

Runge-Kutta 방법을 활용한 물리 운동의 수치해석적 풀이와 구현



나노융합스쿨 디스플레이학과

20163329 홍현준

지도교수: 이종완

INDEX

- 1 개요
- 2 Runge-Kutta Method
- 3 스프링 운동 시뮬레이터 구현
- 4 자동차 시뮬레이터 구현



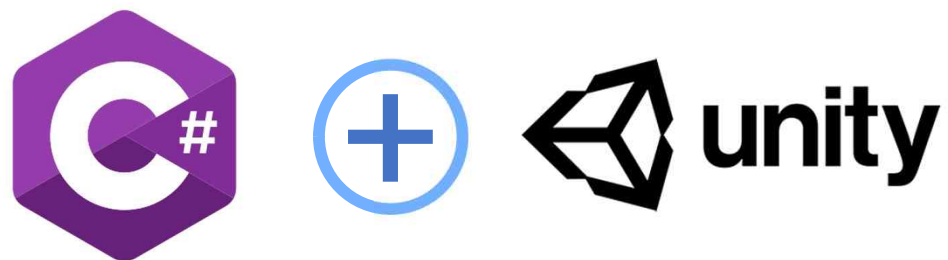
1

개요



1 개요

구현 환경



C# 기반 Window Form과 유니티 엔진을 사용하여 구현

- Visual Studio 2019
- Windows Forms 앱(.NET Framework 4.7.2)
- Unity Ver.2020.3.5f1



1 개요

진행 방식

- 01 수치해석적 풀이인 Runge-Kutta 방법을 이용하여 여러 물리 운동 방정식을 풀이
- 02 C# 언어를 사용하여 물리 방정식을 코드화
- 03 WindowForm을 사용하여 2차원에서 구현
- 04 Unity를 사용하여 3차원에서 구현



2 Runge-Kutta Method



2 Runge-Kutta Method

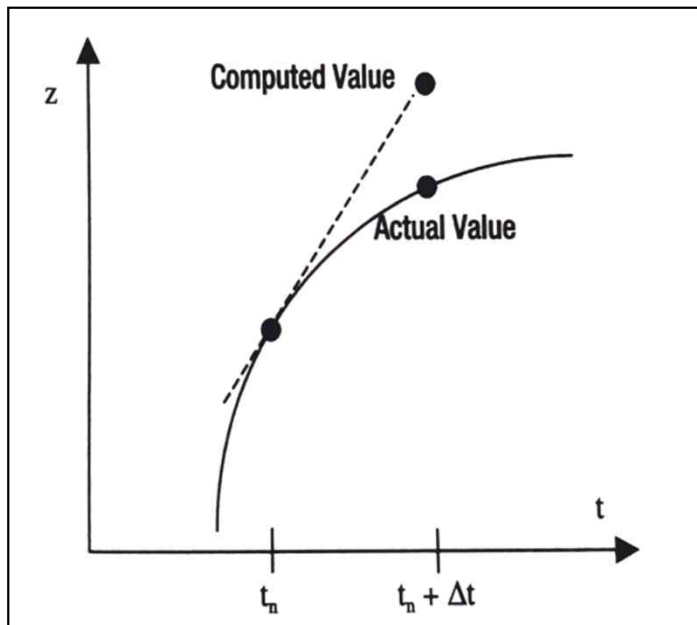


그림 2. 위치-시간 비선형 그래프

$$\Delta z = z_{n+1} - z_n = v_z(z_n, t_n) \Delta t \quad (\text{오일러 법})$$

오일러 법

어떤 물리 모델링에서 위치 값을 원하지만 위치 식을 도출할 수 없고 속도만 알고 있을 때 미분 방정식의 수치해석 풀이를 사용한다.

시간에 따른 속도의 변화가 심한 비선형 그래프에서 오일러 법을 이용하여 t_n 에서 $t_n + \Delta t$ 의 속도를 구하게 되면 실제 값과 계산된 추정 값과의 오차가 커질 수 있다는 한계가 존재한다.



2 Runge-Kutta Method

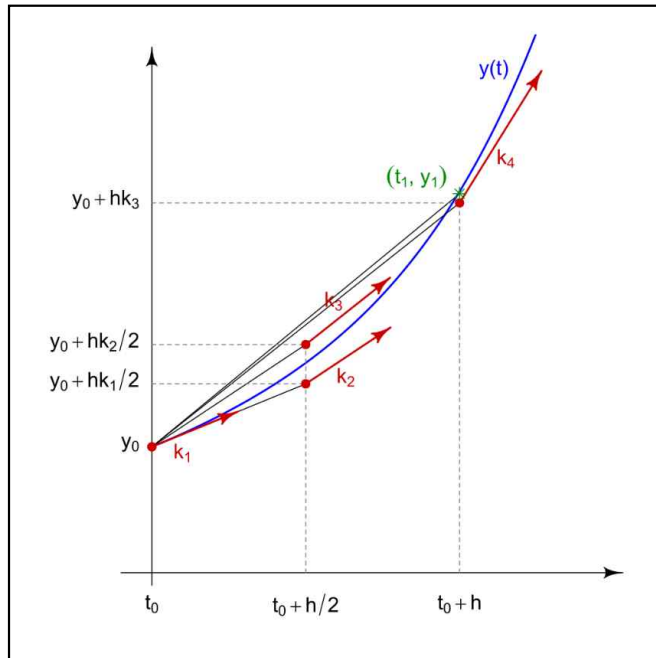


그림 2. 룬게-쿠타 방정식의 시각화 그래프

Runge-Kutta 방법

4차 Runge-Kutta 방정식은 구하고자 하는 시간의 범위 내에서 4개의 시간에 대한 점에서의 기울기를 이용하여 다음 점에서의 함수 값을 계산한다.

이렇게 함으로써 오일러 법에서의 한계였던 위치와 시간에 대한 비선형 그래프에서도 상당히 정확한 값을 얻을 수 있다.



2 Runge-Kutta Method

Boundary Conditions(경계 조건)

룽게-쿠타 방정식을 풀기 위해서는 초기값을 알아야 한다.

여기서 초기값이란 예를 들어 탄환의 비행에 대한 물리를 모델링 한다면 탄환의 초기 속도와 초기 위치라고 할 수 있다.

Runge-Kutta 방정식은 독립 변수를 연속적으로 증가시키는 방법인 Marching이라는 프로세스를 사용하여 이러한 유형의 ODE를 해결하며, 단계마다 종속 변수의 값을 업데이트한다.



2 Runge-Kutta Method

$$\Delta z_1 = v(z_n, t_n) \Delta t$$

$$\Delta z_2 = v\left(z_n + \frac{1}{2} \Delta z_1, t_n + \frac{1}{2} \Delta t\right) \Delta t$$

$$\Delta z_3 = v\left(z_n + \frac{1}{2} \Delta z_2, t_n + \frac{1}{2} \Delta t\right) \Delta t$$

$$\Delta z_4 = v(z_n + \Delta z_3, t_n + \Delta t) \Delta t$$

$$z_{n+1} = z_n + \frac{\Delta z_1}{6} + \frac{\Delta z_2}{3} + \frac{\Delta z_3}{3} + \frac{\Delta z_4}{6}$$

공식 1. 4차 Runge-Kutta 방정식

일반적인 4차 Runge-Kutta 방법

일반적으로 사용하는 룽게-쿠타 방법은 4차 룽게-쿠타 방법이다.
왼쪽 공식에서 초기값인 z_0, t_0 은 주어져 있다고 가정한다.

