

Assignment 1

by William Baker Oliver

Submission date: 30-Nov-2024 06:24AM (UTC+0000)

Submission ID: 244181888

File name: 50534_William_Baker_Oliver_Assignment_1_628791_1468790703.docx (3.03M)

Word count: 2840

Character count: 14755

Games Architecture and
Mechanics
Assignment 1: Numerical
Integration

William Baker Oliver

Student Number: 2314716

Computing (Games Development)

Contents

Part 1: Numerical Integration Methods	3
Part 2: Spring-Powered Car	8
External Links.....	17
Bibliography	18

Part 1: Numerical Integration Methods

Numerical Integration, according to Wolfram Mathworld, is “the approximate computation of an integral using numerical techniques” [1]. It essentially allows us to split up complex functions into different steps for easy computation, such as working out the position of a tennis ball after it has been hit by a racket over a certain period of time.

Numerical integration has a variety of applications in computing, especially in the field of video game development. According to Soumya D Mohanty and Sergio Cantu, developers provide immersive experiences to the player by “relying on physics-based virtual worlds, where the consequences of an action are not pre-programmed but computed in real time using some physical laws.” [2]. Most video games rely heavily on physics in their game as a crucial aspect of what makes them tick.



Take Super Monkey Ball: Banana Mania [3] for example. The game revolves around the player controlling a monkey inside a capsule ball and getting it to the goal by rolling it around and tilting the stage. There are a lot of factors to consider; the direction the ball is going, the speed and velocity of the ball, whether it is speeding up or slowing down, whether the ball is falling from a certain height, whether certain forces are acting against the ball, there is a lot to consider when implementing the physics and control of a game like this. This is where numerical integration comes in handy. We can consider all these factors and compute them in different steps to help simulate the physics that the game requires.



Fig.1: A screenshot of the puzzle platformer game Super Monkey Ball: Banana Mania [3], showing one of the stages where the player must roll down a spherical tube to reach the goal without falling. Screenshot taken in game.

For games, we mainly focus on numerical integration with regards to ordinary differential equations, or ODEs. We update certain quantities in time steps, commonly known as Delta Time. One of the most commonly known methods of numerical integration is known as Euler's. Euler's is known as a second-order differential equation, an equation that regards the rates of change. For example, a car driving down the road will have a constantly changing position, with the velocity being the rate of how much the car's position is changing over time. To work out these rates of change, we can use Euler's method to approximate the changes over time. Going back to Super Monkey Ball, Euler's method can be used to model the velocity, position, and acceleration of the ball, by computing them in individual steps.

$$V_{new} = V + A\Delta t$$

Fig.2: A picture showing the formula for working out velocity with Euler's. Taken from [4].

$$P_{new} = P + V\Delta t$$

Fig.3: A picture showing the formula for working out positional values with Euler's. Taken from [4].

The main problem with using Euler's, however, is accuracy. We're calculating the next position by only using the current velocity and position of the iteration, which can cause issues. The larger the time step or the bigger the change with certain dynamics, the more inaccurate the results. This is what's known as a truncation error, which is the difference between the approximated solution and the exact solution, according to Andrew Dixon [5]. Simply put, we don't have enough information for a more accurate prediction of future values.

If we wanted to use a more accurate method, one such example would be to use the Verlet Method. This method differs from Euler's, as we use acceleration to determine the next positional values, as opposed to using velocity. Another differing characteristic of Verlet is that it uses both current and previous positional values to calculate the next positional value. Verlet has many advantages, including being higher-order, which allows for smaller margins of error, as well as requiring only one step as opposed to two, according to Game Engine Architecture [6].

$$\mathbf{r}(t_1 + \Delta t) = 2\mathbf{r}(t_1) - \mathbf{r}(t_1 - \Delta t) + \mathbf{a}(t_1)\Delta t^2 + O(\Delta t^4).$$

Fig.4: A picture of the formula for the Verlet method. Taken from [6].

