Physics engine

Loi Yi Yang Caden

29 June 2023 - 31 December 2023

Contents

1 Purpose **3**

2 How a physics engine works **3**
2.1 Time step 3
2.2 Force generator 3
2.3 Ordinary differential equation solver(ODES) 4
2.3.1 Euler forward differencing method 4
2.3.2 Runge Kutta Fourth order method(RK4) 4

3 Example simulations **5**
3.1 Terminal velocity of a ball 5
3.2 Thrown ball 6
3.3 Single pendulum 8
3.4 Double pendulum 12

4 Conclusion **16**

5 Reflection **16**

6 References **16**

1 Purpose

Physics engines are most commonly used for simulations of physics in games. However, there are also other purposes of a physics engine such as simulating systems that are hard or cannot be represented by an equation. This are systems like chaotic systems or systems where equation of acceleration cannot be integrated thus making it impossible to find the equation of velocities and displacements of a system. A physics engine, allows one to create a approximate of the velocities and displacement which can be represented in a graph or simply shown on a Graphical interface.

2 How a physics engine works

Physics engines have 3 essential components, a time step, a force generator and a ordinary differential equation solver(ODES).

2.1 Time step

A time step is one of the most important components of a physics engine as physics engine work by calculating the forces, velocities and displacements at every time step. The time step is the amount of time you "jump" in one interation, this means that you run the simulation after every time step. This means that the time step determinds how many iteration your programmes goes through and thus how long it takes to run, but at the same time also determinds the accuracy of the results.

2.2 Force generator

A force generator is a function that calculates the resultant force acting on an object at the time it is at. It does so by inputing known values such as the previous accleration, velocity, displacement of the object into equations which calculates what forces should be acting on it.

This can be done by 2 major methods, the equation based method and the constraint based method.

The equation based method mostly gives you a more accurate calculation and is typically faster as it has a smaller time complexity. It is also simplier to implement as it does not usualy require multiple functions. However

this method is restricted due to the need for the equation to be manually calculated and thus makes it difficult to have very complex simulations with multiple objects.

The constraint based method uses the concept of constraints. For example a rigid body in 2D has 3 degrees of freedom, 2 positional and 1 rotational. A constraint decreases the degrees of freedom of an object. For instance, a constraint that pins an object in space at its center of mass decreases the object's degrees of freedom by 2 as all the positional degrees of freedom are removed and the object can thus now only rotate with 1 degrees of freedom.

In a constraint based physics engine, everything is modled as a constraint including collision contacts, frictions, springs, pulleys. This works by using constrains to solve interactions between 2 or more bodies.

2.3 Ordinary differential equation solver(ODES)

A ODES is a fucntion that calculates the value of the integrated equation at a time step using an approximation to allow you to find an approximation of the velocity and displacement of an object at each time step. This can be done by 2 main methods. Euler forward differencing which is a simpler but more unstable method, and Runge Kutta which has a fourth order method.

2.3.1 Euler forward differencing method

Euler forward differencing method works simply by taylor expanding the ordinary differencial equation (ODE) and then multipling it by the time step (Marenduzzo). However, it's accuracy is also directly correlated to how small the time step is. This forces the timestep to be very small for you to get an accurate result, this slows down the simulation and makes it more complex.

2.3.2 Runge Kutta Fourth order method(RK4)

RK4 uses 4 estimates of the derivatives and takes the a weighted average of these derivatives. The four derivatives are given by the below equations where t is the total time, y is a inital condition that is to be updated and h is the time step.

$$k_1 = f(t, y)$$

$$k_2 = f(t + \frac{h}{2}, y + h\frac{k_1}{2}$$

$$k_3 = f(t + \frac{h}{2}, y + h\frac{k_2}{2}$$
$$k_4 = f(t + h, y + hk_3)$$

This gives a good approximation up to the fourth order and does not require you to run at a very low time step.

3 Example simulations

3.1 Terminal velocity of a ball

A simple simulation to show test and show how the physics simulator works is through simluting the terminal velocity of a ball.