1. Δz_1 은 구간의 시작부분의 기울기에 의한 증분값
2. Δz_2 는 구간의 중간의 기울기에 의한 증분값, $z + \frac{1}{2} \Delta z_1$ 을 사용
3. Δz_3 역시 구간의 중간의 기울기에 의한 증분값이지만, $z + \frac{1}{2} \Delta z_2$ 을 사용
4. Δz_4 는 구간의 끝의 기울기에 의한 증분값, $z + \Delta z_3$ 사용
5. z_{n+1} 은 현재 단계의 값에 위의 네 구간의 크기의 증분치들의 가중평균을 더한 값이다.



2 Runge-Kutta Method

```
public abstract class ODE
{
    // number of Equations : 1차 ODE 방정식의 개수
    private int numEqns;
    // 종속 변수들의 배열
    private double[] q;
    // 독립 변수
    private double s;

    public ODE(int numEqns)
    {
        // numEqns를 매개변수 numEqns로 초기화
        this.numEqns = numEqns;
        // 종속 변수들의 배열의 크기를 numEqns의 크기로 초기화
        this.q = new double[numEqns];
    }

    public abstract double[] GetRightHandSide
    (double s, double[] q, double[] deltaQ, double ds, double qScale);
}
```

이 곳에는 핵심적인 코드들만 실어 놓았으며 전체 코드는 깃허브를 통해 업로드 한다.

ODE class

이 class는 일반적인 ODE(상미분방정식)를 나타낸다. 예를 들어 스프링이나 발사체 운동을 모델링하기 위해 특정 유형의 ODE를 나타내는 class를 ODE class의 하위 클래스로 작성한다.

GetRightHandSide 메소드

ODE 방정식을 풀려면, 미분 방정식의 우측 항을 풀어야 한다. 예를 들어, 스프링 운동에서 두 개의 1차 ODE를 풀기 위해서는 GetRightHandSide 메소드가 $(-\mu v_x - kx)/m$ 과 v_x 를 계산한 후 반환해줄 필요가 있다.

이 메소드는 abstract(추상)로 선언되었기 때문에, ODE의 하위 클래스에서 메소드의 세부 구현을 해 줄 것이다.



2 Runge-Kutta Method

```
public class ODESolver
{
    // 4차 Runge-Kutta ODE solver
    // 여러 곳에서 사용하기 위해서 public static으로 선언
    public static void RungeKutta4(ODE ode, double ds)
    {
        double s; //독립 변수
        double[] q; //종속 변수
        double[] dq1 = new double[ode.NumEqns];
        double[] dq2 = new double[ode.NumEqns];
        double[] dq3 = new double[ode.NumEqns];
        double[] dq4 = new double[ode.NumEqns];

        //종속&독립 변수의 현재 값 검색하여 할당
        s = ode.S;
        q = ode.GetAllQ();

        // getRightHandSide 메소드의 반환값은 4단계 각각에 대한 델타 q의 배열임
        // 4번째 매개변수: 가중치
        dq1 = ode.GetRightHandSide(s, q, q, ds, 0.0);
        dq2 = ode.GetRightHandSide(s + 0.5 * ds, q, dq1, ds, 0.5);
        dq3 = ode.GetRightHandSide(s + 0.5 * ds, q, dq2, ds, 0.5);
        dq4 = ode.GetRightHandSide(s + ds, q, dq3, ds, 1.0);

        // 새 종속 변수 위치에서 종속 변수 및 독립 변수 값을 업데이트하고 값을 ODE 객체 배열에 저장함
        ode.S = s + ds;

        for (int i = 0; i < ode.NumEqns; ++i)
        {
            q[i] = q[i] + 1.0 / 6.0 * (dq1[i] + 2.0 * dq2[i] + 2.0 * dq3[i] + dq4[i]);
            ode.SetQ(q[i], i);
        }

        return;
    }
}
```

ODESolver class

4차 Runge-Kutta 방정식의 5단계의 풀이를 구현한다.

ODE 객체를 사용하여 독립변수(s)와 종속변수(q[])의 현재 값을 가져온다.

dq1[] ~ dq4[] : 종속 변수에 대한 중간 업데이트를 저장할 배열들



3 스프링 운동 시뮬레이터 구현



3 스프링 운동 시뮬레이터 구현

$$m \frac{d^2 x}{dt^2} + \mu \frac{dx}{dt} + kx = 0$$

공식 2. 스프링 운동 방정식

$$m \frac{dv_x}{dt} = -\mu v_x - kx$$

공식 3. 속도와 시간에 관한 부분

먼저 Runge-Kutta 방정식을 사용하여 스프링 운동을 시뮬레이션 해본다.

스프링 방정식은 시간과 관련된 두 개의 연립 1차 미분 방정식으로 나타낼 수 있다.

k : 스프링 상수

μ : 감쇠 계수



3 스프링 운동 시뮬레이터 구현

스프링 운동의 Runge-Kutta 방정식을 이용한 풀이

$$fa = \frac{dv}{dt} = -\frac{\mu v + kx}{m}$$

k : 스프링 상수

μ : 감쇠 계수

초기 값

mass = 1.0 kg

$\mu = 1.5$ N-s/m

k = 20 N/m

x0 = -0.2 m

$$dv_1 = fa(v_0, x_0) * dt$$

$$dx_1 = v_0 * dt$$

$$dv_2 = fa\left(v_0 + \frac{dv_1}{2}, x_0 + \frac{dx_1}{2}\right) * dt$$

$$dx_2 = \left(v_0 + \frac{dv_1}{2}\right) * dt$$

$$dv_3 = fa\left(v_0 + \frac{dv_2}{2}, x_0 + \frac{dx_2}{2}\right) * dt$$

$$dx_3 = \left(v_0 + \frac{dv_2}{2}\right) * dt$$

$$dv_4 = fa(v_0 + v_3, x_0 + x_3) * dt$$

$$dx_4 = (v_0 + v_3) * dt$$

$$v_{n+1} = v_n + \frac{1}{6}(dv_1 + 2dv_2 + 2dv_3 + dv_4)$$

$$x_{n+1} = x_n + \frac{1}{6}(dx_1 + 2dx_2 + 2dx_3 + dx_4)$$



3 스프링 운동 시뮬레이터 구현 - Window Form

```
class SpringODE : ODE
{
    private double mass; // 스프링 끝의 무게
    private double mu; // 감쇠 계수
    private double k; // 스프링 상수
    private double x0; // 초기 스프링 변형(위치)

    //SpringODE 생성자: ODE 생성자를 호출함
    // 스프링 운동은 2개의 1차 ODE
    public SpringODE(double mass, double mu, double k, double x0) : base(2)
    {
        this.mass = mass;
        this.mu = mu;
        this.k = k;
        this.x0 = x0;

        // 종속 변수들의 초기 상태 세팅
        // q[0]=vx : 초기 속도 0
        // q[1]=x : 초기 위치 x0
        SetQ(0.0, 0);
        SetQ(x0, 1);
    }

    // 아래 Get 메소드들은 ODE solver에서 계산된 스프링의 위치와 속도를 반환함
    public double GetVx() { return GetQ(0); } //종속 변수 1

    public double GetX() { return GetQ(1); } //종속 변수 2

    public double GetTime() { return this.S; } //독립 변수 1: ODE 클래스의 S

    // 이 메소드는 운동 방정식을 적분하기 위해
    // 4차 Runge-Kutta 솔버를 사용하며 스프링의 속도와 위치를 업데이트함
    public void UpdatePositionAndVelocity(double dt)
    {
        // this : SpringODE가 ODE를 상속받았으므로 SpringODE를 인자로 넣어줌
        ODESolver.RungeKutta4(this, dt);
    }
}
```

SpringODE class

SpringODE 는 ODE를 상속 받음.

