to use as those constants remain constant.

Navier-Stokes Equations

The Navier-Stokes equations are extensions of Euler's method. These equations describe the pressure, temperature and density of moving fluid [4]. These equations are the study called Computational Fluid Dynamics or CFD. Not many 3D games implement this. A prime example of a 2D game using CFD is Noita, released in 2020.

3

Runge-Kutta Method

$$\begin{aligned}
 y_{n+1} &= y_n + \frac{1}{6} (K_1 + 2K_2 + 2K_3 + K_4) \\
 K_1 &= hf(x_n, y_n) \\
 K_2 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_1\right) \\
 K_3 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_2\right) \\
 K_4 &= hf(x_n + h, y_n + K_3)
 \end{aligned}$$

FIGURE 4 [5]

4

Runge-Kutta (also known as RK4) has the formula above. It uses a fourth-order integral by taking the values of the initial slope of the graph (K1), two values from the centre of the graph (K2, K3) and the end of the graph. The values at the centre are weighted (x2), and then all added together to then by divided by 6. This creates an average at the graph, which calculates more accurately than Euler's when intervals are larger.

The advantage of using RK4 seems to oppose Euler's method, where using RK4 ignores the need for smaller intervals to produce more accurate values. RK4 reduces the number of calculations needed due to not relying on more intervals.

The accuracy provided by using RK4 is far more advantageous when used in real physics simulations/games compared to simple simulations [6]. Games like Kerbal Space Program and Microsoft Flight Simulator most likely take advantage of RK4 to create an immersive experience. This can be a cost to the game engine though, as the calculations can take longer, which can cause performance and visual issues. That's why it's recommended to use Euler's or Heun's when making simpler calculations.

Heun's Method

Heun's method is another example of numerical integration. Heun's method is a modified version of Euler's method by using second-order integration. This was found/created when Runge-Kutta method was being developed. This method was created to attempt to decrease the inaccuracies that Euler creates [7]. This would have the same advantages as Euler, while reducing the disadvantages when using Euler's method.

$$y_{n+1} = y_n + \frac{h}{2} [f(x_n, y_n) + f(x_n + h, y_n + hf(x_n, y_n))], \quad n = 0, 1, 2,$$

FIGURE 8 - HEUN METHOD FORMULA

Part 2

The Simulation Code

In part two, we are asked to create simulation in C++ of a spring-powered car being released and have the code calculate the velocity, acceleration (or deceleration) and position of the car at different time intervals. We can use different numerical integration methods to calculate these factors.

```
#include <iostream>
#include <cmath>
#include "FileWriter.h"

int main()
{
    //x is the max distance of spring being pulled back
    float x = 0.0;

    //Setting constants for Fd equation
    float mass = 1;
    float area = 0.01;
    float density = 1.2;
    float drag = 0.34;
    float grav = 9.81;
    float res = 0.21;
    float acceleration = 0;
    float velocity = 0.0;

    FileWriter fw("SpringSimul.csv"); // creates file of simulation results

    float deltaTime = 0.01;
    float fd1 = res * mass * grav; // first part of FD equation
    float fd2 = 0.5 * drag * density * area * (velocity * velocity); //second part of FD equation
    float time = 0;
```

We first include the libraries “iostream”, “cmath” and the FileWriter.h file.

The screenshot above shows that we first must set the constants provided by the assignment. These constants allow us to calculate the initial force of the compressed spring using “Hooke’s Law”, which is the “ $FT = 1.2 (16-x)$ ” provided in the assignment, with “x” being our distance. The FileWriter function creates a csv file named “SpringSimul” which is where the results are recorded.