A place where you will commonly find Verlet being used in games is with particle simulations, such as in a game like Counter-Strike 2 [7] for example. Whenever you shoot anything in the game, whether that be another player, a brick wall or water, you will typically see particles being created and scattering as a result such as blood, wood chips, droplets, etc. The Verlet method is a valuable choice for calculating how particles work, as the method of finding the current position from the previous position helps to simulate more accurate particle effects, as opposed to using Euler's. Verlet can also be used to calculate ragdoll body physics, such as in Garry's Mod [8] for example. Like Euler, it uses little in the way of computational resources.



Fig.5: A screenshot of the FPS game Counter-Strike 2 [7], showing the player shooting the water, causing particles and ripples to appear. Screenshot taken in game.



Fig.6: A screenshot of the game Garry's Mod [8], showing the ragdoll physics in the game. In this example, the enemies on screen ragdoll after the player has killed them with an RPG. Screenshot taken in game.

A problem with the normal Verlet method is that the time step value must be constant for the method to work correctly, so if your time step is tied to a framerate that is constantly changing, this will potentially cause inaccurate data to be produced with

Verlet, according to Jason Dummer [9]. Another issue with this method is that it does not have a proper way to calculate velocity. However, there exist deviations of the Verlet method, such as Velocity Verlet and leapfrog integration, which can account for velocity.

One more method of note with regards to video game development is Runge-Kutta. Runge-Kutta 4th order, or RK4 for short, is a much more accurate integration compared to both Euler's and Verlet. The way that RK4 works, as the name implies, is that it will calculate both the velocity and position in 4 different steps. Think of it as moving from one point to another: when you take 1 large step, there will be a certain area of ground spanning from where you initially were to where you ended up. Think of RK4 as you taking 4 smaller, individual steps in that space to get to where you ended up as opposed to 1 singular large step.

$$\begin{aligned}
 k_1 &= (\Delta x)y'(x, y) \\
 k_2 &= (\Delta x)y'(x + \Delta x/2, y + k_1/2) \\
 k_3 &= (\Delta x)y'(x + \Delta x/2, y + k_2/2) \\
 k_4 &= (\Delta x)y'(x + \Delta x, y + k_3) \\
 y(x + \Delta x) &= y(x) + 1/6[k_1 + 2(k_2) + 2(k_3) + k_4]
 \end{aligned}$$

Fig.7: A picture showing an example of an RK4 formula. Taken from [10].

Using these smaller estimates of position and velocity, we can gain a more accurate final estimate of our velocity and position as opposed to Euler and Verlet. However, the biggest drawback to this approach is that unlike those two methods, it requires "significantly more computer time than multi-step methods of comparable accuracy" according to Richard Fitzpatrick [11]. As we're calculating 4 different positional and velocity values, you can see how this would be more demanding as opposed to doing one simple calculation.

Still, many games will accept this trade off, as they may need the extra accuracy. An example of a game that may need it is a vehicle simulation game, such as Microsoft Flight Simulator X [12], where to simulate the movement and flight of a plane as close as you can to the real world, you would want the most accurate values possible, so RK4 would be used to calculate the plane's position and velocity. Another example is BeamNG.drive [13], a game famously known for simulating realistic car physics.



Fig.8: A screenshot of the vehicle simulation game BeamNG.drive [13], showing a buggy vehicle driving up a hill. Screenshot taken in game.



Part 2: Spring-Powered Car

For part of this assignment, one of the tasks was to write code which simulates a spring-powered dragster car. This required the implementation of a numerical integration method in the code to achieve this.

The task specified formulas for both the tractive force of the spring-powered car, the drag force which acts against the car, and the acceleration of the car, which could be found by calculating the tractive and drag forces. The formulas are as follows:

$$\text{Tractive Force: } F_t = 1.2(16 - x)$$

(Where x = distance travelled.)

$$\text{Drag Force: } F_d = C_r mg + \frac{1}{2} C_d \rho A V^2$$

(Where C_r = rolling resistance coefficient, m = mass, g = acceleration due to gravity, C_d = drag coefficient, ρ = density of the air, A = car's frontal area, V = velocity.)

For the velocity and position of the car, a numerical integration method had to be used. For this task, two methods were initially chosen: Euler's Method and the Verlet Method, to calculate the position of the car in the simulation.