종속 변수 1: 속도(v)

종속 변수 2: 위치(x)

독립 변수: 시간(t)



3 스프링 운동 시뮬레이터 구현 - Window Form

```
// 두 개의 1차 감쇠 스프링 ODE의 우측 항 반환
// q[0] = vx
// q[1] = x
// dq[0] = dvx = dt * (-mu * dx/dt - k * x) / mass
// dq[1] = dx = dt * vx
public override double[] GetRightHandSide
(double s, double[] q, double[] deltaQ, double ds, double qScale)
{
    double[] dq = new double[4]; // 우측 항 값들
    double[] newQ = new double[4]; // 중간 종속 변수 값들

    // 종속 변수들의 중간 값들을 계산
    for(int i = 0; i < 2; ++i)
    {
        newQ[i] = q[i] + qScale * deltaQ[i];
    }

    // 우측 항 값들 계산
    dq[0] = - ds * (mu * newQ[0] + k * newQ[1]) / mass;
    dq[1] = ds * (newQ[0]);

    return dq;
}
```

GetRightHandSide 메소드

ODE 클래스에서 추상적으로 선언해주었던 GetRightHandSide 메소드를 override하여 스프링 운동에 맞춰 메소드를 작성해준다.

newQ 배열

Ex) Runge-Kutta 에서

$$\text{newQ}[0] = v_x + qScale * dv_n$$

$$\text{newQ}[1] = x + qScale * dx_n$$

※ qScale 은 1, 0.5, 0.5, 1 로 룬게-쿠타 방정식의 단계별로 다르게 할당된다.

dq[0] : 공식 3의 Runge-Kutta 방정식

dq[1] : 공식 4의 Runge-Kutta 방정식



3 스프링 운동 시뮬레이터 구현 - Window Form

```
class RK4Spring
{
    [STAThread]
    static void Main()
    {
        double mass = 1.0; //1kg
        double mu = 1.5; //1.5 N-s/m
        double k = 20.0; //20 N/m
        double x0 = -0.2;

        SpringODE ode = new SpringODE(mass, mu, k, x0);

        //7초까지 0.1초마다 업데이트
        double dt = 0.1;

        Console.WriteLine("t   x   v");
        Console.WriteLine(" " + ode.GetTime() + "   " + (float)ode.GetX() +
            "   " + (float)ode.GetVx());

        while (ode.GetTime() <= 7.0)
        {
            ode.UpdatePositionAndVelocity(dt);
            Console.WriteLine(" " + ode.GetTime() + "   " + (float)ode.GetX()
                + "   " + (float)ode.GetVx());
        }

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new SpringSimulator());

        return;
    }
}
```

RK4Spring class

이 클래스는 0~7초까지의 스프링 운동을 Runge-Kutta 방정식으로 푼 값들을 콘솔창으로 표시하는 역할을 한다. dt=0.1로 0.1초 간격으로 스프링 운동을 계산한다.

초기 값

mass = 1.0 kg

mu = 1.5 N-s/m

k = 20 N/m

x0 = -0.2 m



3 스프링 운동 시뮬레이터 구현 - Window Form

SpringSimulator class

이 클래스에서는 GUI 설정과 초기 값 설정, 그리고 화면을 업데이트를 해준다.

```
public partial class SpringSimulator : Form
{
    private Panel drawingPanel;

    //게임의 스피드를 컨트롤하기 위해 사용함
    private Timer gameTimer;

    SpringODE springODE;

    private void UpdateDisplay()
    {
        Graphics g = drawingPanel.CreateGraphics();
        int width = drawingPanel.Width - 1;

        g.Clear(drawingPanel.BackColor);

        Pen blackPen = new Pen(Color.Black, 2);
        g.DrawLine(blackPen, 0, 20, width, 20);

        SolidBrush brush = new SolidBrush(Color.Black);

        int zPosition = (int)(125 - 100.0 * springODE.GetX());
        g.FillRectangle(brush, 65, zPosition, 20, 20);
        g.DrawLine(blackPen, 75, 20, 75, zPosition + 10);

        g.Dispose();
    }
}
```

```
private void ActionPerformed(object sender, EventArgs e)
{
    // 스프링의 위치를 업데이트하기 위해 ODE solver 사용
    double dt = 0.05;
    springODE.UpdatePositionAndVelocity(dt);

    UpdateDisplay();
}

private void startButton_Click(object sender, EventArgs e)
{
    // 각 초기 값은 프로그램의 Text를 통해 받아온다.
    double mass = Convert.ToDouble(massTextBox.Text);
    double mu = Convert.ToDouble(muTextBox.Text);
    double k = Convert.ToDouble(kTextBox.Text);
    double x0 = Convert.ToDouble(x0TextBox.Text);

    springODE = new SpringODE(mass, mu, k, x0);

    gameTimer.Start();
}

private void resetButton_Click(object sender, EventArgs e)
{
    gameTimer.Stop();

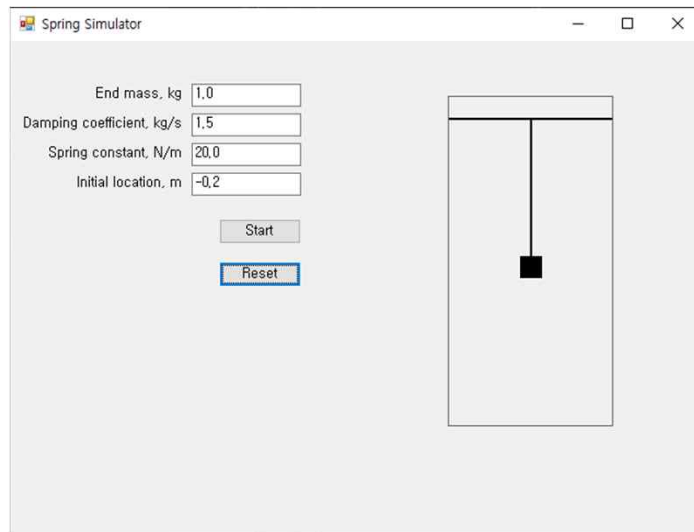
    double x0 = Convert.ToDouble(x0TextBox.Text);
    springODE.SetQ(x0, 1);

    UpdateDisplay();
}
}
```

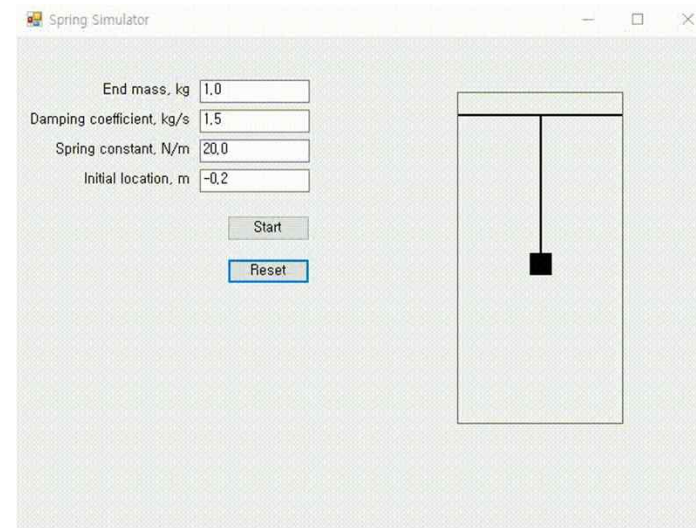


3 스프링 운동 시뮬레이터 구현 - Window Form

Window Form을 사용하여 구현한 스프링 운동 시뮬레이터



초기 세팅



실행 화면



3 스프링 운동 시뮬레이터 구현 - Unity

```
double func_a(double v, double x)
{
    double a = -(mu * v + k * x) / mass;
    return a;
}

void RK4()
{
    double dt = Time.fixedDeltaTime;
    sum_time += dt;

    double dv1 = func_a(v0, x0) * dt;
    double dx1 = v0 * dt;

    double dv2 = func_a(v0 + dv1 / 2f, x0 + dx1 / 2f) * dt;
    double dx2 = (v0 + dv1 / 2f) * dt;

    double dv3 = func_a(v0 + dv2 / 2f, x0 + dx2 / 2f) * dt;
    double dx3 = (v0 + dv2 / 2f) * dt;

    double dv4 = func_a(v0 + dv3, x0 + dx3) * dt;
    double dx4 = (v0 + dv3) * dt;

    v0 = v0 + 1d / 6d * (dv1 + 2d * dv2 + 2d * dv3 + dv4);
    x0 = x0 + 1d / 6d * (dx1 + 2d * dx2 + 2d * dx3 + dx4);

    transform.position = new Vector3(transform.position.x, (float)x0,
                                     transform.position.z);
}
```

Unity를 사용한 스프링 운동 구현

SpringMotion Class

풀이 자체는 이전 WindowForm 을 사용하여 구현한 스프링 운동 시뮬레이터와 동일하다.