This is a fairly simple simulation as it only works in one dimension.

The equation of resultant force can be given by adding the weight, bouyancy force and air resistance, which can be given by these equation, where m is the mass, g is the gravitational acceleration constant, r is the radius of the ball, ρ is fluid density, v is current velocity of the ball and c is the drag coefficient.

$$weight = mg$$

$$air\ resistance = \frac{1}{2}\rho v^2 c \cdot 2\pi r^2$$

$$bouyancy = -\rho g \cdot \frac{4}{3}\pi r^3$$

This can then be added and divided by the mass to find the accleration of the ball.

This can then simply be called at each iteration and run through a ODES to find the velovity and then once again to find the position.

I then output this into a Comma seperated values(CVS) file and create the graph using excel.

In Figure 2 series 1 is the x-componet of velocity, series 2 is the y-component of velocity, series 3 is the x-component of displacement and series 4 is the y-component of displacement.

```cpp
void ball_terminal_velocity(){
    fstream fout;
    fout.open("graph.csv", ios::out | ios::trunc);
    //timestep(h), time(x), velocity(y)
    double s = 0.01, tc = 0;
    vec2d vc = vec2dcreate(0,0,0), pc = vec2dcreate(0,0,0);
    while(vc.first.second != rk4(tc,vc,s,fball).first.second){//end condition
        tc += s;
        vc = rk4(tc, vc, s, fball);
        pc = compound_rk4(tc, vc, pc, s, fball);
        fout << tc << ',' << vc.first.first << ',' << vc.first.second << ',' << pc.first.first << ',' << pc.first.second << "\n";
    }
}
```

Figure 1: Code implementing ball terminal velocity

3.2 Thrown ball

To take the engines abilities to the next step we make it so that it can simulate in 2 dimensions. We can test and demonstrate this by simulating a thrown ball.

To simulate in 2 dimension, we need to have a way to store and add vectors. I stored by vector by using a pair of pairs. The first pairs stores the vector like cartesian coordinates by storing the x and y component. the second pair stores the vector like polar coordinates storing the magnitude and direction of the vector.

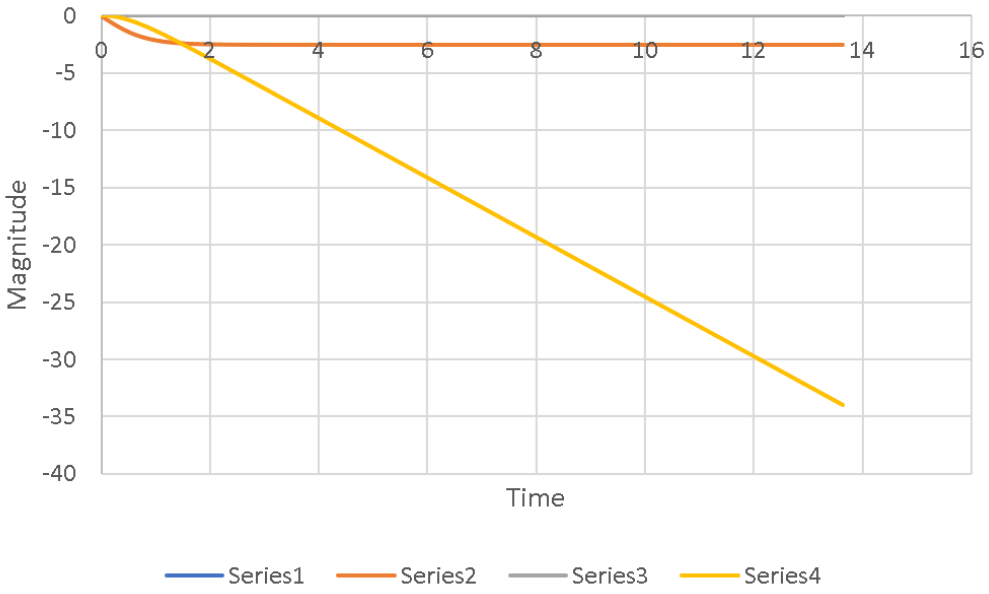

Figure 2: Graph from ball terminal velocity function

```
12
13    typedef pair<double, double> pd;
14    typedef pair<pd,pd> vec2d; //{{cartesian x, y},{polar d, angle}}
15
```

Figure 3: Defining 2D vectors

There is minimal changes needed to be done to the force generator and the ODES which ainly include changing what value types the function takes in.

