

Report

by Mariusz Kalasz

FILE	12027_MARIUSZ_KALASZ_REPORT_234537_862168723.DOCX (2.26M)		
TIME SUBMITTED	07-FEB-2020 02:11PM (UTC+0000)	WORD COUNT	3387
SUBMISSION ID	119753824	CHARACTER COUNT	16705

2020

Part 1a: Games Physics

LECTURER: GORDON DICKERS
MARIUSZ KALASZ (1805456)

MODULE CODE | ACCA5017

Contents

Task 1: Euler's method and other integration methods	2
Task 2: Testing of Vector calculations and descriptions of operations used	8
Task 3: Explanation of 2 spheres colliding	13
References	17

Task 1: Euler's method and other integration methods

In this section Euler the main method used for calculating the dragster car simulation will be described in detail and other common integration methods which are commonly used in gaming.

First, integration methods fall in different categories depending on their level accuracy. So, the categories are; first order, second order and higher order integration, where first order is the least accurate and higher order are the most accurate the way it a method can be checked to what order they belong in is checking the amount of steps which are taken to reach the derivative [1].

Euler Integration: Explicit Euler

Euler is a first order integration method, so it uses only one step to reach the derivative. Euler has three different methods which can implemented, they are; Explicit, Implicit (also known as backwards Euler), and Semi-Implicit (also known as Symplectic Euler) [1].

Explicit Euler method is the one used in code, this is where the method takes the initial position and velocity to calculate the next position and velocity after a given time, this is repeated using a time step value, for example 0.6. When the first values are calculated those values are used to calculate the next set of values and so on.

A simple equation to find velocity would be:

$$v_{n+1} = v_n + (a_n * t)$$

Where, v is velocity, a is acceleration and t is time. And n is the step number.

From the velocity equation the position can be found:

$$x_{n+1} = x_n + (v_n * t)$$

Where x is the position, v is the velocity, and t is the time.

However, this method is not the most accurate as it will always have an error value depending on the step size value, so in case of the example it would be 0.6s. The error value is the difference between the actual value and the estimation, the problem is the more steps the method has the more the error value will stack up which there will be a practical limit to this method as the error value will make the values highly inaccurate [1]. This can be decreased by using smaller time steps like 0.01, this would decrease the error value and make the values more accurate however this will take more processing power as it would have to do more calculations.

Explicit Euler is a simple integration method which is easy and quick to code, however the accuracy of it is lacking unless the step sizes have been decreased. This method is useful when the impact of the error value does not affect any actions which will happen next, for example "a sprinting character jumping the as the character lands it continues to sprint" [1].

The other Euler methods can be used to improve the accuracy of the values.

Implicit Euler

Implicit Euler works on the basis of calculating the derivative and then jumping back to use that derivative to calculate the next estimation [1].

So, to calculate the velocity a simple equation of it would be:

$$v_{n+1} = v_n + (a_{n+1} * t)$$

And when the velocity was calculated the position would be calculated using this:

$$x_{n+1} = x_n + (v_{n+1} * t)$$

This different technique of the Euler method still has an error value, but it is stable, even with larger step values. However, it is more complex to code than the Explicit method as it needs to calculate the updated acceleration and velocity.

This method is not ideal for any rigid body movements, this is due to it takes a good amount of processing power, so it takes longer to process. This method is generally used for items in game, for example springs, calculating friction, drag and changes in acceleration [1].

Semi-Implicit Euler

The last technique of the Euler method is the Semi-Implicit which is the middle ground between the Explicit and Implicit Euler techniques. The reason it is referred to as the middle ground as it calculates the velocity and position at a specific time. It calculates the velocity using the Explicit technique which is using the acceleration of the previous time step:

$$v_{n+1} = v_n + (a_n * t)$$

But the next step in the process uses the Implicit technique which is calculating the position using the newly updated velocity value:

$$x_{n+1} = x_n + (v_{n+1} * t)$$

One of the advantages of using this method is that its more stable than the explicit, so it provides an estimation which is closer to than the actual value, however it does still suffer from error values. Another advantage of it is that it is easier to implement than any of the higher order methods and it is more accurate compared to the explicit technique [1].

This method is used to calculate rigid body movement.

Verlet Method: Position Verlet

The Verlet method is under the second order integration, this makes more accurate compared to Euler. The method of Verlet works on the basis that the position is calculated using acceleration which different compared to Euler method as it calculates velocity from the acceleration first then the position using that velocity. This is how the position would be calculated using Verlet:

$$x_{n+1} = 2 * x_n - x_{n-1} + a * t^2$$

Where x is the position, a is the acceleration, and t is time.

And if needed the velocity would be calculated using:

$$v_{n+1} = (x_n - x_{n-1})/t$$

The advantages of using this method is that it is more accurate compared to Euler, and it still maintains the simplicity of the implementation and in addition it is more efficient as it does not take as much time to process than higher order integrators [1]. However, it does have its disadvantages as well, one is that the method assumes the time step will always be a constant increase therefore the method will not be accurate if the time step varies during the same process. Another disadvantage is that the velocity calculation is just an approximation therefore if the program required velocity which changes a lot it will be very inaccurate, this would also affect if there was a need to calculate instantaneous velocity changes that may happen after a collision [1].

