

# ACCB5017 Assignment 1 - 2112389

*by* Joshua Piecko

---

**Submission date:** 18-Nov-2022 08:51PM (UTC+0000)

**Submission ID:** 189497913

**File name:** 33382\_Joshua\_Piecko\_ACCB5017\_Assignment\_1\_-\_2112389\_467669\_1822851303.docx (132.54K)

**Word count:** 3531

**Character count:** 20623

Trinity University  
[ACCB5017] Game Architecture and Mechanics  
*Game Physics*  
[ASSIGNMENT]

*Joshua Piecko*

*Student Number: 2112389*

*16<sup>th</sup> November 2022*

**Table Of Contents:**

# 1. Numerical Integration used in Games Development:

## Introduction:

Numerical integration is a numerical method that involves the approximate computation of the definite integrals. The methods of numerical integration are used to approximate integrals using a suitable weighted sum of the values of the function being integrated at several points. To acquire more accurate approximations, the integration interval is divided into many smaller fragments. The result is the sum of the estimates of the integrals in the given subintervals. An interval is almost always divided into the same subintervals, but more accurate algorithms adjust the step according to the rate of the variation in the given function. [1]

The simulation of life-like physics properties in games development is a critical element to achieving a sense of realism, from the drag felt whilst racing a car around a track to the curvature of an arrow fired by a bow, you can find physics properties fundamentally everywhere. However, there is no true singular method of achieving such results and the challenge comes without ease, so developers need to determine when and how these methods can be utilized in varying scenarios.

There are several methods to choose from with each having attributed advantages and disadvantages, these methods can be positioned in several different categories dependent on the level of accuracy that each one can approximate for, which is based on the number of steps required to calculate the derivative as a whole:

- First Order Integration
  - o Explicit Euler
  - o Implicit Euler (Backwards Euler)
  - o Semi-Implicit Euler (Symplectic Euler)
- Second Order Integration
  - o Verlet
- Higher Order Integration
  - o Runge-Kutta 4

## Integrating the Equations of Motion

Relation among **velocity**, **distance**, **time**, and **acceleration** is referred to as the equations of motion.[2] For this Newtons second law of motion can be considered as  $F=ma$  and  $a=F/m$  [3], where:

$$Force = Mass \cdot Acceleration$$

**&**

$$Acceleration = \frac{Force}{Mass}$$

It must be understood that acceleration is the derivative of velocity and position is the derivative of acceleration[4]. This can be represented in calculus terms as:

$$\frac{dv}{dt} = a$$

&

$$\frac{dx}{dt} = v$$

## First Order Integration Methods:

### *Explicit Euler*

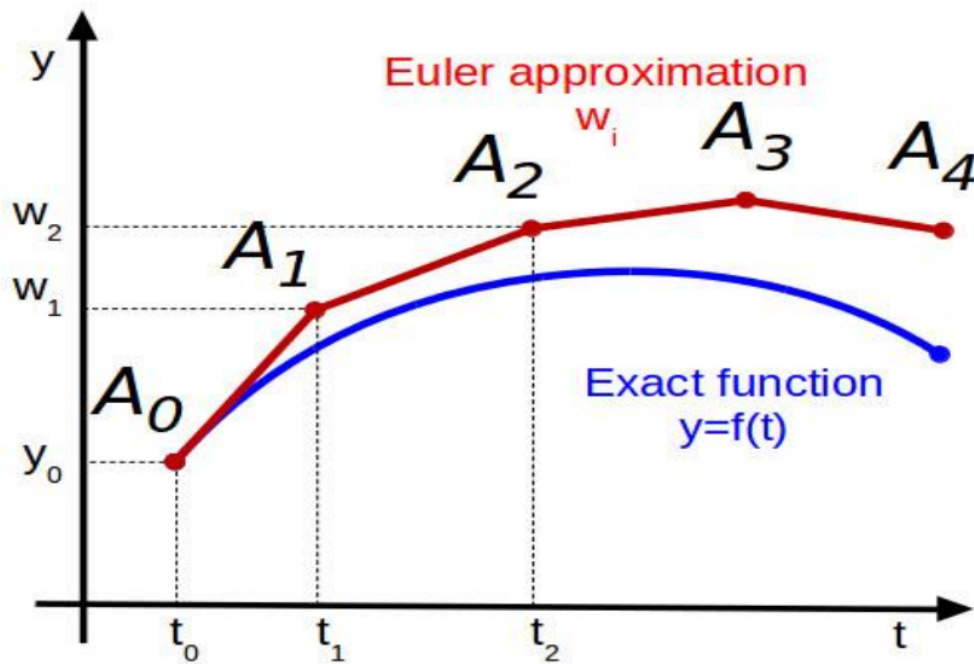
Being one of the simpler numerical integration methods used by game developers. Albeit excellent for its efficiency, the generic Euler method is proven to be the least accurate. This method utilizes an algorithm that takes the initial position and velocity, then calculates the next position and velocity after a given period. This calculation is then repeated using a time-step interval i.e., 250ms – represented by **dt**.