The simulation of the car runs in a while loop, where it will keep iterating and produce results from when it is launched up until velocity reaches exactly or below zero, which means the car has stopped. Once this happens, a break is applied to the loop and both the code and simulation ends. The values are modelled at an iteration rate of 0.01 seconds, known as Delta Time.

As we have different methods of calculation for the car's position, a simple command line menu was created which allows the user to enter a number, which will dictate which method will be used when the simulation is ran. After the traction and drag forces are calculated using the provided formulas, a switch case then be used to switch which method is used depending on the value entered at the start. This makes it easier, as it removes the need to comment and uncomment out multiple lines of code to run a different simulation.

```

//Chooses the method to be used in the loop.
while (true)
{
    std::cout << "1 for Euler's Method, 2 for Verlet, 3 for RK4." << std::endl;
    std::cin >> choice;

    if (choice == 1 || choice == 2 || choice == 3)
    {
        break;
    }

    std::cout << "Invalid Choice." << std::endl;
}

```

Fig.1: A screenshot of the code showcasing the command line menu. Screenshot taken from Visual Studio.

According to Gabor Szauer, the “acceleration of an object affects its velocity. The new velocity is the same as the old velocity plus acceleration over some period of time.” [4]. So, for the Euler method, velocity is calculated by multiplying the current acceleration values by delta time and adding it to the current value of velocity. After this, we work out positional values by adding velocity multiplied by delta time to the current position.

$$V_{new} = V + A\Delta t$$

Fig.2: A picture showing the formula used for working out velocity with Euler's. Taken from [4].

$$P_{new} = P + V\Delta t$$

Fig.3: A picture showing the formula used for working out positional values with Euler's. Taken from [4].

The Verlet method here works by taking the position from the previous iteration and subtracting it from the current iteration's position, which is multiplied by two. For this to work we store the car's current position in a variable named 'currentPos', then

once we have calculated the car's position, we then take the 'currentPos' value and assign it to a variable called 'oldPos'. This ensures that we can keep using the previous iteration's position for this method. We then add the acceleration values multiplied by Delta Time squared by 2. This produces our position values.

$$\mathbf{r}(t_1 + \Delta t) = 2\mathbf{r}(t_1) - \mathbf{r}(t_1 - \Delta t) + \mathbf{a}(t_1)\Delta t^2 + O(\Delta t^4).$$

Fig.4: A picture of the Verlet method formula which was implemented within the Spring-Powered Car code. Taken from [6].

```

class 2:
    //This is the verlet method, where we use both the old and current position of the spring-powered car to calculate the distance that the car will travel, without needing to use velocity.
    velocity += accel * deltaTime;
    //old_pos = "Verlet method" - as old_pos;
    currentPos = carPos;
    carPos = 2 * currentPos - oldPos;
    carPos += accel * std::pow(deltaTime, 2);
    oldPos = currentPos;
    break;

```

Fig.5: A screenshot of the code showcasing the Verlet method implementation. Screenshot taken from Visual Studio.

To save the results after the simulation has been ran, a class called 'fileWriter' was used here, which will write the results to a .csv file for use with Microsoft Excel. This code was written by Nathan Morgan and used with permission, with it being modified slightly so that the acceleration values of each time step could also be saved.

```

//Spring.cpp
#include <string>
#include <iostream>
#include <fstream>
#include <iomanip>

//The code for the FileWriter class was created by Nathan Morgan. Used with permission.

class FileWriter
{
private:
    std::string filepath;
    std::ofstream file;
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 saved to: " << filepath << std::endl;
    }
    void log(float time, float position, float velocity, float accel)
    {
        std::cout << "Time: " << time << "\n";
        std::cout << "Position: " << position << "\n";
        std::cout << "Velocity: " << velocity << "\n";
        std::cout << "Acceleration: " << accel << "\n";
        std::cout << "-----\n";
    }
    void write(float time, float position, float velocity, float accel)
    {
        file << time << "," << position << "," << velocity << "," << accel << "\n";
    }
};

```

Fig.6: A screenshot of the 'fileWriter.h' code, which is used to save the results to a .csv file. Screenshot taken from Visual Studio.

By saving to a file, we can analyse the results given from the simulation and use them to create visual graphs.