We set our initial acceleration and velocity to be 0, as the car has not started moving at 0 seconds. The constant “deltaTime” is the time intervals of 0.01 as stated in the assignment.

$$Fd = C_R mg + \frac{1}{2} C_D \rho A V^2$$

FIGURE 1 - FD FORMULA FROM ASSIGNMENT

The constant “fd1” is the first part of the equation above (figure 1), being the Coefficient of Drag multiplied by mass multiplied by gravity. The constant “fd2” is the remainder of the equation. This will be useful later when we want to calculate the acceleration later in the simulation.

```

std::cout << "Time: " << time << "s position: " << x << "m, Velocity: " << velocity << "m/s, Acceleration: " << acceleration << "m/s^2" << std::endl; //prints the initial results
fw.write(time, x, velocity, acceleration); //this records the results to the csv file "SpringSim"

while (velocity > 0 || time == 0)
{
    for (time = 0; time <= deltaTime;)
    {
        float Fk = 1.2 * (10 - x);
        if (x > 6)
        {
            Fk = 0;
        }
        float Fd = Fk + fd2;
        acceleration = (Fk - Fd) / Mass;
        velocity = velocity + acceleration * deltaTime; //Eulers METHOD
        x = x + velocity * deltaTime;
    }
}

```

The first line of the screenshot above prints out the initial time, position, velocity and acceleration of the car (these being 0), and “fw.write” records this information to the Excel file.

We then start providing the setting the bounds for the calculations. We set a while loop for as long as the velocity is greater than 0 or the time is equal to 0. This will allow the simulation to start. When the velocity becomes less than 0, the simulation should escape the loop, which would indicate the car has stopped moving.

We then set a for loop which will increase the time intervals by 0.01. This will provide us the time intervals of 0.01. Within this loop, it calculates our first equation, being Hooke’s Law. This will calculate the moving (tractive) force of the spring on the car (FT). Calculating FT is fundamental for calculating the acceleration, and then the velocity of the vehicle.

The if loop applies the condition that if the distance travelled by the car is greater than 6 metres, then $F_T = 0$. This means that the car is no longer accelerating and should start decelerating until velocity becomes 0. We then calculate the drag force on the car by adding the constants “fd1” and “fd2”, creating FD.

The calculation for acceleration is $(F_T - F_D) / \text{mass}$. This is a numerical integration, as this is the “change in force” divided by mass. This is a rearranged version of Newtons 2nd Law, $F=MA$. This will allow us to then calculate both the velocity and the position of the car.

The use Eulers Method of numerical integration is to calculate the velocity and position of the car. Using “velocity = velocity + acceleration * deltaTime”, this will update the velocity of the car for every time interval of 0.01 seconds. We can then use the new calculated velocity to calculate the position of the car using “x = x + velocity * deltaTime” (x being the distance travelled by the car).

```

if (velocity <= 0)
{
    fd = 0;
    velocity = 0;
    break;
}

std::cout << "Time: " << time << "s position: " << x << "m, Velocity: " << velocity << "m/s, Acceleration: " << acceleration << "m/s^2" << std::endl;
fw.write(time, x, velocity, acceleration);
}

std::cout << "Time: " << time << "s position: " << x << "m, Velocity: " << velocity << "m/s, Acceleration: " << acceleration << "m/s^2" << std::endl;
fw.write(time, x, velocity, acceleration);

```

The screenshot above applies another if statement, that if the velocity greater than or equal to 0, then fd (drag) becomes 0, and so does the velocity.

Within the for loop, the results are printed to the console, and then written to the Excel file. Outside the loop, the same is done to print out the position, velocity, time and acceleration of the car when the car has stopped.

[illegible]

These are the results using Eulers Method on the console.