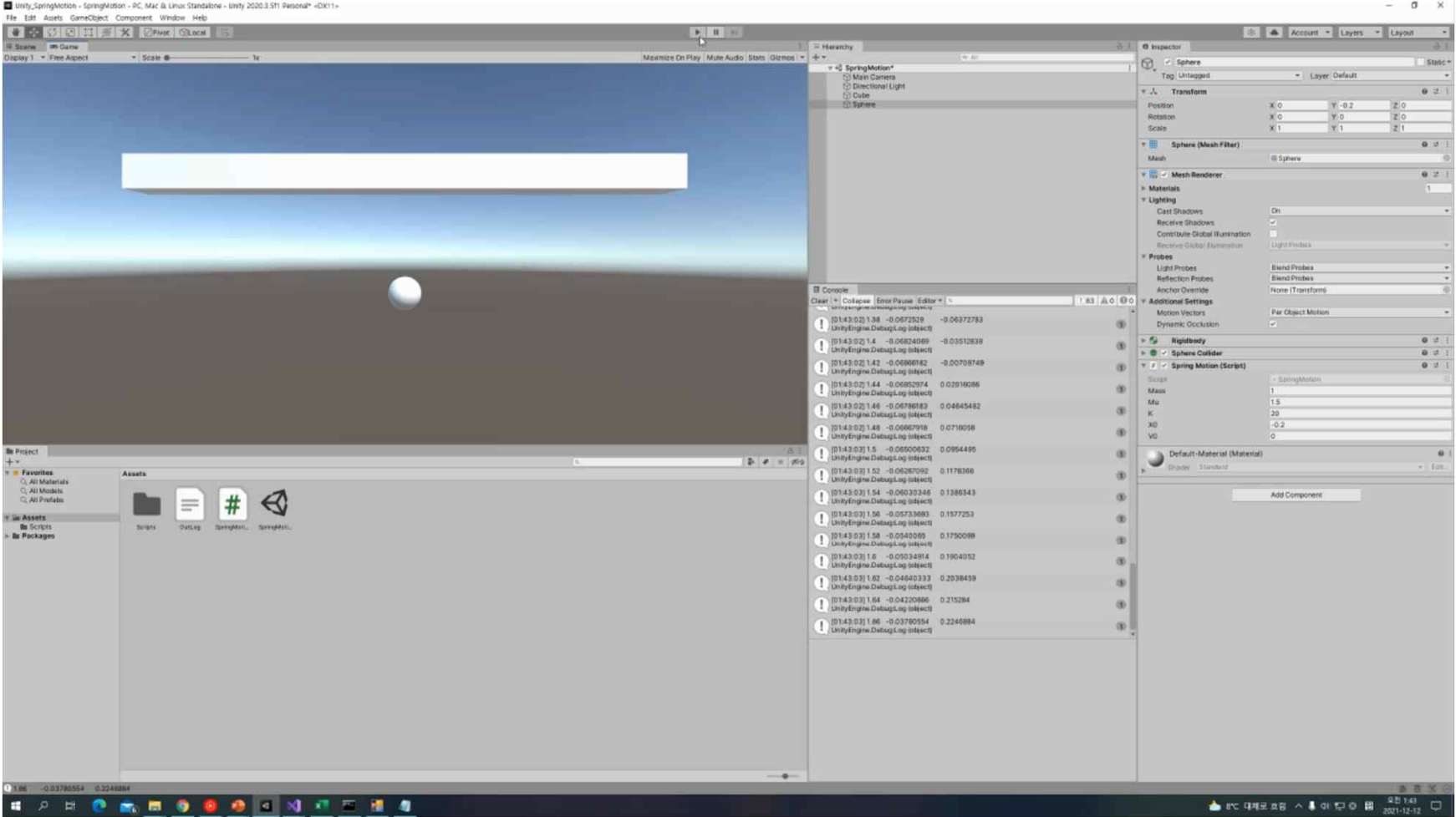
다만 여기서는 룽게-쿠타 메소드를 일반화하지 않고 스프링 운동에서만 쓰기 위하여 간략하게 작성하였다.

dt 는 0.02 초마다 업데이트 된다.

v0, x0는 처음에 초기값으로 주어지며 룽게-쿠타 메소드가 계산되는 0.02초마다 업데이트 된다.

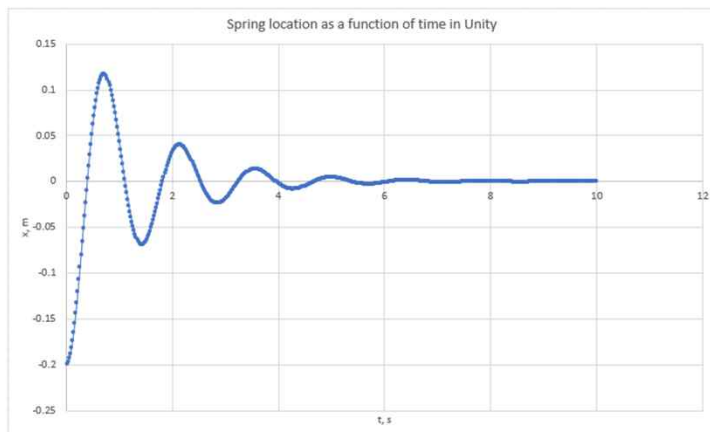
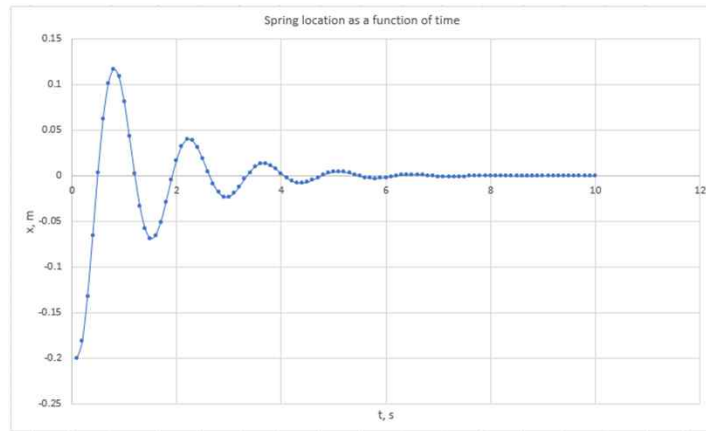