$$\text{Velocity}_{n+1} = \text{Velocity}_n + (\text{Acceleration}_n * dt)$$

&

$$\text{Position}_{n+1} = \text{Position}_n + (\text{Velocity}_n * dt)$$

Once the first variables have been processed, they can be used to calculate the following derivation. When using this method, it is guaranteed to obtain an error value, dependent on the value used for the step size. The value of this error is the difference between the solution and the estimation. The rule of thumb is that the error value will continue to grow as the time passes, therefore suggesting to us that there may be a practical limit for when this method can be utilized by developers. Especially when working with incredibly small interval values. The interval value can be decreased to obtain more accurate results that are a closer representation of the actual solution. However, this is considered impractical. For example, if we were to use a time-step value of 20ms over 20 seconds this would require a total of 1000 calculations to be computed. It is additionally critically vital to note that regardless of how small of value we declare for the time interval the explicit Euler method will always produce an increasing error value over time.



[Figure 1: Graphical Representation of Euler Integration Method]

Ultimately, this Euler method provides a great grounding to develop a conceptual understanding that can be embellished when considering the use of other integration methods. However due to the lack of accuracy given the expectation that the slope at the current point is a good estimate of the slope on the interval, as demonstrated above. We can see that the approximation drifts from the actual function, incrementally adding energy to the system. These approximation displacements become incrementally worse the further we are from the known initial value, especially when using a larger time-step value. This would otherwise require extreme amounts of computation power to navigate.

This integration method is typically used when the impacts of the error value won't have a deciding impact on what occurs next in the game. For example, the transition between the characters jump to sprint animation[7].

### ***Implicit Euler***

This method differs to the previous Euler technique as it calculates the derivative and then proceeds to use that derivative to calculate the approximation. The implicit Euler is often referred to as the “Backwards Euler”, and it comes as no surprise with the technique that is used. The method differs to the previous Euler technique used by calculating the derivative and then proceeding to use that derivative to calculate the new approximation. The formulas used to approximate the new velocity and position can be seen below[6]:

$$\text{Velocity}_{n+1} = \text{Velocity}_n + (\text{Acceleration}_{n+1} * dt)$$

$$\text{Position}_{n+1} = \text{Position}_n + (\text{Velocity}_{n+1} * dt)$$

Albeit another first order integration method, meaning there is still likely to be an error in approximation, it can be considered more stable and more accurate than the explicit Euler method. Even when using a larger time-step interval value. This can be seen as a slower method of integration than the explicit Euler method, due particularly to the unknown value of  $\text{Acceleration}_{n+1}$  we are required to approximate additional values, increasing the overall complexity.

This method can be used by game developers for simulating a handful of different things including but not limited to spring systems, the calculation of a tyre’s friction as it drifts around a corner, and as mentioned above, changes in acceleration. Although the method is extremely accurate and computationally less complex than its counterpart higher order integration methods, we can assume that this method would require too much processing power than necessary for it to be considered for use.

### ***Symplectic Euler***

The Symplectic Euler method is the golden mean between the explicit and implicit Euler methods, hence why some may refer to it as ‘semi-implicit’ Euler. This method approximates the values for both velocity and position using the explicit and implicit methods respectively. Example formulas for approximating the new Velocity and Position can be seen below[6]:

$$\text{Velocity}_{n+1} = \text{Velocity}_n + (\text{Acceleration}_n * dt)$$

$$\text{Position}_{n+1} = \text{Position}_n + (\text{Velocity}_{n+1} * dt)$$

As this method inherits functions used in the explicit and implicit Euler methods many of the key points can be found above. A general advantage to using the symplectic Euler method is that it provides more stability than the standard explicit method. Thus meaning output of more accurate approximations. The method is generally used within the games development industry to simulate approximated rigid body movements [5].

## ***Verlet***

This method can be considered one of the more popular methods of numerical integration in games development. The Verlet method is a higher order integration method, meaning that the approximations we produce are far more accurate yielding lower error coefficient whilst sustaining a simplicity that makes implementation hassle free. It is relatively efficient due to the reduced time required to approximate values when put in contrast to higher order integrators.

The Verlet method differs from the previously mentioned methods of integration as it utilizes the acceleration to approximate the position, whereas previously, for example, in the Explicit Euler method we approximated the velocity from the acceleration then estimated the position using the velocity. Below can be seen an example of how to utilize this formula to approximate and estimate the new values for position and velocity[6]:

$$\text{Position}_{n+1} = 2 * \text{Position}_n + \text{Position}_{n-1} + \text{Acceleration} * dt^2$$

$$\text{Velocity}_{n+1} = (\text{Position}_n - \text{Position}_{n-1})/dt$$

This method is limited in its accuracy due to it being only an approximation, if we were intending to implement in objects where there could be significant fluctuations in velocity changes, for example, a car drifting around a racetrack, we can expect relatively low accuracy. This being a problem again if we assume the need to approximate instantaneous velocity changes that may happen following a collision with another object.