The other numerical integration method used was the Runge-Kutta method, also known as RK4. The RK4 method is inputted in the same place that the Euler's method would be implemented in the simulation.

```
float velocityk1 = acceleration * deltaTime;
float xk1 = velocity * deltaTime;

float velocityk2 = acceleration * deltaTime;
float xk2 = (velocity + velocityk1 * 0.5f) * deltaTime;

float velocityk3 = acceleration * deltaTime;
float xk3 = (velocity + velocityk2 * 0.5f) * deltaTime;

float velocityk4 = acceleration * deltaTime;
float xk4 = (velocity + velocityk3) * deltaTime;

velocity += (velocityk1 + 2.0f * velocityk2 + 2.0f * velocityk3 + velocityk4) / 6.0f;
x += (xk1 + 2.0f * xk2 + 2.0f * xk3 + xk4) / 6.0f;
```

The screenshot above shows how RK4 is implemented, where we are calculating an initial velocity and an initial position. Rather than using the previous velocity and position, RK4 calculates the next three velocity and position data, and then calculates the velocity using the equation of “velocityK1 + 2.0f * velocityK2 + 2.0f * velocityK3 + velocityK4” all divided by 6 (creates an average velocity).

The same is done when calculating the position “x”.

This creates a more accurate estimate of the values of velocity and positioning, as discussed in part 1.

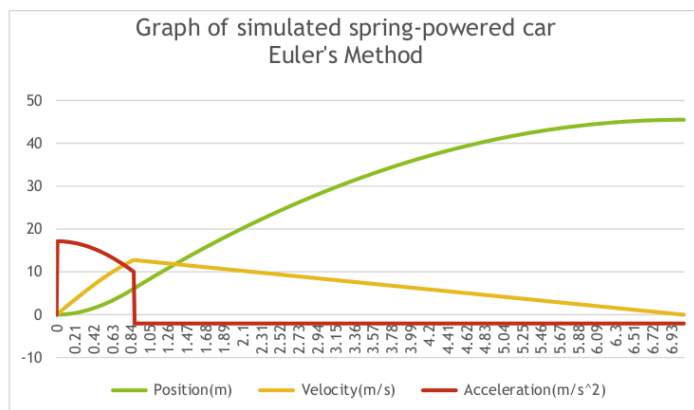
```

Microsoft Visual Studio Debu x + v
Time: 6.84006s position: 45.6524m, Velocity: 0.489138m/s, Acceleration: -2.0601m/s^2
Time: 6.85006s position: 45.6572m, Velocity: 0.468537m/s, Acceleration: -2.0601m/s^2
Time: 6.86006s position: 45.6618m, Velocity: 0.447936m/s, Acceleration: -2.0601m/s^2
Time: 6.87006s position: 45.6662m, Velocity: 0.427335m/s, Acceleration: -2.0601m/s^2
Time: 6.88006s position: 45.6703m, Velocity: 0.406734m/s, Acceleration: -2.0601m/s^2
Time: 6.89006s position: 45.6743m, Velocity: 0.386133m/s, Acceleration: -2.0601m/s^2
Time: 6.90006s position: 45.6781m, Velocity: 0.365532m/s, Acceleration: -2.0601m/s^2
Time: 6.91006s position: 45.6816m, Velocity: 0.344931m/s, Acceleration: -2.0601m/s^2
Time: 6.92006s position: 45.685m, Velocity: 0.32433m/s, Acceleration: -2.0601m/s^2
Time: 6.93006s position: 45.6881m, Velocity: 0.303729m/s, Acceleration: -2.0601m/s^2
Time: 6.94006s position: 45.691m, Velocity: 0.283128m/s, Acceleration: -2.0601m/s^2
Time: 6.95006s position: 45.6938m, Velocity: 0.262527m/s, Acceleration: -2.0601m/s^2
Time: 6.96006s position: 45.6963m, Velocity: 0.241926m/s, Acceleration: -2.0601m/s^2
Time: 6.97006s position: 45.6986m, Velocity: 0.221325m/s, Acceleration: -2.0601m/s^2
Time: 6.98006s position: 45.7007m, Velocity: 0.200724m/s, Acceleration: -2.0601m/s^2
Time: 6.99007s position: 45.7026m, Velocity: 0.180123m/s, Acceleration: -2.0601m/s^2
Time: 7.00007s position: 45.7043m, Velocity: 0.159522m/s, Acceleration: -2.0601m/s^2
Time: 7.01007s position: 45.7058m, Velocity: 0.138921m/s, Acceleration: -2.0601m/s^2
Time: 7.02007s position: 45.7071m, Velocity: 0.11832m/s, Acceleration: -2.0601m/s^2
Time: 7.03007s position: 45.7082m, Velocity: 0.0977188m/s, Acceleration: -2.0601m/s^2
Time: 7.04007s position: 45.709m, Velocity: 0.0771178m/s, Acceleration: -2.0601m/s^2
Time: 7.05007s position: 45.7097m, Velocity: 0.0565168m/s, Acceleration: -2.0601m/s^2
Time: 7.06007s position: 45.7102m, Velocity: 0.0359158m/s, Acceleration: -2.0601m/s^2
Time: 7.07007s position: 45.7104m, Velocity: 0.0153148m/s, Acceleration: -2.0601m/s^2
Time: 7.08007s position: 45.7105m, Velocity: 0m/s, Acceleration: -2.0601m/s^2
Simulation is completed. Results were save to: SpringSimul.csv

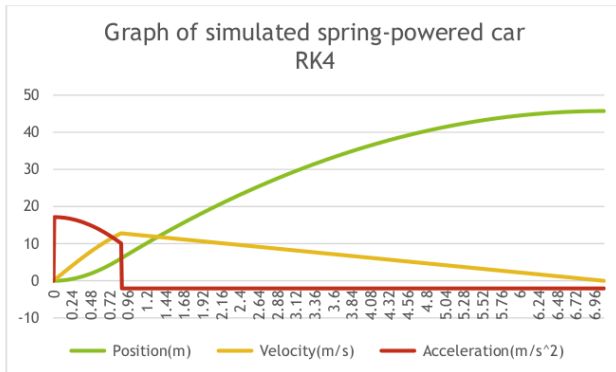
```