Then we can run the programme again after adding initial velocities to simulate a thrown ball.

In Figure 5 , series 1 is the x-componet of velocity, series 2 is the y-component of velocity, series 3 is the x-component of displacement and series 4 is the y-component of displacement.

```
30  void thrown_ball(){
31      fstream fout;
32      fout.open("graph.csv", ios::out | ios::trunc);
33      //timestep(h), time(x), velocity(y)
34      double s = 0.0001, tc = 0;
35      vec2d vc = vec2dcreate(50,0,0), pc = vec2dcreate(0,0,0);
36      while(tc <= 10){//end condition
37          tc += s;
38          vc = rk4(tc, vc, s, fball);
39          pc = compound_rk4(tc, vc, pc, s, fball);
40          fout << tc << ',' << vc.first.first << ',' << vc.first.second << ',' << pc.first.first << ',' << pc.first.second << "\n";
41      }
42  }
```

Figure 4: Code implementing a thrown ball

3.3 Single pendulum

In order to demonstrate how the physics engine simulates a system with angular velocity, we can use a simulation of a simple pendulum.

To simplify the calculations for angular accleration, we can use a lagrangian to solve it instead. We can start create this lagrangian by defining the constraints of the x and y displacements of the bob.

$$x = lsin\theta$$

$$y = -lcos\theta$$

We can then use the first derivitives of this coordinates with respect to time $\dot{\vec{X}} = l\dot{\vec{\theta}}cos\theta$ and $\dot{Y} = l\dot{\theta}cos\theta$ to find the Kinetic energy.

$$Ke = \frac{1}{2}mv^2$$

$$= \frac{1}{2}m(l^2\dot{\theta}^2(cos\theta)^2 + l^2\dot{\theta}^2(sin\theta)^2)$$

$$\frac{1}{2}ml^2\dot{\theta}^2$$

The lagragian will then be,

$$L = \frac{1}{2}ml^2\dot{\theta}^2 - mg(l + y)$$

$$= \frac{1}{2}ml^2\dot{\theta}^2 - mgl + mglcos\theta$$

Now that we have the lagrangian we can use Eular's lagrange equation $\frac{dp\theta}{dt} - \frac{\partial L}{\partial \theta} = 0$ to find the angular acceleration.