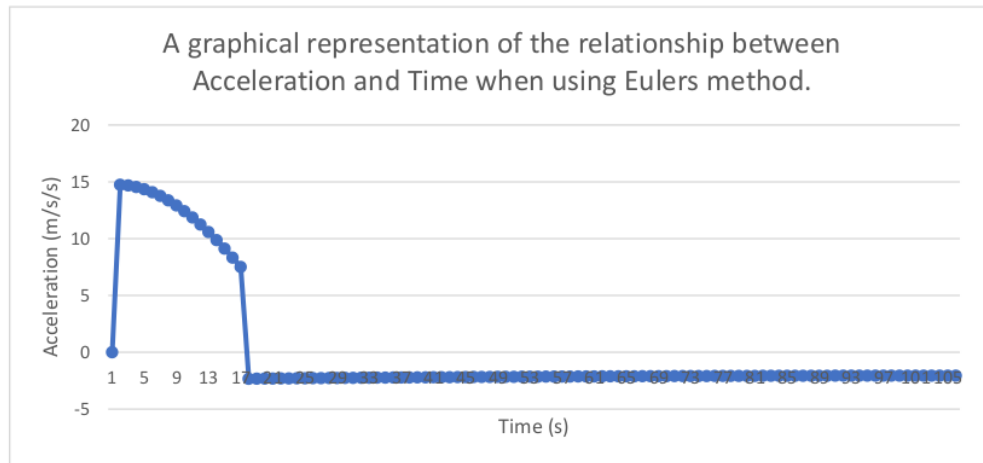
Verlet imposes a complexity that is like that of a Euler step. It worth noting that Verlet is symplectic; this meaning it conserves the internal energy. It is used by many game developers due to the low requirement for high precision floating point values[7]. This can also be considered a disadvantage in some scenarios where the approximations being returned are high precision floating point values themselves. Examples of when games developers might use Verlet integration systems might include for example, how to; Simulate Projectile Trajectories, in particle systems E.g., fire, waterfalls, clouds or the simulation of cloth-like materials[5].

## ***Runge-Kutta 4***

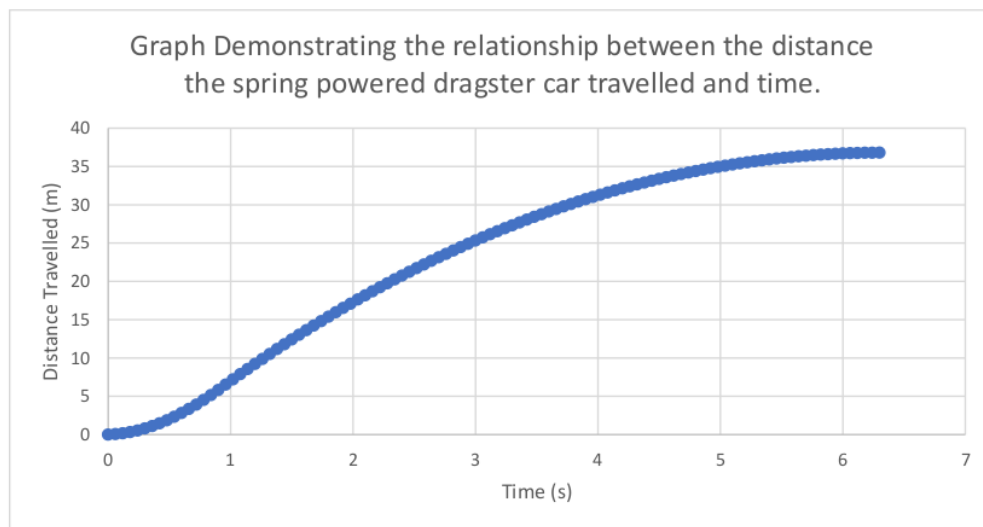
The Runge-Kutta 4<sup>th</sup> Order (RK4) method is the most exercised method of the Runge-Kutta method range that are used in games development. When put in contrast to the integration methods previously mentioned, the RK4 method demonstrates an accuracy like no other. This being thanks to the improved approximation series that is yielded using 4<sup>th</sup> Order precision. Although, this comes at an expense of being 4 times more expensive to compute than the explicit Euler since it requires additional calculations to be completed. Due to the increased computational complexity, this method is reserved by games developers for simulations where accuracy is critical to the results produced, i.e., when a weapon is fired in a FPS or when a dart is thrown in a game. Another great example of when this method may be utilized would be e.g. a satellite orbiting a body, and additionally is used in flight simulators to provide the operator with a realistic flight simulation[5][7].