These are the results provided from RK4.

Results and Comparison



This is the graph created by the data recorded in the Excel sheet. The data on the console and the graph show that the car has travelled 45.49 metres in 7.07 seconds before coming to a stop. The graph shows that the velocity starts to decrease after 6 metres, and the acceleration also drops to a negative value (indicating deceleration), which is correct as FT becomes 0 after the car has travelled 6 metres. The deceleration is a constant -2.0601m/s^2 as the equation of acceleration becomes $-FD$ divided by mass.



We can see the same graph shape when using the RK4 method, but the actual results are different. We can make the statement that the RK4 graph would look more detailed if there was more data to be had (more velocity, distance and time).

6.77	45.41126	0.60032	-2.0601
6.78	45.41706	0.57972	-2.0601
6.79	45.42265	0.55912	-2.0601
6.8	45.42804	0.53852	-2.0601
6.81	45.43322	0.51792	-2.0601
6.82	45.43819	0.49732	-2.0601
6.83	45.44296	0.47672	-2.0601
6.84	45.44752	0.45611	-2.0601
6.85	45.45188	0.43551	-2.0601
6.86	45.45603	0.41491	-2.0601
6.87	45.45997	0.39431	-2.0601
6.88	45.46371	0.37371	-2.0601
6.89	45.46724	0.35311	-2.0601
6.9	45.47057	0.33251	-2.0601
6.91	45.47369	0.31191	-2.0601
6.92	45.4766	0.29131	-2.0601
6.93	45.47931	0.27071	-2.0601
6.94	45.48182	0.2501	-2.0601
6.95	45.48411	0.2295	-2.0601
6.96	45.4862	0.2089	-2.0601
6.97	45.48809	0.1883	-2.0601
6.98	45.48977	0.1677	-2.0601
6.99	45.49124	0.1471	-2.0601
7	45.4925	0.1265	-2.0601
7.01	45.49356	0.1059	-2.0601
7.02	45.49442	0.0853	-2.0601
7.03	45.49507	0.0647	-2.0601
7.04	45.49551	0.04409	-2.0601
7.05	45.49575	0.02349	-2.0601
7.06	45.49578	0.00289	-2.0601
7.07	45.4956	0	-2.0601