So, in summary, this method is accurate and efficient whilst keeping its simplicity, however it is only able to find an approximation for velocity. This method is mainly used to simulate projectile trajectories and particle systems, but it can also be used to simulate cloth and ragdoll effects.

Velocity Verlet

Velocity Verlet is a solution to the approximated velocity values. This method splits the process into four steps where the velocity will be calculated twice, once at half time step and the other after the next acceleration is calculated. So, for example it would look like this:

Step 1

$$v_{n+\frac{1}{2}} = v_n + a_n * t/2$$

Step 2

$$x_{n+1} = x_n + v_{n+\frac{1}{2}} * t$$

Step 3

$$a_{n+1} = F/m$$

Step 4

$$v_{n+1} = v_{n+\frac{1}{2}} + a_{n+1} * t/2$$

However, implementing this method will lose efficiency, but this does allow the Verlet method to be used to simulate physics where the velocity increases and decreases rapidly for example a car racing game. In addition, this method can be used in situation where Position Verlet would have been used and this will make the situations more accurate [1].

Algorithm Used

The algorithm which was implemented in the code used the Explicit Euler method.

Euler.cpp file

```
void EulerCarSim::CarSimulation(float cR, float m, float g, float cD, float p, float area)
{
    float v = 0, fd = 0, ft = 0, a = 0, x = 0, t = 0;
    // v= velocity, fd= drag force, ft= tractive force, a= acceleration, x= distance, t= time
    while (v > 0 || t < 0.07)
    {
        cout << t << " " << v << " " << x << " " << a << endl;
        ft = 1.5*(16 - x);
        fd = (cR*m*g) + (0.5*cD*p*area*pow(v, 2));
        x += v * t;
        if (x > 8)
        {
            ft = 0;
        }
        a = (ft - fd) / m;
        v += a * t;
        if (v < 0)
        {
            fd = 0;
        }
        t += 0.06;
    }
}
```

Euler.h file

```
class EulerCarSim
{
private:
    float cR = 0.21, m = 1, g = 9.81, cD = 0.34, p = 1.2, area = 0.01;
    //cR= coefficient of rolling resistance, m= mass of vehicle, g= gravity, cD= drag coefficient, p= air density, area= frontal area of vehicle
public:
    void CarSimulation(float cR, float m, float g, float cD, float p, float a);

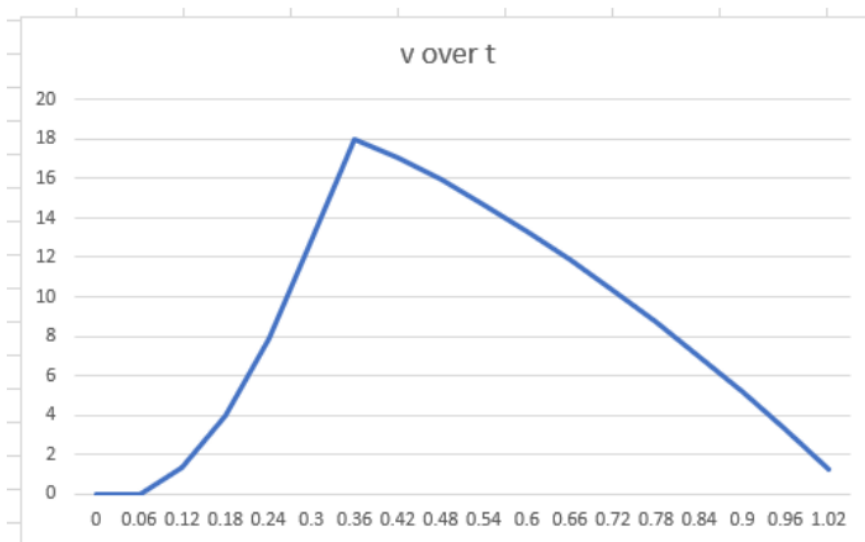
    EulerCarSim()
    {
        CarSimulation(cR, m, g, cD, p, area);
    }
};
```