## 2. Simulating the movement of a spring powered model dragster car using methods of numerical integration.

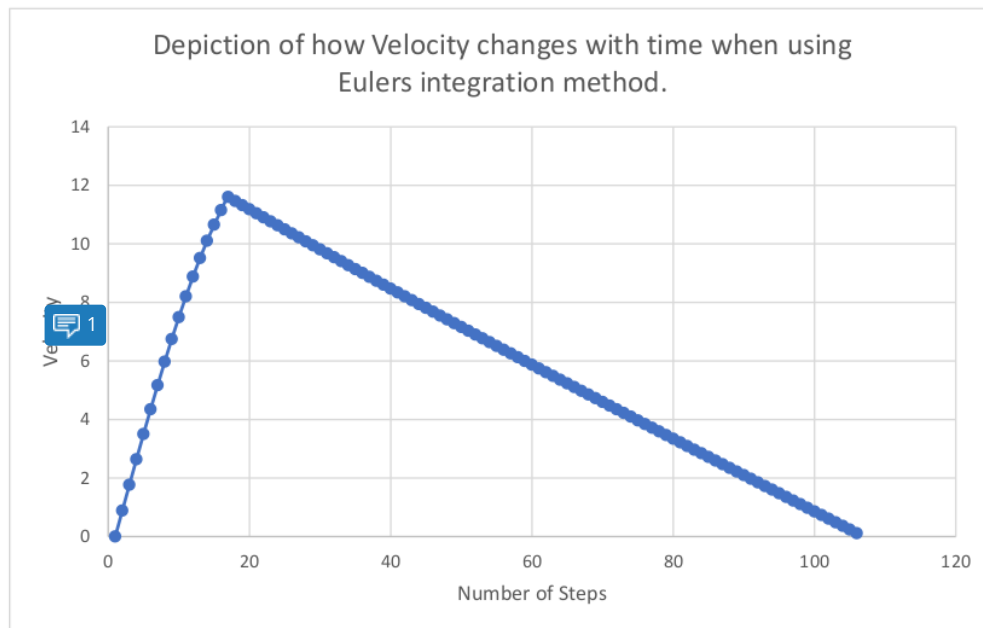
### Graphical Representation of Results



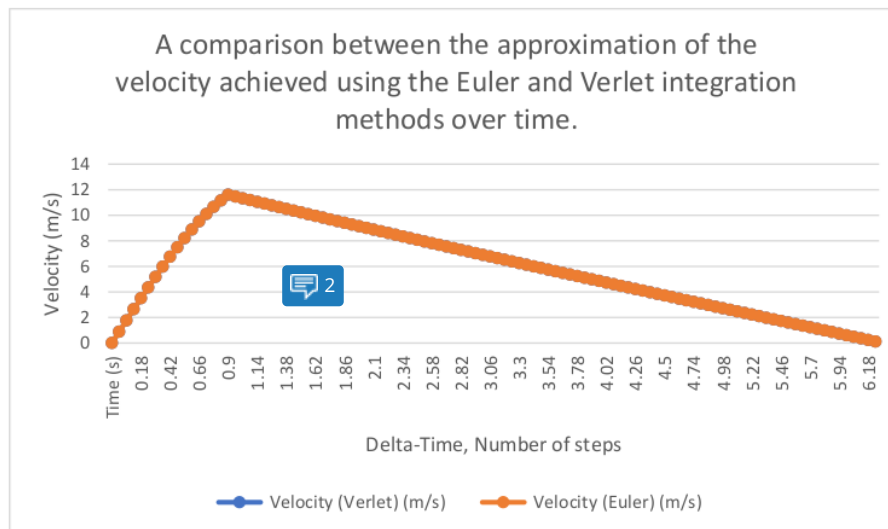
[Figure 2: A graphical representation of the relationship between Acceleration and Time when using Eulers method.]



[Figure 3: A graphical representation of the relationship between Distance Travelled and Time when using Eulers method.]

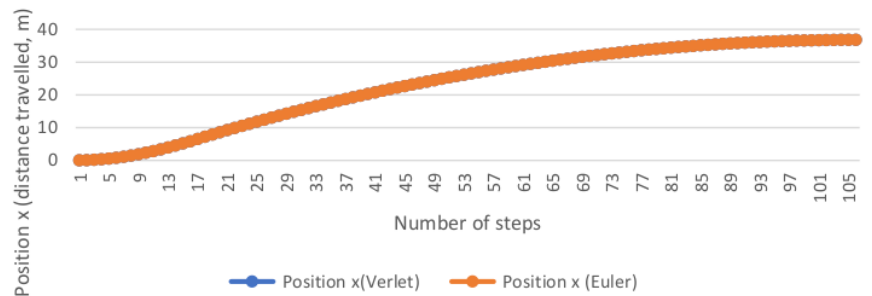


[Figure 4: A graphical representation of the relationship between Velocity and Time when using Eulers integration]