$$ml^2\ddot{\theta} + mglsin\theta$$

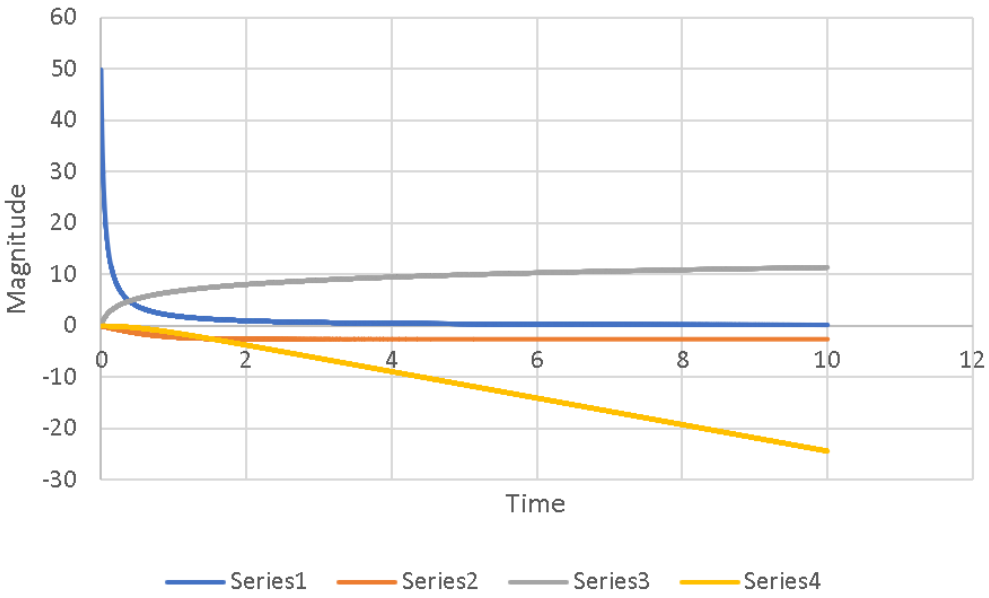

Figure 5: Graph from thrown ball function

$$\ddot{\theta} = \frac{-gsin\theta}{l}$$

The are no changes to be made to the ODES and we can simply run the function to simulate the simple pendulum.

```
42  void single_pendulum(){
43      fstream fout;
44      fout.open("graph.csv", ios::out | ios::trunc);
45      double s = 0.00001, tc = 0;
46      while(tc <= 10){//end condition
47          tc += s;
48          pendulum1.angv = rk4(tc, pendulum1.angv, pendulum1.pos.second.second, s, fpendulum);
49          pendulum1.ang = compound_rk4(tc, pendulum1.angv, pendulum1.ang, s, fpendulum);
50          pendulum1.pos = vec2dcreate(pendulum1.length,PI-pendulum1.ang,1);
51          fout << tc << ',' << pendulum1.angv << ',' << pendulum1.ang << ',' << pendulum1.pos.first.first << ',' << pendulum1.pos.first.second
52      }
53  }
```

Figure 6: Code implementing a simple pendulum

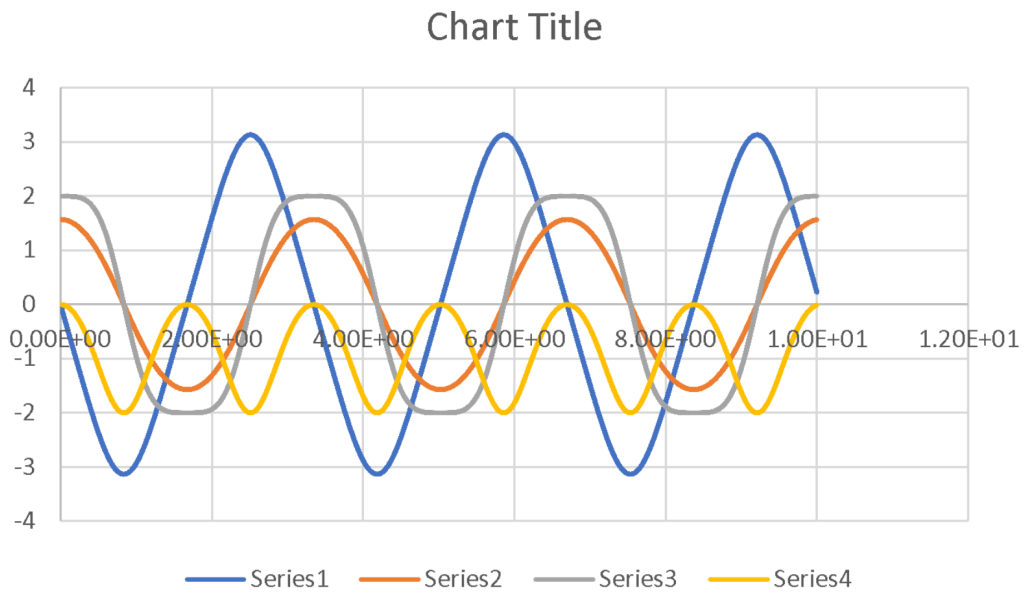

Figure 7: Graph from simple pendulum fucntion

In Figure 7 below, series 1 is the angular velocity, series 2 is the angle from the center, series 3 is the x-component of displacement and series 4 is the y-component of displacement.