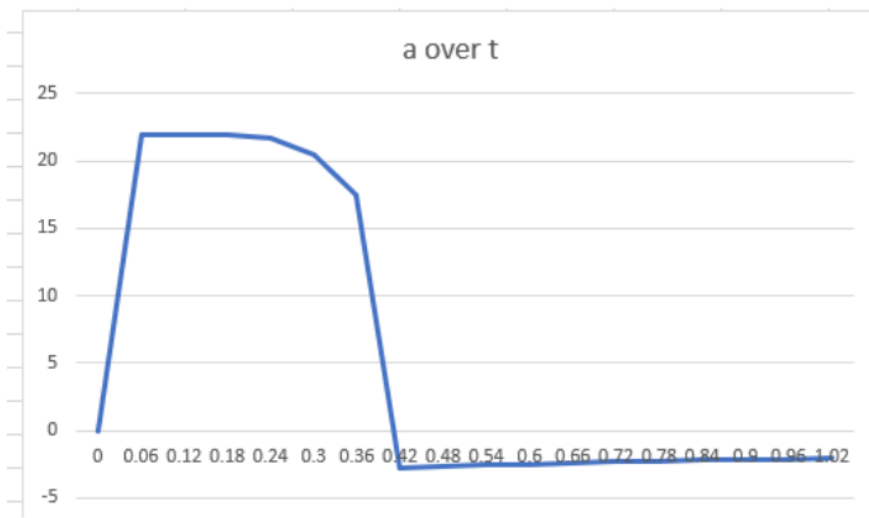
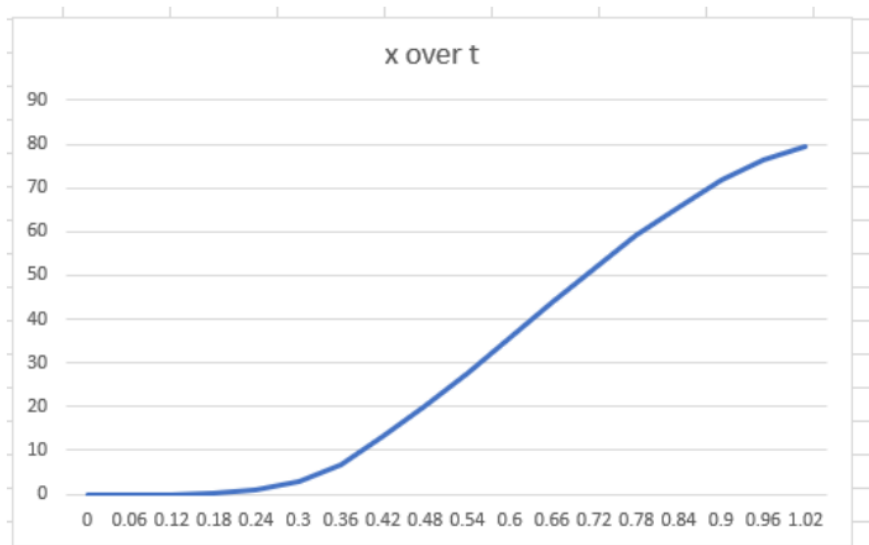
So, the algorithm works on the basis using the current velocity and position to calculate the next estimation. The calculation is inside the while loop which will only break if the velocity is lower than zero or when t is lower than 0.07s. Then the rest of the code serves the purpose which was described in the brief with the checks which set the tractive force to 0 when the distance is greater than 8 and the drag force is set to 0 when the velocity is lower than 0.

The results of this algorithm are:

1			
0	0	0	0
0.06	0	0	21.9399
0.12	1.31639	0	21.9399
0.18	3.94876	0.157967	21.9364
0.24	7.84956	0.868744	21.6711
0.3	12.7722	2.75264	20.5111
0.36	18.0157	6.58431	17.4782
0.42	17.0357	13.0699	-2.72221
0.48	15.9218	20.2249	-2.65214
0.54	14.6847	27.8674	-2.57725
0.6	13.3347	35.7971	-2.50001
0.66	11.881	43.7979	-2.42284
0.72	10.3313	51.6394	-2.34806
0.78	8.69123	59.0779	-2.27784
0.84	6.96415	65.8571	-2.2142
0.9	5.15056	71.707	-2.15904
0.96	3.24777	76.3425	-2.11422
1.02	1.24941	79.4603	-2.08162

First column is time, second velocity, third is distance and fourth is the acceleration of the dragster car. These number then have been transferred to Excel to give graphs:





Task 2: Testing of Vector calculations and descriptions of operations used

In this section the distance between two points and the plane equation functions will be tested and each operation will be explained which are used in those functions.

Distance between two points

This function uses the normal equation for distance between two points, however the code uses 2 variations of it one which requires two vectors to calculate the distance between those points in 3D space and the other uses one vector to calculate the distance, however the latter one is only used in the plane equation function for normalisation.

So, the basic equation for distance between two points is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

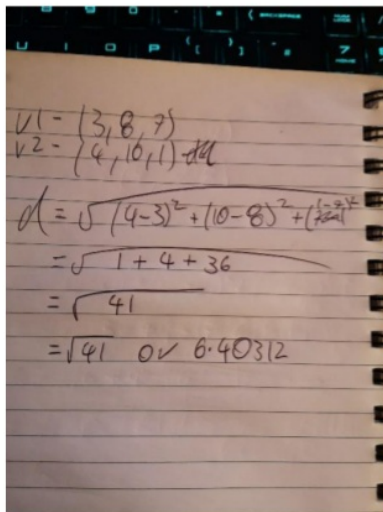
[2]

However, this equation is used in only 2D space, to make it into 3D space the z axis is added to the end so it will look like this:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

The equation works on first choosing the two points where the distance is needed, and it is the difference between the two points' axis, then the difference is squared which then is summed up with other differences. The distance is the square root of the sum of the squared differences in each axis [2].