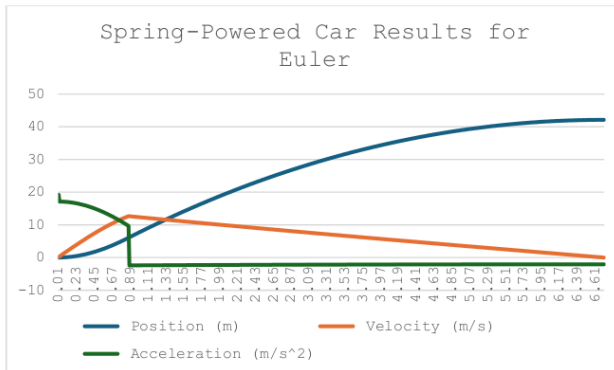


Fig.7: A chart showing the results of the spring-powered car simulation, using Euler's to calculate the position and velocity. Created with Excel.

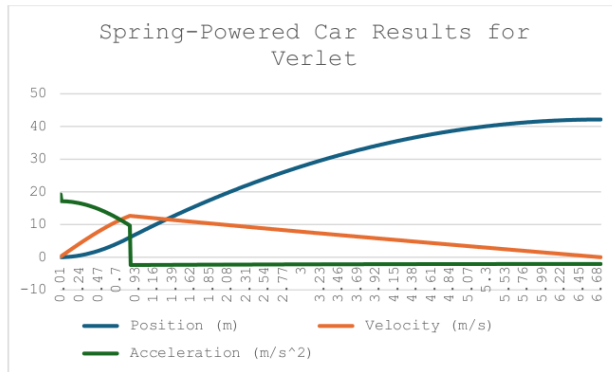


Fig.8: A chart showing the results of the spring-powered car simulation, using Verlet to calculate the position and velocity. Created with Excel.

Using the data gathered from running the simulation with both Euler's and Verlet, one thing that is very much noticeable is that the data produced finishes at the exact same time step, with the modelled values looking nearly identical. However, if you take a closer look at the results for positional values you can see that whilst the values initially look identical, certain time steps have different results. This can be seen later down the tables, as it seems to start out similar, however at a certain point the values seem to be different between the two.  10