One interesting thing we can use this for is to show how the small angle aproximation equation of period of a simple pendulum becomes in accurate when the initial angle becomes to big.

This is something that can also be observed in the derivation of this formular as it makes and assumtion that $sin\theta = \theta$ which is only true for very small values.

We can show this by creating a condition so that the programe will output the time after it makes one oscillation. We can then iterate through many initial angles to create a graph. This can then be compared with the equation to show that the period changes when the intial angle is increased unlike what the equation suggest.

From Figure 9 we can see that the equation slowly becomes more and more inaccurate as the initial angle increases and becomes very inaccurate by 1dp at an initial angle of 1 radian.

It is also worth noting that exel, which i was using to plot my graphs became very slow as the amount of values increased, so i created a python

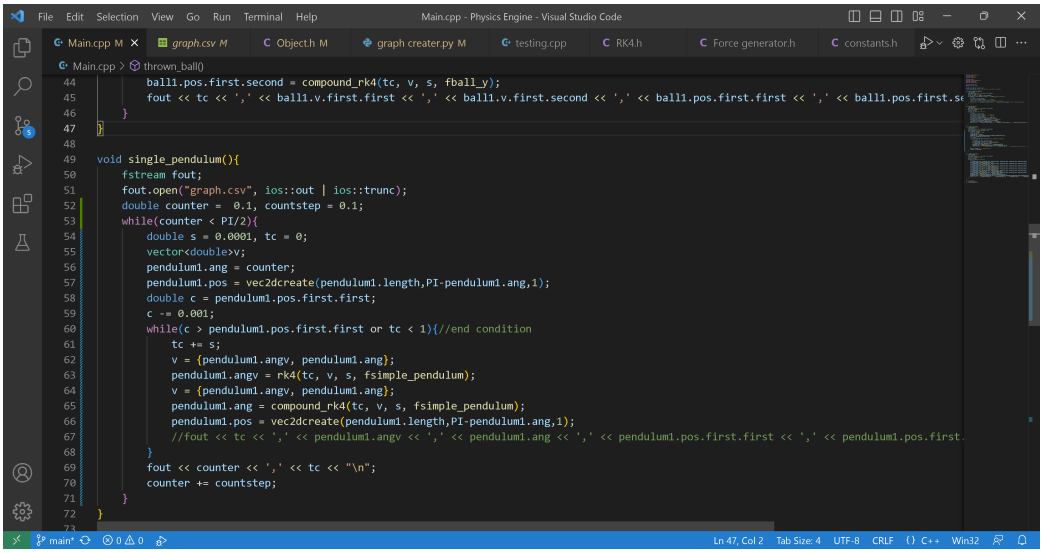

Figure 8: Code implementing the period of a simple pendulum

programme to plot the graph suing matplotlib instead.