3

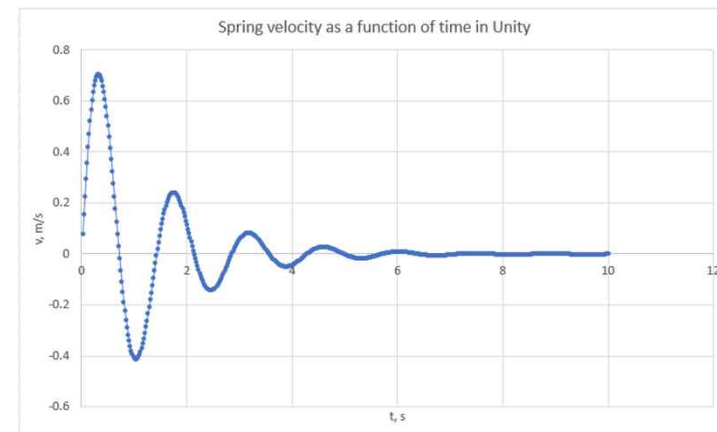
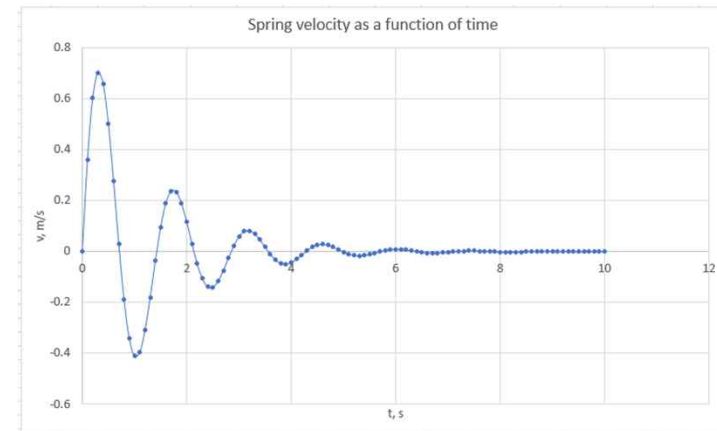


3 스프링 운동 시뮬레이터 구현

스프링 운동 시간-위치 그래프



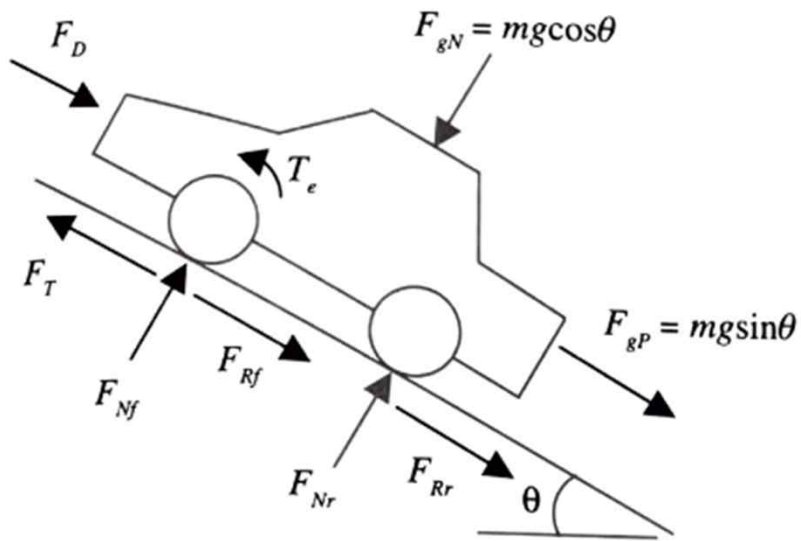
스프링 운동 시간-속도 그래프



4 자동차 시뮬레이터 구현



4 자동차 시뮬레이터 구현



자동차에 가해지는 기본 물리들

$$F_N = F_{Nf} + F_{Nr} = mg \cos \theta$$

차와 지면 간의 작용 반작용

$$F_T = \frac{T_w}{r_w}$$

휠에 가해지는 힘 = 휠에 가해지는 토크 / 휠 반경

$$F_D = \frac{1}{2} C_D \rho v^2 A$$

공기항력 = $\frac{1}{2}$ * 항력 계수 * 공기 밀도 * 속도² * 차량 앞 면적

$$F_{Total} = \frac{T_w}{r_w} - \mu_r m g \cos \theta - m g \sin \theta - \frac{1}{2} C_D \rho v^2 A$$

$$F_{Total} = F_T - (\mu_r F_N + F_{gP} + F_D)$$

차량이 움직이는 방향의 총 힘 = 타이어에 가해지는 힘 -
(롤링마찰력 + 경사면에 수평하게 작용하는 힘 + 공기항력)



4 자동차 시뮬레이터 구현

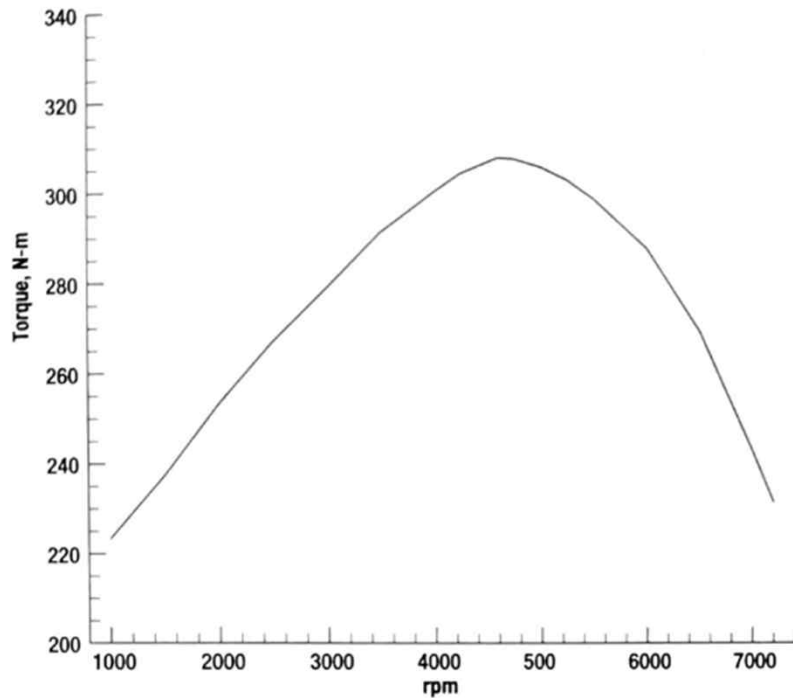


Figure 8-2. A typical torque curve

가속도 & 토크 & 엔진 출력

$$a = \frac{T_w}{r_w m} - \mu_r g \cos \theta - g \sin \theta - \frac{1}{2} \frac{C_D \rho v^2 A}{m}$$