	A	B	C	D	E		A	B	C	D	E
	Time (s)	Position (m)	Velocity (m/s)	Acceleration (m/s²)			Time (s)	Position (m)	Velocity (m/s)	Acceleration (m/s²)	
108	1.08	8.6095	12.166	-2.3632		108	1.08	8.6095	12.166	-2.3632	
109	1.09	8.791	12.1424	-2.362		109	1.09	8.791	12.1424	-2.362	
110	1.1	8.9122	12.1189	-2.3609		110	1.1	8.9122	12.1189	-2.3609	
111	1.11	9.0331	12.0952	-2.3597		111	1.11	9.0331	12.0952	-2.3597	
112	1.12	9.1538	12.0716	-2.3585		112	1.12	9.1538	12.0716	-2.3585	
113	1.13	9.2743	12.0481	-2.3574		113	1.13	9.2743	12.0481	-2.3574	
114	1.14	9.3945	12.0243	-2.3562		114	1.14	9.3945	12.0243	-2.3562	
115	1.15	9.5146	12.0009	-2.3551		115	1.15	9.5146	12.001	-2.3551	
116	1.16	9.6343	11.9774	-2.3539		116	1.16	9.6343	11.9774	-2.3539	
117	1.17	9.7539	11.9539	-2.3528		117	1.17	9.7539	11.9539	-2.3528	
118	1.18	9.8732	11.9304	-2.3516		118	1.18	9.8732	11.9304	-2.3516	
119	1.19	9.9922	11.9069	-2.3505		119	1.19	9.9922	11.9069	-2.3505	
120	1.2	10.1111	11.8834	-2.3493		120	1.2	10.1111	11.8834	-2.3493	
121	1.21	10.2297	11.8599	-2.3482		121	1.21	10.2297	11.8599	-2.3482	
122	1.22	10.348	11.8364	-2.347		122	1.22	10.3481	11.8364	-2.347	
123	1.23	10.4662	11.813	-2.3459		123	1.23	10.4662	11.813	-2.3459	
124	1.24	10.5841	11.7895	-2.3448		124	1.24	10.5841	11.7895	-2.3448	
125	1.25	10.7017	11.7661	-2.3436		125	1.25	10.7018	11.7661	-2.3436	
126	1.26	10.8191	11.7426	-2.3425		126	1.26	10.8192	11.7427	-2.3425	
127	1.27	10.9363	11.7192	-2.3414		127	1.27	10.9364	11.7192	-2.3414	
128	1.28	11.0533	11.6958	-2.3403		128	1.28	11.0533	11.6958	-2.3403	
129	1.29	11.17	11.6724	-2.3392		129	1.29	11.1701	11.6724	-2.3392	
130	1.3	11.2865	11.6491	-2.338		130	1.3	11.2866	11.6491	-2.338	
131	1.31	11.4028	11.6257	-2.3369		131	1.31	11.4028	11.6257	-2.3369	
132	1.32	11.5188	11.6023	-2.3358		132	1.32	11.5189	11.6023	-2.3358	
133	1.33	11.6346	11.579	-2.3347		133	1.33	11.6346	11.579	-2.3347	
134	1.34	11.7501	11.5556	-2.3336		134	1.34	11.7502	11.5557	-2.3336	
135	1.35	11.8655	11.5323	-2.3325		135	1.35	11.8655	11.5323	-2.3325	
136	1.36	11.9806	11.509	-2.3314		136	1.36	11.9806	11.509	-2.3314	
137	1.37	12.0954	11.4857	-2.3303		137	1.37	12.0955	11.4857	-2.3303	
138	1.38	12.21	11.4624	-2.3292		138	1.38	12.2101	11.4624	-2.3292	
139	1.39	12.3246	11.4391	-2.3281		139	1.39	12.3245	11.4391	-2.3281	
140	1.4	12.4386	11.4159	-2.327		140	1.4	12.4386	11.4159	-2.327	
141	1.41	12.5525	11.3926	-2.326		141	1.41	12.5526	11.3926	-2.326	
142	1.42	12.6662	11.3694	-2.3249		142	1.42	12.6663	11.3694	-2.3249	
143	1.43	12.7797	11.3461	-2.3238		143	1.43	12.7797	11.3461	-2.3238	
144	1.44	12.8929	11.3229	-2.3227		144	1.44	12.893	11.3229	-2.3227	

Fig.9: A picture showing the tables for the Euler simulation results (on the left) and the Verlet simulation results (on the right). The positional values are shown to be different in certain time steps. Screenshot taken from Excel.

The velocity and acceleration values are both nearly identical between the two, which makes sense considering they both are using the same equations to work out these values. With the Verlet method, we aren't using velocity to work out position, so we must use the same line of code "velocity += accel * deltaTime;" as we still need it for calculating the drag force and acceleration.

According to Dev Ramtal and Adrian Dobre, methods like Euler's are "fast and very easy to code, but it is inaccurate and can be unstable, which can lead to problems with some types of simulations." [14]. To see the difference a more accurate and computationally heavy method would have on the simulation results, an RK4 implementation was added to the code as well. This method works by calculating the velocity and position in 4 different steps. Unlike the Verlet method implementation, the RK4 implementation has a dedicated way to calculate velocity. Essentially, RK4 is "using a trial step at the midpoint of an interval to cancel out lower-order error terms", according to Wolfram Mathworld [15].

```

//This is the Runge-Kutta method. Run. This method splits velocity and position calculation into 4 steps, before calculating the final result, which allows the calculated results to be more precise.
//Add const = "Using the Method" to the end.

velocityp1 = accel * deltaTime;
positiond1 = velocity * deltaTime;

velocityp2 = accel * deltaTime;
positiond2 = (velocity + velocityp1 * const1) * deltaTime;

velocityp3 = accel * deltaTime;
positiond3 = (velocity + velocityp2 * const1) * deltaTime;

velocityp4 = accel * deltaTime;
positiond4 = (velocity + velocityp3 * const1) * deltaTime;

velocity = (velocityp1 + const2 + velocityp2 + const2 + velocityp3 + const2 + velocityp4) / const3;
position = (positiond1 + const2 + positiond2 + const2 + positiond3 + const2 + positiond4) / const3;
carPos += position;
break;

```

Fig.10: A screenshot of the RK4 code implementation for the Spring-Powered Car simulation. Screenshot taken from Visual Studio.