For example:



A photograph of a piece of lined paper with handwritten calculations. At the top, two points are listed: $V1 = (3, 8, 7)$ and $V2 = (4, 10, 1)$. Below these, the distance formula is written: $d = \sqrt{(4-3)^2 + (10-8)^2 + (1-7)^2}$. The calculation is then simplified step-by-step: $= \sqrt{1 + 4 + 36}$, then $= \sqrt{41}$, and finally the decimal result $= \sqrt{41} \approx 6.40312$ is written.

These numbers will also be used as test data.

Test Number	Description	Expected result	Actual result	Pass/Fail
1	In this test the test data used in the example above will be used in the calculation.	The expected result is 6.40312.	<pre> 2 please enter v1.x: 3 please enter v1.y: 8 please enter v1.z: 7 please enter v2.x: 4 please enter v2.y: 10 please enter v2.z: 1 the distance: 6.40312 </pre>	Pass

Plane Equation

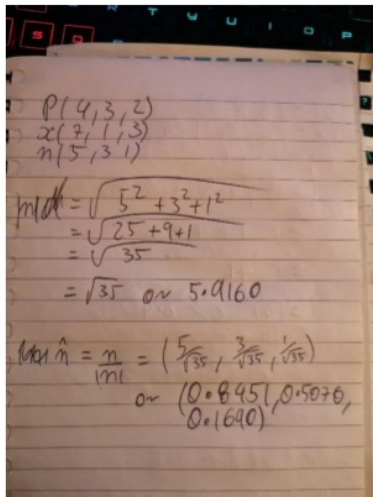
This function is more complex as it uses different functions to get the answer, which are the distance between two points and the dot product. The equation uses 3 points; n which is the normal to the plane, P which is a point on the plane and x which is point somewhere around the plane.

Distance between two points have already been explained. However, it uses the other method which was mentioned previously:

$$|n| = \sqrt{(x)^2 + (y)^2 + (z)^2}$$

This method only uses one vector and it is used in the process of normalisation which is dividing a point by its scalar value this will result in a unit vector. To complete normalisation firstly the distance needs to be calculated which was described previously. The method uses the same principles as distance between two points.

For example:



$P(4,3,2)$
 $x(2,1,3)$
 $n(5,3,1)$
 $|n| = \sqrt{5^2 + 3^2 + 1^2}$
 $= \sqrt{25 + 9 + 1}$
 $= \sqrt{35}$
 $= \sqrt{35} \text{ or } 5.9160$
 $\text{Unit } \hat{n} = \frac{n}{|n|} = \left(\frac{5}{\sqrt{35}}, \frac{3}{\sqrt{35}}, \frac{1}{\sqrt{35}} \right)$
 $\text{or } (0.8451, 0.5070, 0.1690)$

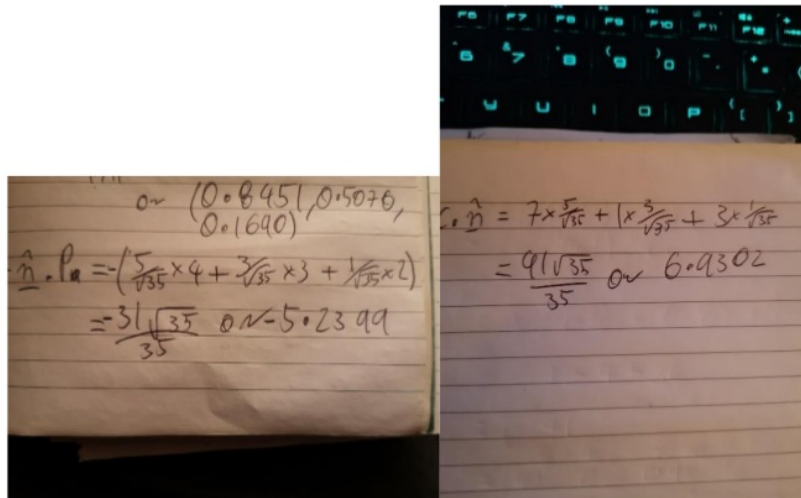
The next step is using the unit vector is to calculate the dot product of $-\hat{n} \cdot P$ and $\hat{n} \cdot x$.