$$\text{시간당 가속도} = F_{Total} / m$$

$$T_e = T_e(\Omega_e)$$

$$\text{엔진 토크} = \text{엔진 토크}(\text{엔진 회전율(rpm)})$$

$$P_e = T_e \omega_e$$

$$\text{엔진 출력(힘)} = \text{엔진 토크} * \text{엔진 각속도}$$

$$\omega_e = \frac{2\pi\Omega_e}{60}$$

$$\text{엔진 각속도} = 2\pi/60 * \text{엔진 회전율}$$



4 자동차 시뮬레이터 구현

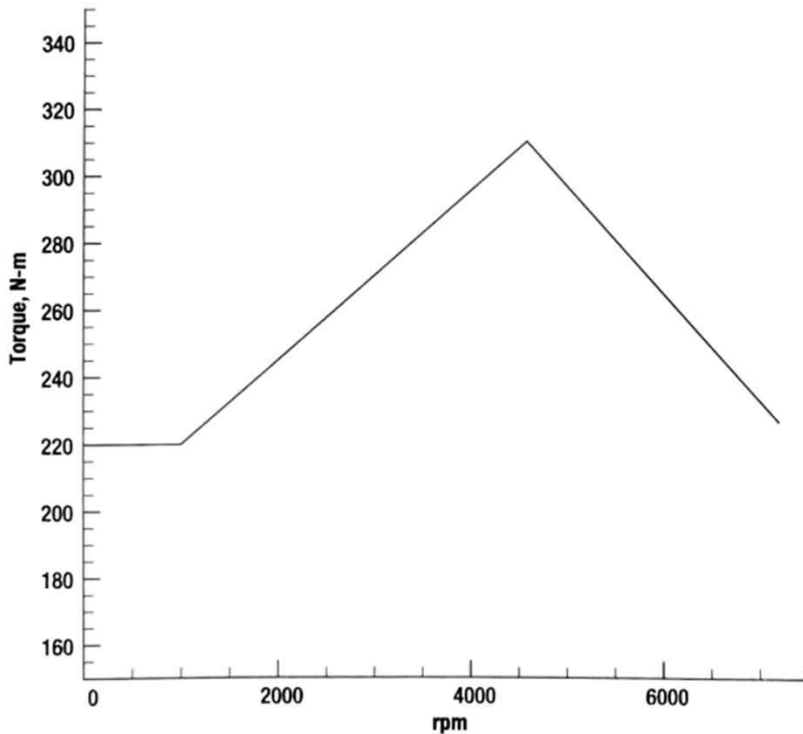


Figure 8-7. Simplified torque curve for the Porsche Boxster S

토크 곡선의 디지털화

$$\Omega_e(new) = \Omega_e(old) \frac{g_k(new)}{g_k(old)}$$

현재 엔진 회전율 = 이전 엔진 회전율 * 현재 기어비 / 이전 기어비

$$T_e = b\Omega_e + d$$

엔진 토크 = 선의 기울기 * 엔진 회전율 + d 의 방정식으로 나타낼 수 있음

$$v = r_w \omega_w = \frac{2\pi r_w \Omega_e}{60 g_k G}$$

속도 = 휠 반지름 * 휠 각속도

$$a = \frac{60 g_k^2 G^2 b v}{2\pi m r_w^2} + \frac{g_k G d}{m r_w} - \frac{1}{2} \frac{C_D \rho v^2 A}{m} - \mu_r g \cos \theta - g \sin \theta$$

G : final drive ratio(최종 감속 비)를 의미

$$a = \frac{dv}{dt} = c_1 v^2 + c_2 v + c_3$$

시간당 가속도 방정식의 치환 및 정리



4 자동차 시뮬레이터 구현

최고 속도와 그에 따른 가속도

$$v_{\max} = \frac{2\pi r_w \Omega_{redline}}{60 g_k G}$$

저단 기어에서 최대 속도는 차량의 순 가속도가 0에 도달하기 전에 차량의 최대 엔진 회전율 (redline)에 도달하기 때문에 최대 속도는 최대 엔진 회전율에 영향을 받는다.

$$a = c_1 v^2 + c_2 v + c_3 = 0$$

고단 기어에서는 공기항력에 제한(drag force $\propto v^2$)을 받기 때문에 차량의 최대 엔진 회전율에 도달하기 전에 가속도 0에 도달한다.

가속도가 0이 되는 지점: 휠에 가해지는 토크가 차량의 공기항력과 롤링마찰력에 의해 정확히 균형을 이루는 속도에 도달하는 지점



4 자동차 시뮬레이터 구현

$$v_{\max} = \frac{-c_2 \pm \sqrt{c_2^2 - 4c_1c_3}}{2c_1}$$

$C_d = 0.31$ = 항력계수

$\rho = 1.2$ = 공기밀도

$A = 1.94$ = 차량 전면 면적

$g_k = 0.84$ = 6단 기어의 기어비

$b = 0.032, d = 457.2$ = 토크 곡선

$m = 1393$ = 차량 무게

$r_w = 0.3186$ = 휠 반지름

$G = 3.44$ = final drive ratio

최고 속도의 계산

최고 속도 공식은 이와 같다.

중간 과정을 생략한다면 $C_1 = -2.59e - 4$, $C_2 = -0.018$, $C_3 = 2.83$ 이고

$$v_{max} = \frac{0.018 \pm \sqrt{0.018^2 + 4 * 2.59e - 4 * 2.83}}{-2 * 2.59e - 4} = 75.4 \frac{m}{s} = 271.5 \frac{km}{h}$$

이므로 차량의 최고 속도는 271.5 km/h 이다.



4 자동차 시뮬레이터 구현

브레이크 공식

$$T_{eb} = \mu_{eb} \frac{\Omega_e}{60}$$

엔진 브레이크

엔진 브레이크 중 토크 = 엔진 제동계수 * 엔진 회전율

$$a_b = -\frac{v_0^2}{2x} = -10.4 \times \frac{m}{s^2}$$

브레이크

브레이크 가속도 = - 초기 속도² / 2 * 제동 거리



4 자동차 시뮬레이터 구현

```
public class Car : DragProjectile
{
    private double muR; // 롤링 마찰계수
    private double omegaE; // 엔진 회전율(rpm)
    private double redline;
    private double finalDriveRatio;
    private double wheelRadius;
    private int gearNumber; // 현재 기어 번호
    private int numberOfGears; // 차량의 총 기어 수
    private string mode; // 등속, 가속, 제동 등
    private double[] gearRatio; // 기어 비

    // 생성자
    // DragProjectile 클래스 생성자를 호출하고 Car클래스 변수를 초기화
    public Car(double x, double y, double z, double vx,
        double vy, double vz, double time, double mass,
        double area, double density, double Cd, double redline,
        double finalDriveRatio, double wheelRadius,
        int numberOfGears) :
        base(x, y, z, vx, vy, vz, time, mass, area, density, Cd)
    {
        // 생성자에 전달된 값으로 변수 초기화
        this.redline = redline; // redline rpm
        this.finalDriveRatio = finalDriveRatio; // final drive ratio
        this.wheelRadius = wheelRadius; // wheel radius
        this.numberOfGears = numberOfGears; // number of gears

        // 기어 비 배열 초기화
        // 기어 비 배열 0번째 인덱스는 사용하지 않을 것이기 때문에
        // gearRatio 배열의 크기를 총 기어 비 +1로 설정하고 0번째 인덱스는 0으로 초기화
        // 모든 기어 비를 일단 1.0으로 초기화
        gearRatio = new double[numberOfGears + 1];
        gearRatio[0] = 0.0;
        for (int i = 1; i < numberOfGears + 1; ++i)
        {
            gearRatio[i] = 1.0;
        }

        // 모든 차량에 동일하게 설정할 변수들
        muR = 0.015; // 롤링 마찰 계수
        omegaE = 1000.0; // engine rpm
        gearNumber = 1; // 초기 기어
        mode = "accelerating"; // 초기 모드: 가속
    }
}
```

Car Class

해당 클래스에서는 앞 슬라이드의 차량의 물리 법칙에 관한 내용을 코드화 하였음

클래스 간 상속 관계

SimpleProjectile : ODE

DragProjectile : SimpleProjectile

Car : DragProjectile

BoxsterS : Car

GitHub: https://github.com/hhj3258/GamePhysics_RealisticCarPhysics



4 자동차 시뮬레이터 구현

```
// 변수 각각의 getter와 setter 선언
public double MuR
{
    get { return muR; }
}

public double OmegaE
{
    get { return omegaE; }
    set { omegaE = value; }
}

public double FinalDriveRatio
{
    get { return finalDriveRatio; }
}

// 기어 비 배열의 getter와 setter
public double GetGearRatio()
{
    return gearRatio[gearNumber];
}

public void SetGearRatio(int index, double value)
{
    gearRatio[index] = value;
}
```

```
// 기어 변속 메소드
public void ShiftGear(int shift)
{
    // 현재 기어가 가장 고단일 때 고단으로 변속 시 변화 없음
    if (shift + this.GearNumber > this.NumberOfGears)
    {
        return;
    }
    // 현재 기어가 가장 저단일 때 저단으로 변속 시 변화 없음
    else if (shift + this.GearNumber < 1)
    {
        return;
    }
    // 정상적인 기어 변속일 때
    // 기어 변경, 엔진 rpm 값 재계산
    else
    {
        double oldGearRatio = GetGearRatio();
        this.GearNumber = this.GearNumber + shift;
        double newGearRatio = GetGearRatio();
        this.OmegaE = this.OmegaE * newGearRatio / oldGearRatio;
    }

    return;
}
```