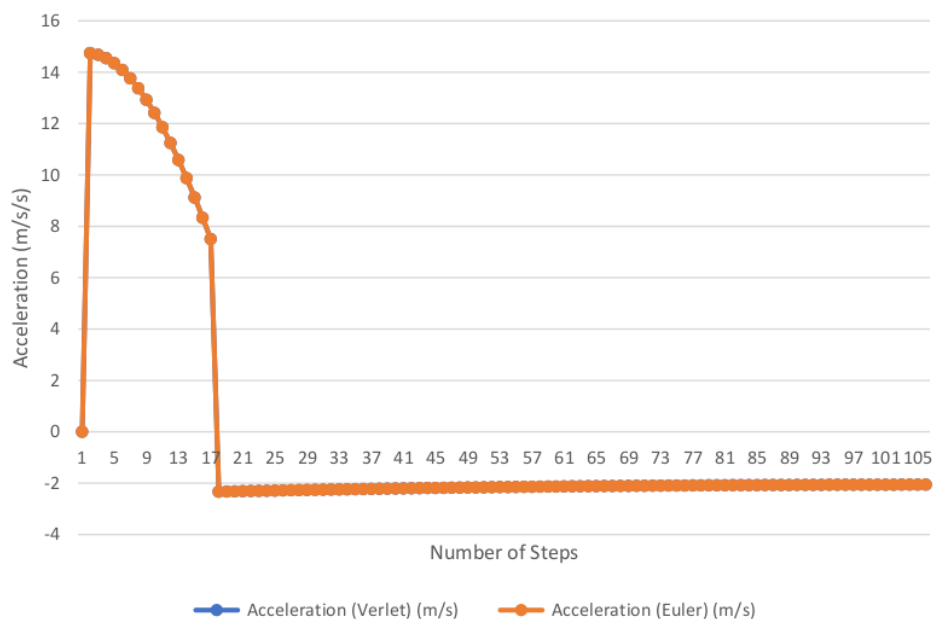
[Figure 5: A comparison between the approximation of the velocity achieved using Euler and Verlet integration methods over time.]

A comparison between the approximation of the position calculated using the Euler and Verlet integration methods over time.



[Figure 6: A comparison between the approximation of the position calculated using the Euler and Verlet integration methods over time.]

A comparison between the approximation of the velocity achieved using the Euler and Verlet integration methods over time.



[Figure 7: A comparison between the approximation of the Velocity calculated using the Euler and Verlet integration methods over time.]

## Analytically Derived Approximations – True Approximations

**Table:**

| <i>Dt<br/>(delta<br/>time)</i> | <i>Tractive<br/>Force</i> | <i>Drag Force</i>  | <i>Acceleration</i> | <i>Velocity</i>    | <i>Position</i>      |
|--------------------------------|---------------------------|--------------------|---------------------|--------------------|----------------------|
| 0                              | 16.8                      | 0                  | 0                   | 0                  | 0                    |
| 0.06                           | 16.8                      | 2.0601             | 14.7399             | 0.884394           | 0.05306364           |
| 0.12                           | 16.736323632              | 2.0614296596703012 | 14.67489397233      | 1.7648876383398    | 0.158956898300388    |
| 0.18                           | 16.60925172204            | 2.06539520823914   | 14.54385651380086   | 2.6375190291678514 | 0.317208040050459084 |

### Conclusion

Given the graphical representation of data across the series of time-steps demonstrating approximately similar results we can propose the idea that given the smaller value used for the time increment in this calculation, there are no significant differences between the overall results. Across the chart there are few discrepancies where the values seem to be off by an error of 0.000001%, otherwise we can enumerate that if we were to have given a larger value for the incremental time-step modifier we could have expected to see proportionately different results. This is an outcome that is a result of both machine level approximation errors when using floating point numbers in the lower level language used as well as the overall efficiency and increased precision achieved using Verlet's method of integration. To achieve a greater level of precision an algorithm similar to the likes of Rutte-Kunga 4 can be used.