The dot product is the sum of products of the corresponding entries in a sequence of numbers [3]. An example of the formula:

$$u \cdot v = u_1 * v_1 + u_2 * v_2 + u_3 * v_3$$

[4]

Example of this:



Handwritten calculations showing the dot product of two vectors. The first vector is $\vec{n} = (0.8451, 0.5076, 0.1640)$ and the second vector is $\vec{p} = (5, 3, 2)$. The calculation is as follows:

$$\vec{n} \cdot \vec{p} = (0.8451 \times 5) + (0.5076 \times 3) + (0.1640 \times 2)$$

$$= 4.2255 + 1.5228 + 0.328$$

$$= 6.0763$$

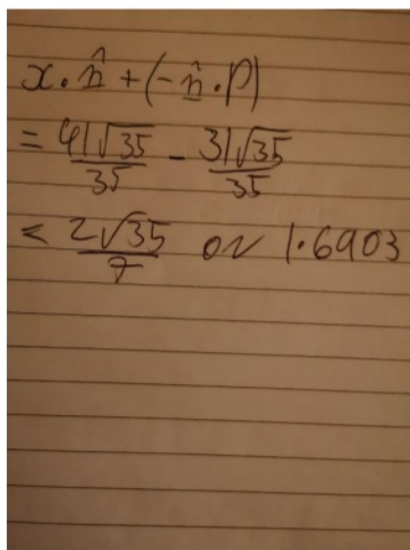
These now can be used in the final equation, which is the planar equation itself, which is:

$$x \cdot n + d = 0$$

But rearranged will give the d value which is the distance between the point x and the plane.

$$\vec{n} \cdot x - \vec{n} \cdot P$$

So, the final step is:



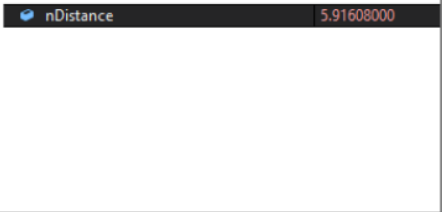
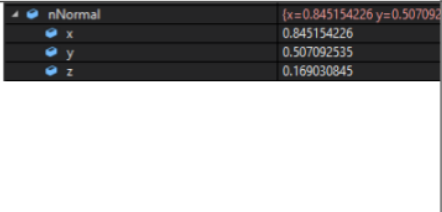
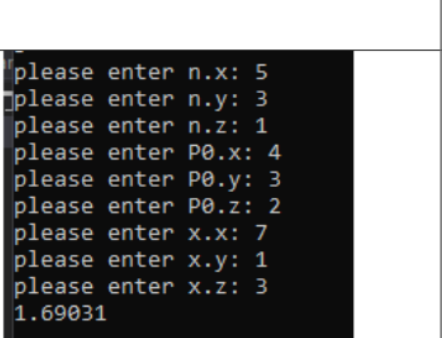
Handwritten calculation showing the final step of the planar equation. The calculation is as follows:

$$x \cdot \vec{n} + (-\vec{n} \cdot P)$$

$$= \frac{41\sqrt{35}}{35} - \frac{31\sqrt{35}}{35}$$

$$= \frac{10\sqrt{35}}{35} \text{ or } 1.6903$$

The data use in the handwritten examples will be used to test the function.

Test number	Description	Expected results	Actual results	Pass/Fail
1	This test will check if the scalar value is calculated correctly	It should be 5.9160		Pass
2	This test will check if the normalisation of n is calculated correctly	It should be (0.8451,0.5070,0.19690)		Pass
3	This test will check if the dot product for $-\hat{n} \cdot P$ is correctly calculated	It should be -5.2399		Pass
4	This test will check if the dot product of $\hat{n} \cdot x$ is correctly calculated	It should be 6.9302		Pass
5	This test will check if the final answer is correctly calculated	It should be 1.6903		Pass

The whole reason of why using these methods as they are the best choice to calculate the distance between two points and the plane equation. Dot product and distance between two points are try and tested methods which give the right values also this author has used both multiple of times.

The algorithms

Both distance between two points and the dot product functions

```
float Vector::ptpDistance2V(Vector & v1, Vector & v2)
{
    float distance;
    distance = sqrt(pow((v2.x - v1.x), 2) + pow((v2.y - v1.y), 2) + pow((v2.z - v1.z), 2));

    return distance;
}

float Vector::ptpDistance(Vector & v1)
{
    float distance;
    distance = sqrt(pow((v1.x), 2) + pow((v1.y), 2) + pow((v1.z), 2));

    return distance;
}

float Vector::dotProduct(Vector normal, Vector & v1)
{
    float dot;
    dot = ((normal.x*v1.x) + (normal.y*v1.y) + (normal.z*v1.z));
    return dot;
}
```

Plane.cpp file

```
#include "plane.h"
#include <cmath>
// #include <math.h>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

void Plane::planeEqn(Vector &x)
{
    float nDistance = ptpDistance(n);
    Vector nNormal;
    nNormal.x = n.x / nDistance;
    nNormal.y = n.y / nDistance;
    nNormal.z = n.z / nDistance;

    float nPo = dotProduct(nNormal, p)*-1;
    float nx = dotProduct(nNormal, x);

    float distanceFromPlane = nx + nPo;

    cout << distanceFromPlane<<endl;
}
```

Plane.h file

```
#pragma once
#include "Vector.h"

class Plane : Vector
{
public:
    Vector p;
    Vector n;
    void planeEqn(Vector &x);
};
```

The whole theory of how the calculations work has been explained previously, so what the user needs to do is to insert their values via the UI and the program does the rest. There are probably more efficient ways of making this work as the application itself lags a bit when its loading up, so the memory usage is relatively high.