4 자동차 시뮬레이터 구현

```
// The GetRightHandSide() 메소드는 six first-order projectile ODEs의 오른쪽 항을 반환
// q[0] = vx = dxdt
// q[1] = x
// q[2] = vy = dydt
// q[3] = y
// q[4] = vz = dzdt
// q[5] = z
public override double[] GetRightHandSide(double s, double[] q,
double[] deltaQ, double ds, double qScale)
{
    double[] dQ = new double[6];
    double[] newQ = new double[6];

    // 종속 변수의 중간 값 계산
    for (int i = 0; i < 6; ++i)
    {
        newQ[i] = q[i] + qScale * deltaQ[i];
    }

    // torque curve 방정식
    // 엔진 회전율에 따라 세 부분으로 나누어서 정의
    double b, d;
    if (this.OmegaE <= 1000.0)
    {
        b = 0.0;
        d = 220.0;
    }
    else if (this.OmegaE < 4600.0)
    {
        b = 0.025;
        d = 195.0;
    }
    else
    {
        b = -0.032;
        d = 457.2;
    }

    // 속도의 중간값을 나타내기 위한 변수들 선언
    double vx = newQ[0];
    double vy = newQ[2];
    double vz = newQ[4];
    double v = Math.Sqrt(vx * vx + vy * vy + vz * vz) + 1.0e-8;
```

```
// 공기항력 변수 선언 및 계산
double density = this.Density;
double area = this.Area;
double cd = this.Cd;
double Fd = 0.5 * density * area * cd * v * v;

// 롤링 마찰 힘 계산
// G=-9.81
double mass = this.Mass;
double Fr = muR * mass * G;

// right-hand sides of the six ODEs 을 계산하면 속도의 중간 값이 나옴
// 차량의 가속도는 모드(가속, 제동, 등속)에 의해 달라짐.
// 제동 가속도는 -5.0 m/s^2 로 가정.
if (String.Equals(mode, "accelerating"))
{
    double c1 = -Fd / mass;
    double tmp = GetGearRatio() * finalDriveRatio / wheelRadius;
    double c2 = 60.0 * tmp * tmp * b * v / (2.0 * Math.PI * mass);
    double c3 = (tmp * d + Fr) / mass;
    dQ[0] = ds * (c1 + c2 + c3);
}
else if (String.Equals(mode, "braking"))
{
    // 속도가 양수일 때만 브레이크 작동
    if (newQ[0] > 0.1)
    {
        dQ[0] = ds * (-5.0);
    }
    else
    {
        dQ[0] = 0.0;
    }
}
else
{
    dQ[0] = 0.0;
}

dQ[1] = ds * newQ[0];
dQ[2] = 0.0;
dQ[3] = 0.0;
dQ[4] = 0.0;
dQ[5] = 0.0;

return dQ;
}
```



4 자동차 시뮬레이터 구현

```
public class BoxsterS : Car
{
    // BoxsterS 생성자는 Car 생성자를 호출한 후에 BoxsterS의 기어 비를 설정함.
    // 박스터S의 스펙
    // mass = 1393.0 kg (with 70 kg driver)
    // area = 1.94 m^2
    // Cd = 0.31
    // redline = 7200 rpm
    // finalDriveRatio = 3.44
    // wheelRadius = 0.3186
    // numberOfGears = 6;

    public BoxsterS(double x, double y, double z, double vx, double vy, double vz,
double time, double density) :
        base(x, y, z, vx, vy, vz, time, 1393.0, 1.94, density, 0.31, 7200.0, 3.44,
0.3186, 6)
    {

        // Set the gear ratios.
        SetGearRatio(1, 3.82);
        SetGearRatio(2, 2.20);
        SetGearRatio(3, 1.52);
        SetGearRatio(4, 1.22);
        SetGearRatio(5, 1.02);
        SetGearRatio(6, 0.84);
    }
}
```

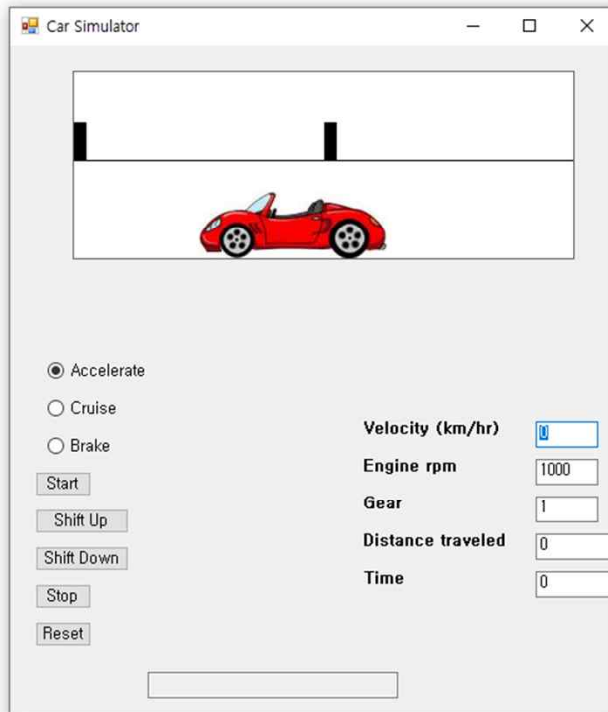
BoxsterS class

해당 클래스에서 특정 차량의 스펙을 일괄 입력한다.
해당 변수의 값들을 변경하여 다양한 차량의 스펙을
테스트 할 수 있다.