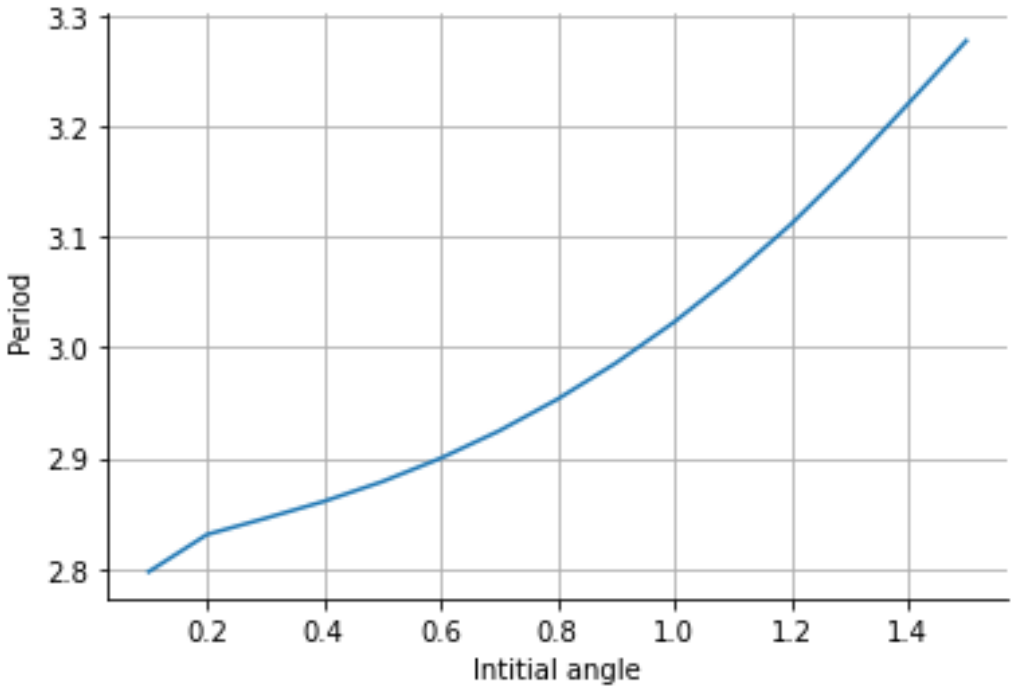

Figure 9: Graph from period of simple pendulum fucntion

3.4 Double pendulum

Another interesting simulation we can do is a double pendulum. This is interesting as it demonstrates that physics engine can simulate choatic systems.

We can start doing this by solving a lagrangian as it would be much simpler then solving using newtonian mechanics.

$$x_1 = l_1 sin\theta_1$$

$$y_1 = -l_1 cos\theta_1$$

$$x_2 = l_1 sin\theta_1 + l_2 sin\theta_2$$

$$y_2 = -l_1 cos\theta_1 - l_2 cos\theta_2$$

$$\dot{x}_1 = l_1 \dot{\theta}_1 cos\theta_1$$

$$\dot{y}_1 = l_1 \dot{\theta}_1 sin\theta_1$$

$$\dot{x}_2 = l_1\dot{\theta}_1 cos\theta_1 + l_2\dot{\theta}_2 cos\theta_2$$
$$\dot{y}_2 = l_1\dot{\theta}_1 sin\theta_1 + l_2\dot{\theta}_2 sin\theta_2$$

The lagrangian would then be,

$$L = \frac{1}{2}m_1 v_1^2 + \frac{1}{2}m_2 v_2^2 - mg(l_1 + y_1) - mg(l_1 + l_2 + y_2)$$

$$= \frac{1}{2}m_1 l_1^2\dot{\theta}_1^2 + \frac{1}{2}m_2 l_1^2\dot{\theta}_1^2 + \frac{1}{2}m_2 l_2^2\dot{\theta}_2^2 + m_2 l_1 l_2\dot{\theta}_1\dot{theta}_2 cos(\theta_1 - \theta_2) - m_1 g l_1$$
$$+m1gl_1 cos\theta_1 - m_2 g l_1 - m_2 g l_2 + m_2 g l_1 cos\theta_1 + m_2 g l_2 cos\theta_2$$

The eular lagrange equation for $\ddot{\theta}_1$ would then be,

$$m_1 l_1^2\ddot{\theta}_1 + m_2 l_1]^2\ddot{\theta}_1 + m_2 l_1 l_2\ddot{\theta}_2 cos(\theta_1 - \theta_2) - m_2 l_1 l_2\dot{\theta}_1\dot{\theta}_2 sin(\theta_1 - \theta 2)$$

$$+m_2 l_1 l_2\dot{\theta}_2^2 sin(\theta_1 - \theta_2) + m_2 l_1 l_2\dot{\theta}_1\dot{\theta}_2 sin(\theta_1 - \theta_2) + m_1 g l_1 sin\theta_1 + m_2 g l_1 sin\theta_1 = 0$$