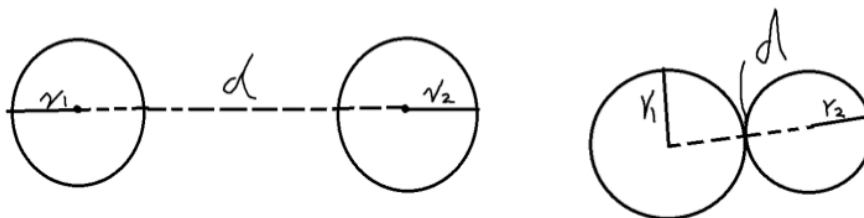
Task 3: Explanation of 2 spheres colliding

In this section the collision of two sphere will be discussed, mainly the maths behind the how the solution works in the application. Following tests for the function as well.

The current solution

The solution currently works on the basis of compering 2 values and see if one is larger than the other. So, firstly the distance between the two centres of the spheres is calculated, the calculations are the same as the previously mentioned. Then the sum of both radii is calculated which then is checked if the distance is greater to the sum of the radii. If it is the spheres have not collided if the distance is shorter or equal to the sum of the radii they have collided.

Diagrams:



Examples of how it works:

test for sphere collision

st sphere 1: (3, 8, 7)
 $r=6$

st sphere 2: (4, 1, 5)
 $r=3$

DTP

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

$$= \sqrt{(4 - 3)^2 + (1 - 8)^2 + (5 - 7)^2}$$

$$= \sqrt{1 + 49 + 4}$$

$$= \sqrt{54}$$

$$= 7.348$$

Rad Sum is $d > \text{rad sum}$
 $6 + 3 = 9$ no collision

sphere 1: (4, 8, 2)
 $r=3$

sphere 2: (3, 7, 8)
 $r=4$

DTP

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

$$= \sqrt{(3 - 4)^2 + (7 - 8)^2 + (8 - 2)^2}$$

$$= \sqrt{1 + 1 + 36}$$

$$= \sqrt{38}$$

$$= 6.164$$

Sum Rad
 $3 + 4 = 7$

Is distance $>$ Sum Rad?
 Yes $\therefore$ no collision

sphere 1: (8, 5, 12)
 $r=4$

sphere 2: (4, 3, 8)
 $r=2$

DTP

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

$$= \sqrt{(4 - 8)^2 + (3 - 5)^2 + (8 - 12)^2}$$

$$= \sqrt{16 + 4 + 16}$$

$$= \sqrt{36}$$

$$= 6$$

Sum Rad
 $4 + 2 = 6$

Is $d >$ Sum Rad?
 No $\therefore$ They have collided
 as they are equal

The data in the examples will be used as test data.

Test number	Description	Expected results	Actual results	Pass/Fail
1	To check if the function recognises a collision	"The spheres have collided"	<pre> please enter the x of first sphere: 3 please enter the y of the first sphere: 8 please enter the z of the first sphere: 7 please enter the radius of sphere 1: 6 please enter the x of first sphere: 4 please enter the y of the first sphere: 1 please enter the z of the first sphere: 5 please enter the radius of sphere 1: 3 spheres have collided </pre>	Pass
2	To check if the function recognises no collision	"The spheres have not collided"	<pre> please enter the x of first sphere: 4 please enter the y of the first sphere: 1 please enter the z of the first sphere: 2 please enter the radius of sphere 1: 3 please enter the x of first sphere: 3 please enter the y of the first sphere: 7 please enter the z of the first sphere: 8 please enter the radius of sphere 1: 4 spheres have not collided </pre>	Pass
3	To check if the function recognises a collision but when sum of radii and distance is equal	"The spheres have collided"	<pre> please enter the x of first sphere: 8 please enter the y of the first sphere: 5 please enter the z of the first sphere: 12 please enter the radius of sphere 1: 4 please enter the x of first sphere: 4 please enter the y of the first sphere: 3 please enter the z of the first sphere: 8 please enter the radius of sphere 1: 2 spheres have collided </pre>	Pass

The algorithm

Sphere.h File

```

#pragma once
#include "Vector.h"
class Sphere : Vector
{
public:
    float radius;
    Vector sphereCenter;
    void sphereCollision(Sphere sphere);

    Sphere(float rad, float x, float y, float z);
};

```

Sphere.cpp file

```
#include "sphere.h"
#include "plane.h"
#include <cmath>
// #include <math.h>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;
Sphere::Sphere(float rad, float x, float y, float z)
{
    radius = rad;
    sphereCenter = Vector(x, y, z);
}

void Sphere::sphereCollision(Sphere sphere)
{
    float distance = sphereCenter.ptpDistance2V(sphereCenter, sphere.sphereCenter);
    float sumRad = radius + sphere.radius;

    if (distance > sumRad)
    {
        cout << "spheres have not collided" << endl;
    }
    else
    {
        cout << "spheres have collided" << endl;
    }
}
```

References

- [1] jdicksongames, “Understanding Games Development,” jdicksongames, 22 January 2015. [Online]. Available: <https://jdickinsongames.wordpress.com/2015/01/22/numerical-integration-in-games-development-2/>. [Accessed 7 February 2020].
- [2] MathBlog, “The Distance Between Two Points,” MathBlog, [Online]. Available: <https://mathblog.com/reference/arithmetic/distance-two-points/>. [Accessed 7 February 2020].
- [3] Wikipedia, “Dot product,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Dot_product. [Accessed 7 February 2020].
- [4] math.la.asu.edu, “The Dot and Cross Product,” [Online]. Available: <https://math.la.asu.edu/~surgent/mat272/dotcross.pdf>. [Accessed 7 February 2020].