## References

- [1] EduGamesS.A., Numerical Integration, mathematics.live, 2021. [Online]. Available: <https://mathematics.live/en/numerical-integration> [Accessed: Nov, 2. 2022].
- [2] G. Elert, Equations of Motion, physics.info, 2022. [Online]. Available: <https://physics.info/motion-equations/> [Accessed: Nov, 2. 2022].
- [3] Britannica, Newton's Second Law  $F=ma$ , Britannica, 2022. [Online]. Available: <https://www.britannica.com/science/Newtons-laws-of-motion/Newtons-second-law-F-ma> [Accessed: Nov, 1. 2022].
- [4] University of Austin Texas, In Physics, UT Calculus, 2022. [Online]. Available: [https://web.ma.utexas.edu/users/m408n/CurrentWeb/LM3-7-2.php#:~:text=In%20Physics,Derivatives%20with%20respect&text=Velocity%20is%20the%20derivative%20of,\(x\(t\)\).](https://web.ma.utexas.edu/users/m408n/CurrentWeb/LM3-7-2.php#:~:text=In%20Physics,Derivatives%20with%20respect&text=Velocity%20is%20the%20derivative%20of,(x(t)).) [Accessed: Nov 4. 2022].
- [5] J. Dickinson, Numerical Integration in Games Development, jdickinsongames, 2015. [Online]. Available: <https://jdickinsongames.wordpress.com/2015/01/22/numerical-integration-in-games-development-2/#comments>. [Accessed: Nov 4. 2022].
- [6] G. Fiedler, Integration Basics, GafferOnGames, 2004. [Online]. Available: [https://gafferongames.com/post/integration\\_basics/](https://gafferongames.com/post/integration_basics/) [Accessed: Nov 8. 2022].
- [7] D. Ramtal & A. dobre, Physics for JavaScript Games Animation and Simulations, Google Docs. [Online]. Available: [https://canvas.projekti.info/ebooks/Apress%20Physics%20for%20JavaScript%20Games%20Animation%20and%20Simulations,%20With%20HTML5%20Canvas%20\(2014\).pdf](https://canvas.projekti.info/ebooks/Apress%20Physics%20for%20JavaScript%20Games%20Animation%20and%20Simulations,%20With%20HTML5%20Canvas%20(2014).pdf) [Accessed: Nov, 12. 2022].
- [Figure 1] <https://x-engineer.org/euler-integration/>  
[Figures 2-7] J. Piecko, UWTSd, Swansea, 2022.

## Appendices

### Appendix 1: Euler Method of integration - Code

```
#include <iostream>
#include <fstream>

/*****
Copyright (c). Joshua Piecko. All rights reserved.

Original Author: Joshua Piecko
Date Written 15/11/2022

Date Last Modified: 16/11/2022

Version Comments:

*****/

class Car {
public:
    /*
        Declarations of Class Dependencies
    */
    // Dependencies required to calculate Drag Force, most unlikely to change
    // during simulation.
    float area; // m/^2
    float airDensity; // kg/m/^3
    float dragCoefficient;
    double dragForce; // Changing value - N, Newtons
    float gravity; // m/s/^2
    float mass; // kg
    float rollResistCoeff;

    // Independent and changing variables for the calculation of Velocity and
    // Position.
    double Acceleration = 0.00; // m/s
    double TractiveForce = 0.00; // N, Newtons

    // Current and Newly generated Velocities, 'V'.
    double Velocity; // m/s

    // Total distance Travelled, 'x'.
    double totalDistance; // m, meters

    /*
        Class Functions
    */

    // Function updates Tractive Force based on a conditional that estimates
    // whether the object is still within range of effect from the spring.
    void updateTForce() {

        if (totalDistance < 6) {
            // Tractive Force is calculated by 1.2(14-x) where 'x' is the
            // total distance travelled.
        }
    }
}
```

```

        TractiveForce = 1.2 * (14 - totalDistance);
    }
    else {
        TractiveForce = 0; // No additional force can be applied if Car
isn't touching spring.
    }
}

// Updates acceleration using the formula (Ft-Fd)/m.
void accelerationUpdate() {
    Acceleration = (TractiveForce - dragForce) / mass;
}

// Updates Velocity based on current acceleration multiplied by the
difference in time passed, dt.
void velocityUpdate(float deltaTick) {
    Velocity += (Acceleration * deltaTick);
}

// Calculates drag force, Fd, using a number of independent variables, using
formula - CRmg+1/2CdpAV^2
void dragForceUpdate() {
    double VelocitySquared = Velocity * Velocity; // Calculates Velocity
squared prior to equation to minimize risk of arithmetic overflow.
    dragForce = ((rollResistCoeff * mass * gravity) + ((dragCoefficient *
airDensity * area * VelocitySquared) / 2)); // Fd = CRmg + 1/2CdpAV^2
}

// Updates total distance travelled, x, using ODE x+(V*dt).
void distanceUpdate(float deltaTick) {
    totalDistance = totalDistance + (Velocity * deltaTick);
}

/*
Remaining functions below are used in the process of retrieving values from the
object that are used in the file-stream for csv export, so the data can be opened in
Excel.
*/
double getTForce() {
    return TractiveForce;
}
double getAcceleration() {
    return Acceleration;
}
double getDragForce() {
    return dragForce;
}
double getDistance() {
    return totalDistance;
}
double getVelocity() {
    return Velocity;
}
};

// Used for Clock, delta-Time - dt
float deltaTime = 0.00; // Total time passed.
float deltaTick = 0.06; // Value stored for deltaTime increment

int main() {