$$\ddot{\theta}_1 = \frac{-m_2 l_2\ddot{\theta}_2 cos(\theta_1 - \theta_2) + m_2 l_2\dot{\theta}_1\dot{\theta}_2 sin(\theta_1 - \theta 2) - m_2 l_2\dot{\theta}_2^2 sin(\theta_1 - \theta_2)}{m_1 l_1 + m_2 l_1}$$
$$+\frac{-m_2 l_2\dot{\theta}_1\dot{\theta}_2 sin(\theta_1 - \theta_2) - m_1 g sin\theta_1 - m_2 g l_1 sin\theta_1}{m_1 l_1 + m_2 l_1}$$

and the eular lagrange equation for $\ddot{\theta}_2$ would then be,

$$m_2 l_2^2\ddot{\theta}_2 + m_2 l_1 l_2\ddot{\theta}_1 cos(\theta_1 - \theta_2) - m_2 l_1 l_2\dot{\theta}_1^2 sin(\theta_1 - \theta_2) + m_2 g l_2 sin\theta_2 = 0$$

$$\ddot{\theta}_2 = \frac{-l_1\ddot{\theta}_1 cos(\theta_1 - \theta_2) + l_1\dot{\theta}_1^2 sin(\theta_1 - \theta_2) - g sin(\theta_2)}{l_2}$$

However as this is a very complex function compared to all the previous function as it takes in multiple variables, we have to modify our ODES and force generator functions to allow it to accept and calculate correctly for multiple vairable.

If you are using RK4 for your ODES the variables would simply be edited the same way each and then inputed into the force function. This also allows our ODES to be more general and be able to be applied to many different function

```cpp
double rk4(double x, vector<double> v,double s, double (*f)(double,vector <double>)){
    vector <double> vk1, vk2, vk3, vk4;

    double k1 = f(x,v);
    for(auto i: v) vk1.push_back(i+s*k1*0.5);
    double k2 = f(x+0.5 * s,v);
    for(auto i: v) vk2.push_back(i+s*k2*0.5);
    double k3 = f(x+0.5 * s,vk2);
    for(auto i: v) vk3.push_back(i+s*k3);
    double k4 = f(x+s,vk3);
    double a = v[0]+s/6*(k1 + 2*k2 + 2*k3 +k4);
    return a;
}
```

Figure 10: Code implementing general RK4

We can then create 2 seperate functions for the angular acceleration of bob1 and bob2 and create a new function calling both functions seperately.

The following graphs are of the double pendulum when it starts to an intial angle of 30 degrees each.

```cpp
void double_pendulum(){
    fstream fout;
    fout.open("graph.csv", ios::out | ios::trunc);
    double s = 0.0001, tc = 0;
    vector<double>v;
    while(tc <= 10){//end condition
        tc += s;
        v = {pendulum2.angv1, pendulum2.ang1, pendulum2.anga1, pendulum2.angv2, pendulum2.ang2, pendulum2.anga2};
        pendulum2.anga1 = fdouble_pendulum_1(tc,v);
        v = {pendulum2.angv2, pendulum2.ang2, pendulum2.anga2, pendulum2.angv1, pendulum2.ang1, pendulum2.anga1};
        pendulum2.anga2 = fdouble_pendulum_2(tc,v);
        v = {pendulum2.angv1, pendulum2.ang1, pendulum2.anga1, pendulum2.angv2, pendulum2.ang2, pendulum2.anga2};
        pendulum2.angv1 = rk4(tc, v, s, fdouble_pendulum_1);
        v = {pendulum2.angv2, pendulum2.ang2, pendulum2.anga2, pendulum2.angv1, pendulum2.ang1, pendulum2.anga1};
        pendulum2.angv2 = rk4(tc, v, s, fdouble_pendulum_2);
        v = {pendulum2.angv1, pendulum2.ang1, pendulum2.anga1, pendulum2.angv2, pendulum2.ang2, pendulum2.anga2};
        pendulum2.ang1 = compound_rk4(tc, v, s, fdouble_pendulum_1);
        v = {pendulum2.angv2, pendulum2.ang2, pendulum2.anga2, pendulum2.angv1, pendulum2.ang1, pendulum2.anga1};
        pendulum2.ang2 = compound_rk4(tc, v, s, fdouble_pendulum_2);
        pendulum2.pos1 = vec2dcreate(pendulum2.length1,PI-pendulum2.ang1,1);
        pendulum2.pos2 = vec2dcreate(pendulum2.pos1.first.first+pendulum2.length2*sin(pendulum2.ang2),pendulum2.pos1.first.second-pendulum2.
        fout << tc << ',' << pendulum2.angv1 << ','  << pendulum2.angv2 << ',' << pendulum2.ang1 << ',' << pendulum2.ang2 << ',' << pendulu
    }
}
```