Appendix

Euler.h file

```
#pragma once

class EulerCarSim
{
private:
    float cR = 0.21, m = 1, g = 9.81, cD = 0.34, p = 1.2, area = 0.01;
    //cR= coefficient of rolling resistance, m= mass of vehicle, g= gravity, cD=
    drag coefficient, p= air density, area= frontal area of vehicle
public:
    void CarSimulation(float cR, float m, float g, float cD, float p, float a);

    EulerCarSim()
    {
        CarSimulation(cR, m, g, cD, p, area);
    }
};
```

Plane.h file

```
#pragma once
#include "Vector.h"

class Plane : Vector
{
public:
    Vector p;
    Vector n;
    void planeEqn(Vector &x);
};
```

Sphere.h file

```
#pragma once
#include "Vector.h"
class Sphere : Vector
{
public:
    float radius;
    Vector sphereCenter;
    void sphereCollision(Sphere sphere);

    Sphere(float rad, float x, float y, float z);
};
```

Vector.h file

```
#pragma once
#ifndef VECTOR_H
#define VECTOR_H

class Vector
{
public:
    //construct
    Vector(float x, float y, float z);
    Vector(void);
    void Scale(float s);
    void Add(Vector & v1);
    void Add(Vector & v1, Vector & v2);
    Vector getVector(void);
    //private:    //these are currently public - should they be private?
    float x;
    float y;
    float z;

    float ptpDistance2V(Vector & v1, Vector & v2);
    float ptpDistance(Vector & v1);
    //void planeEqn(Vector & v1, Vector & v2, Vector & v3);
    float dotProduct(Vector normal, Vector & v1);

};
#endif
```

Euler.cpp file

```
#include "Euler.h"
#include <iostream>
#include <string>
#include <sstream>
#include <cmath>

using namespace std;

void EulerCarSim::CarSimulation(float cR, float m, float g, float cD, float p, float
area)
{
    float v = 0, fd = 0, ft = 0, a = 0, x = 0, t = 0;
    // v= velocity, fd= drag force, ft= tractive force, a= acceleration, x=
distance, t= time
    while (v > 0 || t < 0.07)
    {
```

```

        cout << t << "          " << v << "    " << x << "    " << a << endl;
        ft = 1.5*(16 - x);
        fd = (cR*m*g) + (0.5*cD*p*area*pow(v, 2));
        x += v * t;
        if (x > 8)
        {
            ft = 0;
        }
        a = (ft - fd) / m;
        v += a * t;
        if (v < 0)
        {
            fd = 0;
        }
        t += 0.06;
    }
}

```

Plane.cpp file

```

#include "plane.h"
#include <cmath>
// #include <math.h>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

void Plane::planeEqn(Vector &x)
{
    float nDistance = ptpDistance(n);
    Vector nNormal;
    nNormal.x = n.x / nDistance;
    nNormal.y = n.y / nDistance;
    nNormal.z = n.z / nDistance;

    float nPo = dotProduct(nNormal, p)*-1;
    float nx = dotProduct(nNormal, x);

    float distanceFromPlane = nx + nPo;

    cout << distanceFromPlane<<endl;
}

```

Source.cpp file

```

#include "vector.h"
#include <cmath>
// #include <math.h>
#include <iostream>
#include <string>
#include <sstream>
#include "plane.h"
#include "sphere.h"
#include "Euler.h"

using namespace std;

```

```

void PlaneRun()
{
    Plane plane;
    Vector x;
    cout << "please enter n.x: ";
    cin >> plane.n.x;
    cout << "please enter n.y: ";
    cin >> plane.n.y;
    cout << "please enter n.z: ";
    cin >> plane.n.z;
    cout << "please enter P0.x: ";
    cin >> plane.p.x;
    cout << "please enter P0.y: ";
    cin >> plane.p.y;
    cout << "please enter P0.z: ";
    cin >> plane.p.z;
    cout << "please enter x.x: ";
    cin >> x.x;
    cout << "please enter x.y: ";
    cin >> x.y;
    cout << "please enter x.z: ";
    cin >> x.z;

    plane.planeEqn(x);
}

void DistanceBetween2Points()
{
    Vector vector;
    Vector v1;
    Vector v2;

    cout << "please enter v1.x: ";
    cin >> v1.x;
    cout << "please enter v1.y: ";
    cin >> v1.y;
    cout << "please enter v1.z: ";
    cin >> v1.z;

    cout << "please enter v2.x: ";
    cin >> v2.x;
    cout << "please enter v2.y: ";
    cin >> v2.y;
    cout << "please enter v2.z: ";
    cin >> v2.z;

    cout << "the distance: " << vector.ptpDistance2V(v1, v2) << endl;;
}

void SphereCollisionRun()
{
    float xS, yS, zS, r;

    cout << "please enter the x of first sphere: ";
    cin >> xS;
    cout << "please enter the y of the first sphere: ";
    cin >> yS;
    cout << "please enter the z of the first sphere: ";
    cin >> zS;
    cout << "please enter the radius of sphere 1: ";