FIGURE 5 - EULERS RESULTS

6.77	45.61315	0.63334	-2.0601
6.78	45.61938	0.61274	-2.0601
6.79	45.6254	0.59214	-2.0601
6.8	45.63122	0.57154	-2.0601
6.81	45.63683	0.55094	-2.0601
6.82	45.64224	0.53034	-2.0601
6.83	45.64744	0.50974	-2.0601
6.84	45.65243	0.48914	-2.0601
6.85	45.65722	0.46854	-2.0601
6.86	45.6618	0.44794	-2.0601
6.87	45.66618	0.42733	-2.0601
6.88	45.67035	0.40673	-2.0601
6.89	45.67431	0.38613	-2.0601
6.9	45.67807	0.36553	-2.0601
6.91	45.68162	0.34493	-2.0601
6.92	45.68496	0.32433	-2.0601
6.93	45.6881	0.30373	-2.0601
6.94	45.69104	0.28313	-2.0601
6.95	45.69376	0.26253	-2.0601
6.96	45.69629	0.24193	-2.0601
6.97	45.6986	0.22132	-2.0601
6.98	45.70071	0.20072	-2.0601
6.99	45.70261	0.18012	-2.0601
7	45.70431	0.15952	-2.0601
7.01	45.7058	0.13892	-2.0601
7.02	45.70709	0.11832	-2.0601
7.03	45.70817	0.09772	-2.0601
7.04	45.70904	0.07712	-2.0601
7.05	45.70971	0.05652	-2.0601
7.06	45.71017	0.03592	-2.0601
7.07	45.71043	0.01531	-2.0601
7.08	45.71048	0	-2.0601

FIGURE 6 - RK4 RESULTS

We can see many different in results between the two methods used. Figure 5's results' end position is lower than the end position of Figure 6's results', by approximately 0.20 metres. We can also see that the car's velocity becomes 0 at 7.07 when using Euler's, but 7.08 when using RK4. The

position and velocity are different at every time interval when comparing the two results. When we compared the two methods in Part 1, the conclusion was that RK4 is more accurate.

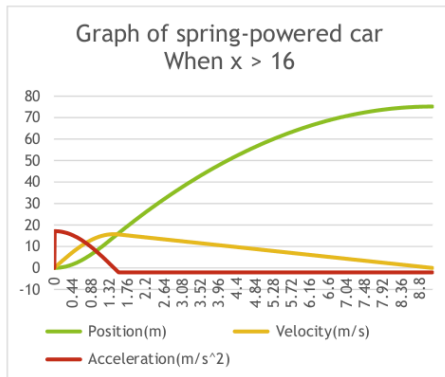
<pre>float mass = 5; float area = 5; float density = 1.2; float drag = 0.34; float grav = 9.81; float res = 0.21; float acceleration = 0; float velocity = 0.0;</pre>	<pre>Time: 4.38801s position: 9.09664m, Velocity: 0.368596m/s, Acceleration: -2.0601m/s^2 Time: 4.39801s position: 9.10914m, Velocity: 0.339995m/s, Acceleration: -2.0601m/s^2 Time: 4.40801s position: 9.12144m, Velocity: 0.319394m/s, Acceleration: -2.0601m/s^2 Time: 4.41801s position: 9.13653m, Velocity: 0.298793m/s, Acceleration: -2.0601m/s^2 Time: 4.42801s position: 9.1542m, Velocity: 0.278192m/s, Acceleration: -2.0601m/s^2 Time: 4.43801s position: 9.1721m, Velocity: 0.257591m/s, Acceleration: -2.0601m/s^2 Time: 4.44801s position: 9.19057m, Velocity: 0.23699m/s, Acceleration: -2.0601m/s^2 Time: 4.45801s position: 9.11694m, Velocity: 0.216389m/s, Acceleration: -2.0601m/s^2 Time: 4.46801s position: 9.1189m, Velocity: 0.195788m/s, Acceleration: -2.0601m/s^2 Time: 4.47801s position: 9.12075m, Velocity: 0.175187m/s, Acceleration: -2.0601m/s^2 Time: 4.48801s position: 9.1224m, Velocity: 0.154586m/s, Acceleration: -2.0601m/s^2 Time: 4.49801s position: 9.1238m, Velocity: 0.133985m/s, Acceleration: -2.0601m/s^2 Time: 4.50801s position: 9.12508m, Velocity: 0.113384m/s, Acceleration: -2.0601m/s^2 Time: 4.51801s position: 9.12611m, Velocity: 0.0927827m/s, Acceleration: -2.0601m/s^2 Time: 4.52801s position: 9.12694m, Velocity: 0.0721817m/s, Acceleration: -2.0601m/s^2 Time: 4.53801s position: 9.12755m, Velocity: 0.0515807m/s, Acceleration: -2.0601m/s^2 Time: 4.54801s position: 9.12797m, Velocity: 0.0309797m/s, Acceleration: -2.0601m/s^2 Time: 4.55801s position: 9.12817m, Velocity: 0.0103787m/s, Acceleration: -2.0601m/s^2 Time: 4.56801s position: 9.12818m, Velocity: 0m/s, Acceleration: -2.0601m/s^2</pre>
---	---