Figure 11: Code implementing general Double pendulum

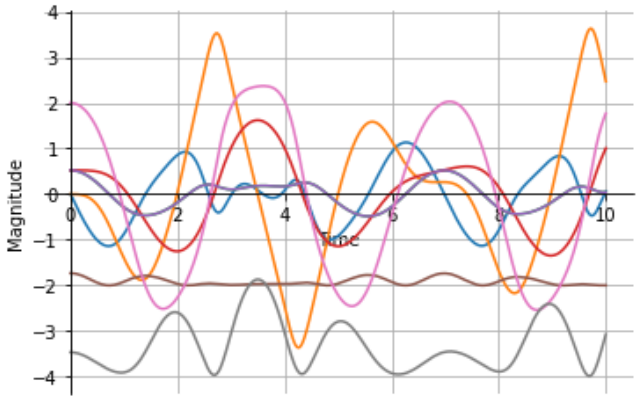

Figure 12: Graph of double pendulum function

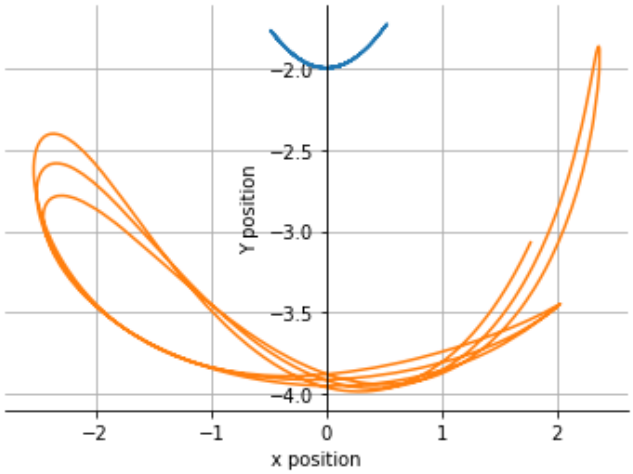

Figure 13: Graph of movement from double pendulum graph

4 Conclusion

A physics simulator can be used for many things and the physics simulator i have built can be further advanced to add constraints and to make it more general so that i can be applied to more complex systems with many more objects.

The code of my physics engine can be found at https://github.com/cadenlyy/Physics-Engine.

5 Reflection

This article focuses much on the software and mathematics used to simulate physics. It attempts to show the applications of such engines in analysis of physics. However, this engine could be made more generic through constrains and generalising the functions used. This would allow creation of more complex systems with ease as it would allow one to simply create an object and define it initial conditions to simulate the system.

This system could also have been made more reader freindly by using diagrams and psuedo code in explainations as it is easier to read than C++. I could also have added a graphical interface to allow better demonstration on how the system moves instead of simply using graphs. Further more, I could also have included a comparison between the use of RK4 and Eular's intergration to show the difference instead of simply explaining it qualitatively.

I first started writing with physics engine in January and then adapted it to fit this portflio. This project took me a good 6 months to finish as through it i learned many new things such as RK4, how to use and calculate lagraingians, what constrains are and so on. This project was somewhat of a personal interest and i had a lot of fun learning how to create a physics engine through watching videos and reading articles.

6 References

Davide Marenduzzo, University of Edinburgh. https://www2.ph.ed.ac.uk/ dmarendu/MVP/Double