```

```

    // Initialization of Car object and all initial key dependencies for
    estimation.
    Car Car1;
    Car1.area = 0.01;
    Car1.airDensity = 1.2;
    Car1.dragCoefficient = 0.34;
    Car1.gravity = 9.81;
    Car1.mass = 1.0;
    Car1.rollResistCoeff = 0.21;

    // Initialized to only write to file.
    std::ofstream myFile;
    myFile.open("Euler.csv"); //output text file as .csv format.
    myFile << "Steps, Time (s), Position (x), Velocity (m/s), Acceleration (m/s/s)"
    << std::endl;

    while (deltaTime == 0 || Car1.getVelocity() >= 0) {
        // Calculates step number at the beginning of each cycle through loop.
        int stepNumber = deltaTime / deltaTick;

        // At deltaTime = 0, none of these values should be found to be of a
        value greater than 0 when dt is at initial value.
        if (deltaTime == 0) {
            Car1.Acceleration = 0;
            Car1.Velocity = 0;
            Car1.dragForce = 0;
            Car1.totalDistance = 0;
            Car1.updateTForce();
            int stepNumber = 0;
        }

        // Else if deltaTime > 0.00, calls functions on object to estimate new approximation
        for Velocity and Position to be written to file.
        else {
            Car1.updateTForce(); // Updates Tractive Force
            Car1.dragForceUpdate(); // Drag Force
            Car1.accelerationUpdate();
            Car1.velocityUpdate(deltaTick);
            Car1.distanceUpdate(deltaTick); // Position Update
        }

        // Only writes to File if the Velocity of the object is equal to or
        greater than 0, used to prevent recording approximation of distance travelled when
        receding.
        if (Car1.getVelocity() >= 0) {
            myFile << stepNumber << "," << deltaTime << "," <<
            Car1.getDistance() << "," << Car1.getVelocity() << "," << Car1.getAcceleration() <<
            std::endl;

            // Outputs data to terminal for comparison to hand-calculated
            values.
            std::cout << "Velocity: " << Car1.getVelocity() << " dT: " <<
            deltaTime << " distanceTravelled: " << Car1.getDistance() << " Acceleration: " <<
            Car1.getAcceleration() << " dragForce: " << Car1.getDragForce() << " Tractive Force: "
            << Car1.getTForce() << "\n";
        }

        deltaTime += deltaTick;
    }
    myFile.close();
    return 0;
}

```

## Appendix 2: Verlet Method of Integration – Code

```
#include <iostream>
#include <fstream>

/*****
 *
 *
 * This is a program that can approximate the velocity and position of a spring powered
 * dragster car using Verlet integration.
 *
 *
 * Copyright (c). Joshua Piecko. All rights reserved.

Original Author: Joshua Piecko

Date Written 15/11/2022

Date Last Modified: 16/11/2022

Version Comments:

*****/

// Used for Clock, delta-Time - dt
float deltaTime = 0.00; // Total time passed.
float deltaTick = 0.06; // Value stored for deltaTime increment

class Car {
public:
    /*
        Declarations of Class Dependencies

        */
        // Dependencies required to calculate Drag Force, most unlikely to change
        during simulation.
        float area = {}; // m/^2
        float airDensity = {}; // kg/m/^3
        float dragCoefficient = {};
        double dragForce = {}; // Changing value - N, Newtons
        float gravity = {}; // m/s/^2
        float mass = {}; // kg
        float rollResistCoeff = {};

        // Independent and changing variables for the calculation of Velocity and
        Position.
        double Acceleration = 0.00; // m/s
        double TractiveForce = 0.00; // N, Newtons

        // Current and Newly generated Velocities, 'V'.
        double Velocity = {}; // m/s
```

```

// Previous and new value totals for distance Travelled, 'x'.
double previousTotalDistance = {}; // m, meters
double totalDistance = {}; // m, meters

/*
Class Functions
*/

// Function updates Tractive Force based on a conditional that estimates
whether the object is still within range of effect from the spring.
void updateTForce() {
    if (totalDistance < 6) {
        // Tractive Force is calculated by  $1.2(14-x)$  where 'x' is the
total distance travelled.
        TractiveForce = 1.2 * (14 - totalDistance);
    }
    else {
        TractiveForce = 0; // No additional force can be applied if Car
isn't touching spring.
    }
}

// Updates acceleration using the formula  $(F_t - F_d)/m$ .
void accelerationUpdate() {
    Acceleration = (TractiveForce - dragForce) / mass;
}

// Calculates drag force,  $F_d$ , using a number of independent variables, using
formula -  $CRmg + 1/2CdpAV^2$ 
void dragForceUpdate() {
    double VelocitySquared = Velocity * Velocity; // Calculates Velocity
squared prior to equation to minimize risk of arithmetic overflow.
    dragForce = ((rollResistCoeff * mass * gravity) + ((dragCoefficient *
airDensity * area * VelocitySquared) / 2)); //  $F_d = CRmg + 1/2CdpAV^2$ 
}

// Updates total distance travelled, x, using ODE  $x+(V*dt)$ .
void distanceUpdate(float deltaTick) {
    double newTotalDistance = (2*totalDistance - previousTotalDistance +
Acceleration * (deltaTick*deltaTick));
    previousTotalDistance = totalDistance;
    totalDistance = newTotalDistance;
}

// Updates Velocity based on current acceleration multiplied by the
difference in time passed, dt.
void velocityUpdate(float deltaTick) {
    Velocity = ((totalDistance - previousTotalDistance) / deltaTick);
}

/*