By changing the constants, we get different results. The mass has been increased by x5 and the frontal area of the vehicle to 5m² with the remaining constants kept the same. This caused the car to stop at only 9.13metres in 4.5 seconds. This is because the car is not accelerating as quickly due to the equation (FT-FD)/mass. The acceleration is directly affected by the mass of the car. The FD is directly affected due to the increase in the frontal area of the vehicle. This causes FD to be larger than FT sooner, causing deceleration to happen sooner in the simulation.

<pre>float mass = 5; float area = 5; float density = 0.020; float drag = 0.34; float grav = 3.73; float res = 0.21; float acceleration = 0; float velocity = 0.0;</pre>	<pre>Time: 8.6901s position: 23.9592m, Velocity: 0.118431m/s, Acceleration: -0.7833m/s^2 Time: 8.7001s position: 23.9644m, Velocity: 0.110596m/s, Acceleration: -0.7833m/s^2 Time: 8.7101s position: 23.9614m, Velocity: 0.102755m/s, Acceleration: -0.7833m/s^2 Time: 8.7201s position: 23.9624m, Velocity: 0.0949321m/s, Acceleration: -0.7833m/s^2 Time: 8.7301s position: 23.9633m, Velocity: 0.0870991m/s, Acceleration: -0.7833m/s^2 Time: 8.7401s position: 23.9642m, Velocity: 0.0792661m/s, Acceleration: -0.7833m/s^2 Time: 8.7501s position: 23.9649m, Velocity: 0.0714331m/s, Acceleration: -0.7833m/s^2 Time: 8.7601s position: 23.9656m, Velocity: 0.0636001m/s, Acceleration: -0.7833m/s^2 Time: 8.7701s position: 23.9662m, Velocity: 0.0557671m/s, Acceleration: -0.7833m/s^2 Time: 8.7801s position: 23.9667m, Velocity: 0.0479341m/s, Acceleration: -0.7833m/s^2 Time: 8.7901s position: 23.9671m, Velocity: 0.0401011m/s, Acceleration: -0.7833m/s^2 Time: 8.8001s position: 23.9675m, Velocity: 0.0322681m/s, Acceleration: -0.7833m/s^2 Time: 8.8101s position: 23.9678m, Velocity: 0.0244351m/s, Acceleration: -0.7833m/s^2 Time: 8.8201s position: 23.968m, Velocity: 0.0166021m/s, Acceleration: -0.7833m/s^2 Time: 8.8301s position: 23.9681m, Velocity: 0.00876912m/s, Acceleration: -0.7833m/s^2 Time: 8.8401s position: 23.9682m, Velocity: 0.000936123m/s, Acceleration: -0.7833m/s^2 Time: 8.8501s position: 23.9681m, Velocity: 0m/s, Acceleration: -0.7833m/s^2 Simulation is completed. Results were save to: SpringSimul.csv</pre>
---	---