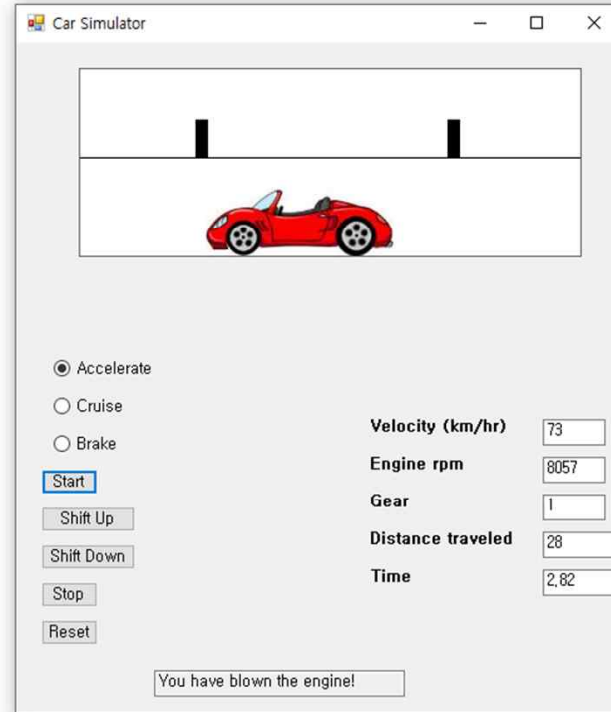
4 자동차 시뮬레이터 구현

Window Form을 이용한 자동차 시뮬레이터



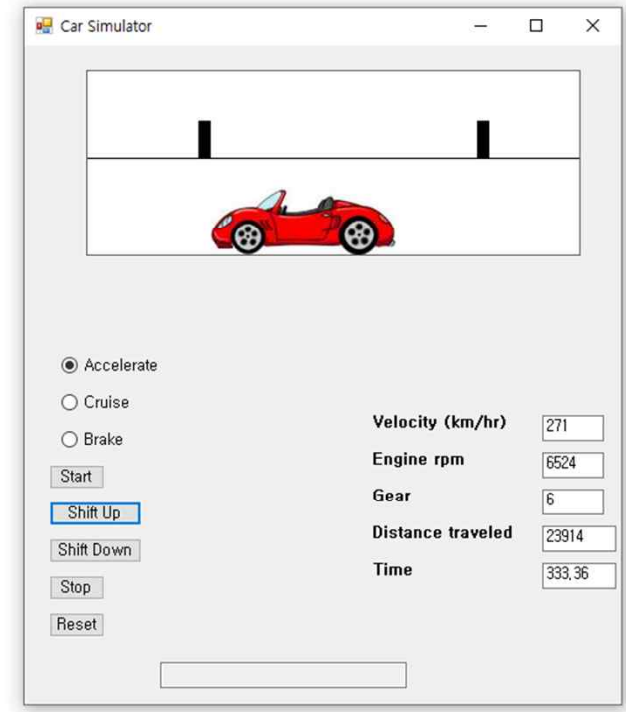
Car Simulator window showing initial settings. The car is a red sports car on a track. The controls include radio buttons for Accelerate (selected), Cruise, and Brake. The Velocity (km/hr) is 0, Engine rpm is 1000, Gear is 1, Distance traveled is 0, and Time is 0. Buttons include Start, Shift Up, Shift Down, Stop, and Reset.

초기 세팅



Car Simulator window showing engine warning. The car is a red sports car on a track. The controls include radio buttons for Accelerate (selected), Cruise, and Brake. The Velocity (km/hr) is 73, Engine rpm is 8057, Gear is 1, Distance traveled is 28, and Time is 2.82. Buttons include Start, Shift Up, Shift Down, Stop, and Reset. A message box at the bottom says "You have blown the engine!".

1단 기어에서 최고 rpm 도달 시
하단 경고 메시지 출력과 함께 정지



Car Simulator window showing maximum speed. The car is a red sports car on a track. The controls include radio buttons for Accelerate (selected), Cruise, and Brake. The Velocity (km/hr) is 271, Engine rpm is 6524, Gear is 6, Distance traveled is 23914, and Time is 333.36. Buttons include Start, Shift Up, Shift Down, Stop, and Reset.

최고 기어에서의 이론상
최대 속도(271 km/h)



4 자동차 시뮬레이터 구현

Window Form으로 구현한 자동차 시뮬레이터

Car Simulator

☒ Accelerate
☐ Cruise
☐ Brake

Start
Shift Up
Shift Down
Stop
Reset

Velocity (km/hr) 0
Engine rpm 1000
Gear 1
Distance traveled 0
Time 0

Accelerate

Car Simulator

☒ Accelerate
☐ Cruise
☐ Brake

Start
Shift Up
Shift Down
Stop
Reset

Velocity (km/hr) 254
Engine rpm 7424
Gear 5
Distance traveled 2115
Time 40.14

Warning: Exceeding redline rpm

Cruise

Car Simulator

☐ Accelerate
☒ Cruise
☐ Brake

Start
Shift Up
Shift Down
Stop
Reset

Velocity (km/hr) 262
Engine rpm 6311
Gear 6
Distance traveled 3709
Time 62.16

Brake

Car Simulator

☐ Accelerate
☐ Cruise
☒ Brake

Start
Shift Up
Shift Down
Stop
Reset

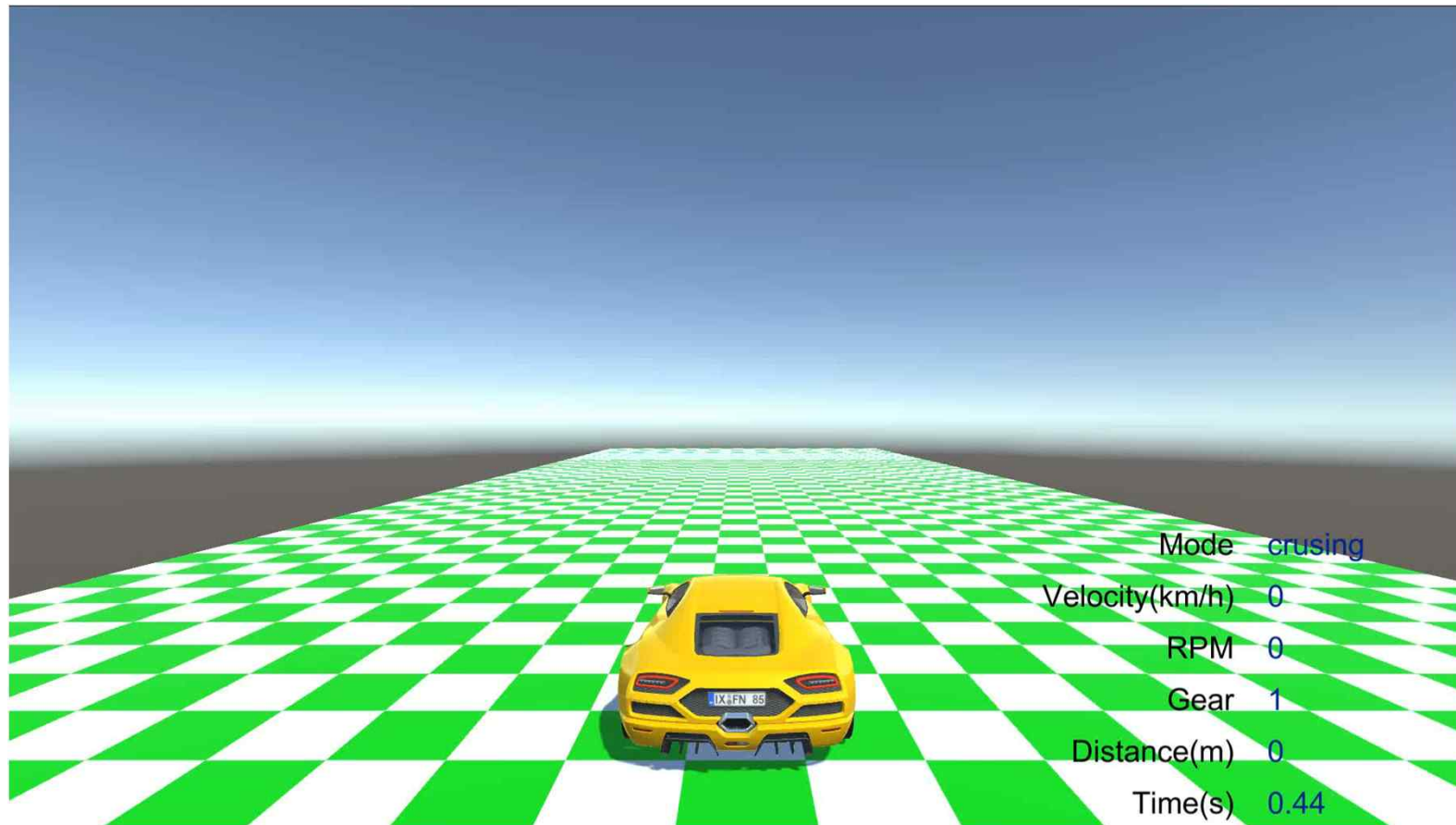
Velocity (km/hr) 0
Engine rpm 2
Gear 6
Distance traveled 4270
Time 77.22

수동 조작



4 자동차 시뮬레이터 구현

Unity로 구현한 자동차 시뮬레이터



Reference

- 01 Physics for Game Programmers – Grant Palmer
- 02 게임엔진물리학 Game Engine Physics - 이종완
- 03 AssetStore: HQ Racing Car Model No.1203
- <https://assetstore.unity.com/packages/3d/vehicles/land/hq-racing-car-model-no-1203-139221>



감사합니다