```

Remaining functions below are used in the process of retrieving values from the object that are used in the file-stream for csv export, so the data can be opened in Excel.

```

    */
    double getTForce() {
        return TractiveForce;
    }
    double getAcceleration() {
        return Acceleration;
    }
    double getDragForce() {
        return dragForce;
    }
    double getDistance() {
        return totalDistance;
    }
    double getVelocity() {
        return Velocity;
    }
};

int main() {

    // Initialization of Car object and all initial key dependencies for
    estimation.
    Car Car1;
    Car1.area = 0.01;
    Car1.airDensity = 1.2;
    Car1.dragCoefficient = 0.34;
    Car1.gravity = 9.81;
    Car1.mass = 1.0;
    Car1.rollResistCoeff = 0.21;

    // Initialized to only write to file.
    std::ofstream myFile;
    myFile.open("Verlet.csv"); //output text file as .csv format.
    myFile << "Steps, Time (s), Position (x), Velocity (m/s), Acceleration (m/s/s)"
    << std::endl;

    while (deltaTime == 0 || Car1.getVelocity() >= 0) {
        // Calculates step number at the beginning of each cycle through loop.
        int stepNumber = deltaTime / deltaTick;

        // At deltaTime = 0, none of these values should be found to be of a
        value greater than 0 when dt is at initial value.
        if (deltaTime == 0) {
            Car1.Acceleration = 0;
            Car1.Velocity = 0;
            Car1.dragForce = 0;
            Car1.totalDistance = 0;
            Car1.updateTForce();
            int stepNumber = 0;
        }
        // Else if deltaTime > 0.00, calls functions on object to estimate new
        approximation for Velocity and Position to be written to file.
        else {
            Car1.updateTForce(); // Updates Tractive Force
            Car1.dragForceUpdate(); // Drag Force
            Car1.accelerationUpdate();
            Car1.distanceUpdate(deltaTick); // Position Update
            Car1.velocityUpdate(deltaTick);
        }
    }
}

```

```

    }

    // Only writes to File if the Velocity of the object is equal to or
    greater than 0, used to prevent recording approximation of distance travelled when
    receeding.
    if (Car1.getVelocity() >= 0) {
        myFile << stepNumber << "," << deltaTime << "," <<
Car1.getDistance() << "," << Car1.getVelocity() << "," << Car1.getAcceleration() <<
std::endl;

        // Outputs data to terminal for comparison to hand-calculated
values.
        std::cout << "Velocity: " << Car1.getVelocity() << " dT: " <<
deltaTime << " distanceTravelled: " << Car1.getDistance() << " Acceleration: " <<
Car1.getAcceleration() << " dragForce: " << Car1.getDragForce() << " Tractive Force: "
<< Car1.getTForce() << "\n";
    }

    deltaTime += deltaTick;
}
myFile.close();
return 0;
}

```

# ACCB5017 Assignment 1 - 2112389

## GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

71 /100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9



### Comment 1

units?



### Comment 2

Can't see blue Verlet

PAGE 10

PAGE 11

PAGE 12

GRADING FORM: SMU GENERIC\_GJ

JOSHUA PIECKO

70.50

#### RESEARCH-INFORMED LITERATURE



Some very good work here. A quite wide ranging and detailed review. Try to avoid too many electronic online refs. Why not use Bishop and van Verth?

7

#### KNOWLEDGE AND UNDERSTANDING



A very good understanding of the methods reviewed generated. Well done.

15.5

Relevant equations/algorithms presented and discussed - good.

## ANALYSIS



Comparisons of methods presented together with advantages and disadvantages - well done. The graphs presented were analysed too. Well done.

**14**

The comparison between methods discussion was well structured but the graphs showed little/no difference between methods - some graph plots appeared to be missing a curve - perhaps overdrawn by another? If so then this problem needs solving.

## PRACTICAL APPLICATION AND DEP.



Euler and Verlet applied Units included in comments- generally comments were helpful - good. Some evidence of good SE practices here.

**30.25**

Some further consideration of Verlet and Euler would help here. Explore why results are so similar for this problem.

## SKILLS FOR PROFESSIONAL PRAC.



Table of comments missing. Reference original source for figures used in the figure description.

**3.75**

Well written in general. Well structured and well done!