The constants for air density and gravity have now been changed (Mars' air density and gravity). The data shows that despite the mass and area being larger, the constants being reduced increase the distance the car travels. Comparing the distance travelled of the original car to the distance travelled by the car above, we can see that the velocity does not reach as high (maximum of 5.28761m/s with Mars "climate", and 12.788m/s with original constants).

The higher acceleration/deceleration and the higher velocity causes the original car to come to a stop quicker than the "Mars" car, while also travelling a further distance. These small changes in constants allows for programmers/game developers to create different physics within their games (driving on Mars compared to driving on Earth like Beam.NG).



```
Time: 9.06011s position: 75.1469m, Velocity: 0.14163m/s, Acceleration: -2.0601m/s^2
Time: 9.06011s position: 75.1472m, Velocity: 0.121244m/s, Acceleration: -2.0601m/s^2
Time: 9.07011s position: 75.1483m, Velocity: 0.100643m/s, Acceleration: -2.0601m/s^2
Time: 9.08011s position: 75.1492m, Velocity: 0.0800418m/s, Acceleration: -2.0601m/s^2
Time: 9.09011s position: 75.1499m, Velocity: 0.0594408m/s, Acceleration: -2.0601m/s^2
Time: 9.10011s position: 75.1504m, Velocity: 0.0388398m/s, Acceleration: -2.0601m/s^2
Time: 9.11011s position: 75.1507m, Velocity: 0.0182388m/s, Acceleration: -2.0601m/s^2
Time: 9.12011s position: 75.1508m, Velocity: 0m/s, Acceleration: -2.0601m/s^2
Simulation is completed. Results were save to: SpringsSimul.csv
```

When changing the parameters in the if statement " $X > 6$ ", (where $FT = 0$ after 6 metres), the graph shows the same shape/trend. When this statement is changed to " $\text{if}(X > 16)$ " this creates the maximum force potential provided by this spring using Hooke's Law. Although all constants are the same, the car travels the furthest distance possible with this spring potential. We can then assume that changing the spring potential in the equation " $FT = 1.2(16-x)$ " would also increase or decrease the distance travelled.

This code should have implemented functions for each method used, as this would have made the code easier to follow and much tidier.

Conclusion

Even when simulating something as simple as a spring-powered car, we can see how important using numerical integration methods are. We even understand that each method has their advantages and disadvantages, and that we can implement different numerical integrations within the same game engine. The importance of choosing when and where to implement these methods will affect the game's performance, as well as keeping the customer immersed and content with the experience they are being provided.

Appendices

```

public:
    FileWriter(const std::string& filepath) : filepath(filepath)
    {
        file.open(filepath);
        if (!file.is_open())
        {
            std::cerr << "Error: could not open the file for writing.\n";
        }
        else
        {
            file << "Time(s), Position(m), Velocity(m/s), Acceleration(m/s^2)\n";
        }
    }

    ~FileWriter()
    {
        if (file.is_open())
        {
            file.close();
        }
        std::cout << "Simulation is completed. Results were save to: " << filepath;
    }

    void write(float time, float position, float velocity, float acceleration)
    {
        if (file.is_open())
        {
            file << std::fixed << std::setprecision(2) << time << ", "
                << std::fixed << std::setprecision(5) << position << ", "
                << std::fixed << std::setprecision(5) << velocity << ", "
                << std::fixed << std::setprecision(5) << acceleration << "\n";
        }
    }

    void log(float time, float position, float velocity, float acceleration)
    {
        std::cout << "Time: " << time << "s Position: " << position << "m, Velocity: " << velocity << "m/s, Acceleration: " << acceleration << "m/s^2" << std::endl;
    }
};

```

This is the FileWriter.h file, showcasing the code that saves the recorded data to a suitable application.