```

```

        cin >> r;

        Sphere sphere1(r, xS, yS, zS);

        cout << "please enter the x of first sphere: ";
        cin >> xS;
        cout << "please enter the y of the first sphere: ";
        cin >> yS;
        cout << "please enter the z of the first sphere: ";
        cin >> zS;
        cout << "please enter the radius of sphere 1: ";
        cin >> r;

        Sphere sphere2(r, xS, yS, zS);

        sphere1.sphereCollision(sphere2);
    }

    int main()
    {
        bool exit = false;
        while (exit != true)
        {
            int choice = 0;

            cout << "please select a function" << endl;
            cout << "1. Euler Car Simulation" << endl;
            cout << "2. Distance between two points" << endl;
            cout << "3. Plane Equation" << endl;
            cout << "4. Sphere Collision calculation" << endl;
            cout << "5. Exit" << endl;
            cin >> choice;

            switch (choice)
            {
                case 1:
                    EulerCarSim();
                    break;
                case 2:
                    DistanceBetween2Points();
                    break;
                case 3:
                    PlaneRun();
                    break;
                case 4:
                    SphereCollisionRun();
                    break;
                case 5:
                    exit = true;
                    break;
                default:
                    cout << "please enter valid input" << endl;
            }
        }
        cin.get();
        return 0;
    }

```

Sphere.cpp file

```
#include "sphere.h"
#include "plane.h"
#include <cmath>
// #include <math.h>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;
Sphere::Sphere(float rad, float x, float y, float z)
{
    radius = rad;
    sphereCenter = Vector(x, y, z);
}

void Sphere::sphereCollision(Sphere sphere)
{
    float distance = sphereCenter.ptpDistance2V(sphereCenter, sphere.sphereCenter);
    float sumRad = radius + sphere.radius;

    if (distance > sumRad)
    {
        cout << "spheres have not collided" << endl;
    }
    else
    {
        cout << "spheres have collided" << endl;
    }
}
```

Vector.cpp file

```
#include "vector.h"
#include <cmath>
// #include <math.h>
#include <iostream>
#include <string>
#include <sstream>

//construct
using namespace std;
Vector::Vector(void) {};
Vector::Vector(float myx, float myy, float myz)
{
    x = myx;
    y = myy;
    z = myz;
}

void Vector::Scale(float s)
{
    x = s * x;
    y = s * y;
    z = s * z;
}

void Vector::Add(Vector & v1, Vector & v2)
{
    v1.x = v2.x + v1.x;
    v1.y = v2.y + v1.y;
    v1.z = v2.z + v1.z;
}
```

```

}
Vector Vector::getVector(void)
{
    return Vector(x, y, z);
}
void Vector::Add(Vector & v1)
{
    x = x + v1.x;
    y = y + v1.y;
    z = z + v1.z;
}

float Vector::ptpDistance2V(Vector & v1, Vector & v2)
{
    float distance;
    distance = sqrt(pow((v2.x - v1.x), 2) + pow((v2.y - v1.y), 2) + pow((v2.z -
v1.z), 2));

    return distance;
}

float Vector::ptpDistance(Vector & v1)
{
    float distance;
    distance = sqrt(pow((v1.x), 2) + pow((v1.y), 2) + pow((v1.z), 2));

    return distance;
}

float Vector::dotProduct(Vector normal, Vector & v1)
{
    float dot;
    dot = ((normal.x*v1.x) + (normal.y*v1.y) + (normal.z*v1.z));
    return dot;
}

```

Report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

73/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

GRADING FORM: SMU GENERIC_GJ

MARIUSZ
KALASZ

72.50

RESEARCH-INFORMED LITERATURE



Topics covered in detail. Well done. Refs included but overuse of online refs. Try Bishop and van Verth, etc.

6.5

KNOWLEDGE AND UNDERSTANDING



A very good, detailed report. Implementation uses some good SE practices.

16

ANALYSIS



A very good analysis here but you could have explained more clearly how the results are related to the physical dragster setup.

15

PRACTICAL APPLICATION AND DEP.



Some good work here. Some attributes remain public - not private. Try using accessor and mutator methods. Testing documented

31.75

SKILLS FOR PROFESSIONAL PRAC.



Report detailed and well structured. Ref used but are on-line. Avoid over-reliance on on-line refs.

3.25