Running the simulation with the RK4 method, we can see the simulation finishes in a much quicker time, with less time steps. This means that whilst we have less data than the Euler and Verlet simulations, we have much more accurate information as a result, due to the computational power used here.

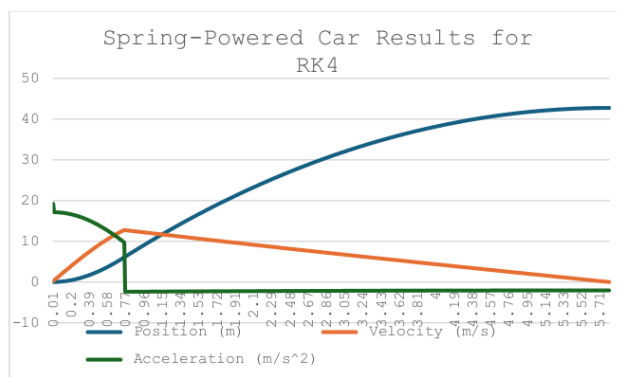


Fig.11: A chart showing the results of the spring-powered car simulation, using RK4 to calculate the position and velocity. Created with Excel.

After the simulations had been completed with the original constants, more simulations were run, however with changed values for certain constants, such as increasing the air density or decreasing the mass of the car. For example, a simulation was run with the rolling resistance coefficient changed to 0.42f instead of 0.21f. The results showed that the car stopped much quicker than usual at a shorter distance, as velocity dropped at a much faster rate. This shows the effect the rolling resistance has on the car.

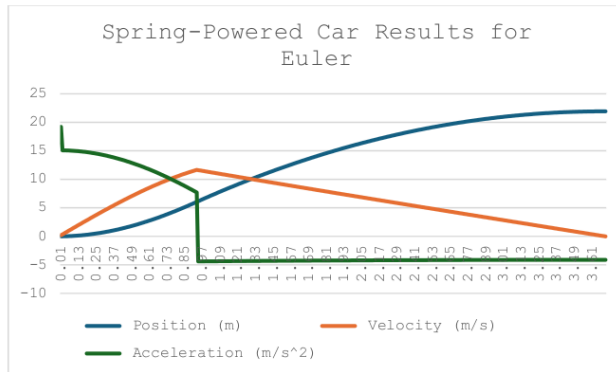


Fig.12: A chart showing the results of the spring-powered car simulation with modified rolling resistance coefficient values, using Euler's to calculate the position and velocity. Created with Excel.

These were all saved to a singular Excel file, organised into separate sheets with the results of each simulation plotted to a line graph. A video was also created to show the code in action running the simulation.

External Links

Spring-Powered Car Implementation with C++: [SpringPoweredCar.zip](#)

Spring-Powered Car Simulation Results: [Spring-Powered Car Simulation Results.xlsx](#)

YouTube video showing the code being ran:

<https://www.youtube.com/watch?v=r1wf7uP1HS8>

Bibliography



- [1] E. W. Weisstein, "Numerical Integration," Wolfram Research, Inc., 20 November 2024. [Online]. Available: <https://mathworld.wolfram.com/NumericalIntegration.html>. [Accessed 20 November 2024].
- [2] S. D Mohanty and S. Cantu, "Teaching introductory undergraduate physics using commercial video games," *Physics Education*, vol. 46, no. 5, p. 570–577, 2011.
- [3] Ryu Ga Gotoku Studio, "Super Monkey Ball: Banana Mania," Sega, Tokyo, 2021.
- [4] G. Szauer, *Game Physics Cookbook*, Birmingham: Packt Publishing, 2017.
- [5] A. Dixon, *Modern Aspects of Power System Frequency Stability and Control*, Cambridge, Massachusetts: Academic Press, 2019, pp. 135–189.
- [6] J. Gregory, *Game Engine Architecture*, Boca Raton: CRC Press, 2009, pp. 640–641.
- [7] Valve Corporation, "Counter-Strike 2," Valve Corporation, Bellevue, 2023.
- [8] Facepunch Studios, "Garry's Mod," Valve Corporation, Bellevue, 2006.
- [9] J. Dummer, "A Simple Time-Corrected Verlet Integration Method," *gamedev.net*, 1 January 2011. [Online]. Available: <http://archive.gamedev.net/archive/reference/programming/features/verlet/>. [Accessed 20 November 2024].
- [10] D. M. Bourg, *Physics for Game Developers*, Sebastopol: O'Reilly, 2002, pp. 180–185.
- [11] R. Fitzpatrick, "Advanced integration methods," 29 March 2006. [Online]. Available: <https://farside.ph.utexas.edu/teaching/329/lectures/node40.html>. [Accessed 20 November 2024].
- [12] M. G. Studios, "Microsoft Flight Simulator X," Xbox Game Studios, Redmond, 2006.
- [13] BeamNG GmbH, "BeamNG.drive," BeamNG GmbH, Bremen, 2013.
- [14] A. D. Dev Ramtal, *Physics for JavaScript Games, Animation, and Simulations*, New York City: Apress, 2014, p. 377.

- [15] E. W. Weisstein, "Runge-Kutta Method," Wolfram Mathworld, 20 November 2024. [Online]. Available: <https://mathworld.wolfram.com/Runge-KuttaMethod.html>. [Accessed 20 November 2024].

Assignment 1

GRADEMARK REPORT

FINAL GRADE

78/100

GENERAL COMMENTS

Engagement with Literature: 10/15

Knowledge and Understanding: 10/15

Cognitive and Intellectual Skills: 15/20

Practical Skills: 40/45

Transferable Skills: 3/5

Overall 78/100

Comment:

Overall a great piece of work, Will.

A table of figures to go with the table of contents would have been nice.

See inline comments also

PAGE 1

PAGE 2

PAGE 3



Comment 1

as great as they are, a website isnt an ideal source of information when there are more academically credible sources stating the same things



Comment 2

In academic writing, this phrase may be too casual or colloquial.

PAGE 4



Comment 3

good



Comment 4

Well said



Comment 5

This is where references would be ideal. Confirming what you have stated.



Comment 6

Something I notice about your writing style is that you often structure it as

"fact... according to ...[11]."

you could just simply write

"the fact [11]."

or structure it as

"Richard Fitzpatrick states ... fact [11]."



Comment 7

Not a strong start to a sentence



Comment 8

Screenshots should only be included when they serve a clear purpose in illustrating or demonstrating a specific point. Irrelevant or excessive screenshots may detract from the clarity and focus of your work. Ensure that each screenshot directly contributes to the content being discussed.



Comment 9

In academic writing, particularly in scientific and technical papers, the use of the first person (e.g., "I", "we") is generally discouraged as it can make the writing appear subjective and less formal. The focus of academic work should be on the research, analysis, or findings rather than the researcher. Using an objective tone ensures that the argument or evidence takes precedence over personal opinions or experiences, which enhances the perceived credibility and professionalism of the work. For example, instead of saying, "I conducted the experiment," you could write, "The experiment was conducted," which maintains an impersonal and objective stance. To improve in this area, aim to use passive voice judiciously or rephrase sentences to emphasise the action rather than the actor. Additionally, consider adopting phrases like "the research indicates" or "it was observed"

to convey findings without self-referencing. Reviewing published papers in your field can provide valuable examples of how to maintain an objective tone while ensuring clarity and precision.

PAGE 12

PAGE 13



Comment 10

Interesting research, and a good review of data.

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18



Comment 11

A table of references and a bibliography both list sources used in academic work, but they differ in purpose and scope. A table of references includes only the sources directly cited within the text, providing full details of each reference as they appear in the work. It is organised in alphabetical order and ensures that readers can easily locate the sources mentioned in the document. In contrast, a bibliography is a broader list that includes all sources consulted during the research process, whether cited in the text or not. While a table of references is focused on specific citations, a bibliography may also include background readings, general sources of information, or additional materials that contributed to the development of the work.

PAGE 19