If the file is open when the simulation is run, then the data will not be saved as it will print out the error "Could not open file for writing". This can be useful if someone else is accessing this data, as it does not allow the simulation to overwrite this file until closed.

This just allows the user to change how the data is displayed in the Excel files, as we can set the precision of the data recorded (significant figures/decimal points), without having to manually edit the Excel sheet themselves.

References

[1] P. J. Davis and P. Rabinowitz. P. J. Davis and P. Rabinowitz. *Methods of Numerical Integration Methods of Numerical Integration* (first ed.) 2007, 1pp.

Available: https://books.google.co.uk/books?hl=en&lr=&id=gGCKdqka0HAC&oi=fnd&pg=PP1&dq=what+is+numerical+integration&ots=NEwwyzBNdM&sig=97rQsDrG7bp004CR45-k1XxXUhU&redir_esc=y#v=onepage&q=what%20is%20numerical%20integration&f=false.

[2] B. N. BISWAS, S. CHATTERJEE, S. P. MUKHERJEE AND S. PAL, A discussion on euler method: A review. A discussion on euler method: A review. *Electronic Journal of Mathematical Analysis and Applications Electronic Journal of Mathematical Analysis and Applications* Vol. 1Vol. 1pp. 296. pp. 296. 2013.

Available: https://journals.ekb.eg/article_309811_cb4e94d217da1b8e29e3c9294ccd8e0a.pdf.

- [3] T. S. Qingkai Kong Alexandre Bayen. T. S. Qingkai Kong Alexandre Bayen. The euler method — python numerical methods. The euler method — python numerical methods, 2020.
Available: <https://pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter22.03-The-Euler-Method.html>.
- [4] Nancy Hill, NASA, Navier-stokes equations. Navier-stokes equations. 2021.
Available: <https://www.grc.nasa.gov/www/k-12/airplane/nseqs.html>.
- [5] Anonymous (-01-31T20:35:31+00:00). *Runge-Kutta 4th Order Method to Solve Differential Equation*. Available: <https://www.geeksforgeeks.org/runge-kutta-4th-order-method-solve-differential-equation/>.
- [6] Anonymous Anonymous Numerical integration in games development. Numerical integration in games development. 2015.
Available: <https://jdickinsongames.wordpress.com/2015/01/22/numerical-integration-in-games-development-2/>.
- [7] Vladimir Dobrushkin. Vladimir Dobrushkin. MATHEMATICA TUTORIAL, part 1.3: Heun methods. MATHEMATICA TUTORIAL, part 1.3: Heun methods. N/A.
Available: <https://www.cfm.brown.edu/people/dobrush/am33/Mathematica/heun.html>.

Assignment 1 ACCB017

GRADEMARK REPORT

FINAL GRADE

88/100

GENERAL COMMENTS

Engagement with Literature: 12/15

Knowledge and Understanding: 14/15

Cognitive and Intellectual Skills: 20/20

Practical Skills: 40/45

Transferable Skills: 2/5

Overall 88/100

PAGE 1

PAGE 2

PAGE 3



Comment 1

Great work

PAGE 4



Comment 2

Am I missing figures one and two?

Ideally what you would do is just after the introduction provide a table of figures providing the source of the figure etc there, rather than using an intext citation like this



Comment 3

Interesting research here!



Comment 4

(A follow on from the comment on the previous figure)

What would have been a good use of caption here is to provide a brief but informative title for the figure.. for example

"A screenshot of the RK4 Methods formulae"

PAGE 5



Comment 5

Fantastic research again. Good work.



Comment 6

This wasn't exactly hookes law, but an adaption.

While $F_t = 1.2(16 - x)$ exhibits a linear relationship between force and a variable (similar to Hooke's Law), it models a system with a linearly diminishing force where the force drops to zero when $x = 16$



Comment 7

Great idea