FYS2130 - Project

Candidate - 15266

April 2018

Problems

Problem 1

We are asked to solve the equation

$$ma(t) + kx(t) = 0, \tag{1}$$

numerically using the Runge-Kutta 4 method where we have written the code ourself. We start by setting up an expression we can feed into our function, which is basicly just to solve for a which yields

$$a(t) = -\frac{k}{m}x(t). \tag{2}$$

Using the constants and initial conditions given in the problem, $x(0) = 1$ and $v(0) = 0$ produces an elipse as **??** shows. **??** shows plots of the energy as the particle moves, whichs shows the energy is constant as expected. Because the total energy is constant, we thus have have

$$E_{tot} = E_k + E_p = \frac{1}{2}mv^2 + \frac{1}{2}kx^2 = C, \tag{3}$$

where C is a constant. This is the equation for an elipse. Changing the initial conditions only will always produce an elipse.

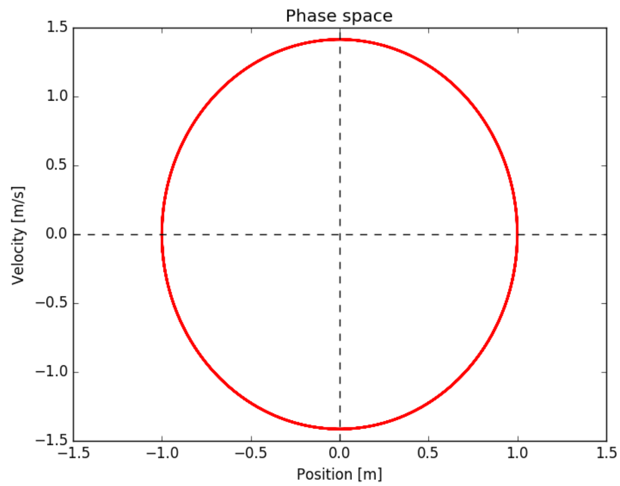

Figure 1: Phase diagram of a harmonic oscilator.

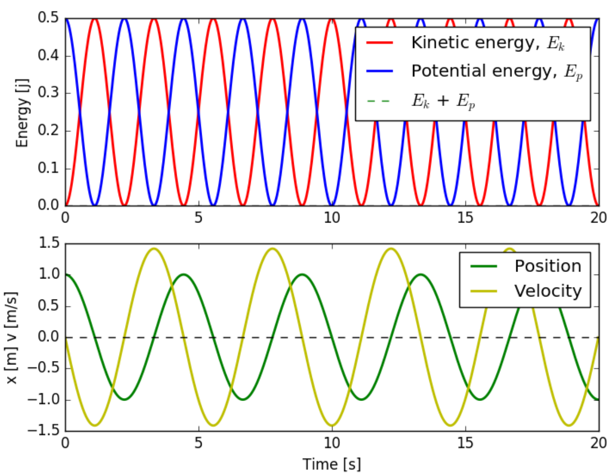

Figure 2: Energy for harmonic oscillator.

Problem 2

We will now include damping which yields

$$a(t) = -\frac{k}{m}x(t) - \frac{b}{m}v(x). \tag{4}$$

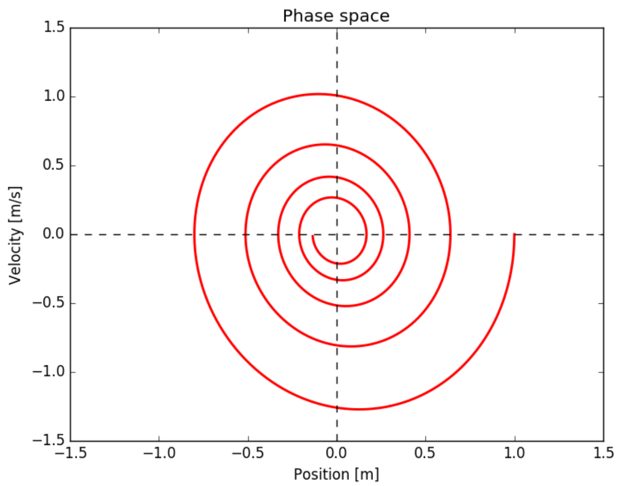

Figure 3: Phase diagram of a harmonic oscilator.

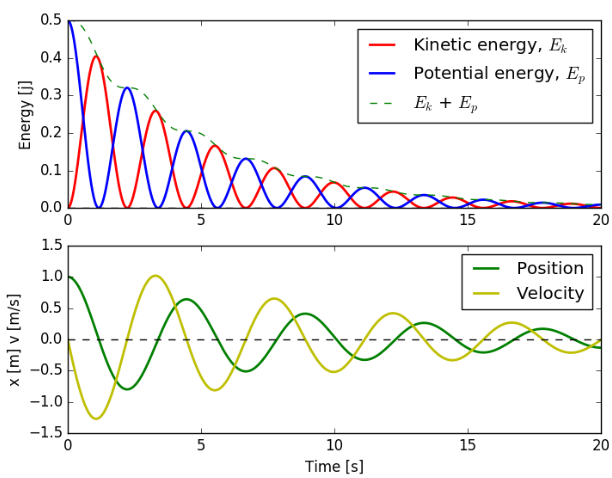

Figure 4: Energy for damped harmonic oscillator.

Problem 3

We are asked to solve the following differential equation

$$m\ddot{x}(t) + kx(t) = F_D \cos \omega_D t, \tag{5}$$

analytically. This type of equation is called *the Periodically Forced Harmonic Oscillator*. $b = 0$, which means this is a undamped system. This is a second order nonlinear, nonhomogeneous differential equation. A differential equation on this form is solved by adding the solution of the homogeneous and the solution of the particular solution. By the method of undetermined coefficients, the general solution is

$$x(t) = c_1 \cos(\omega_0 t) + c_2 \sin(\omega_0 t) + \frac{F_D}{m(\omega_D^2 - \omega_0^2)} \cos(\omega_D t), \tag{6}$$

which yields

$$\dot{x}(t) = -c_1\omega_0 \sin(\omega_0 t) + c_2\omega_0 \sin(\omega_0 t) - \omega_D \frac{F_D}{m(\omega_D^2 - \omega_0^2)} \sin(\omega_D t), \tag{7}$$

where $\omega_0 = \sqrt{k/m}$ is the *natural frequency* of the undamped harmonic oscillator. We need to solve this for the initial conditions $x(0) = 2.0$ and $\dot{x}(0) = 0.0$, namely

$$x(0) = c_1 + \frac{F_D}{m(\omega_D^2 - \omega_0^2)} = 2.0 \rightarrow c_1 = 2.0 - \frac{F_D}{m(\omega_D^2 - \omega_0^2)} \text{ and} \tag{8}$$

$$\dot{x}(0) = c_2\omega_0 \cos(\omega_0 t) = 0.0 \rightarrow c_2 = 0.0. \tag{9}$$

Using the initial conditions given, we get the following plots

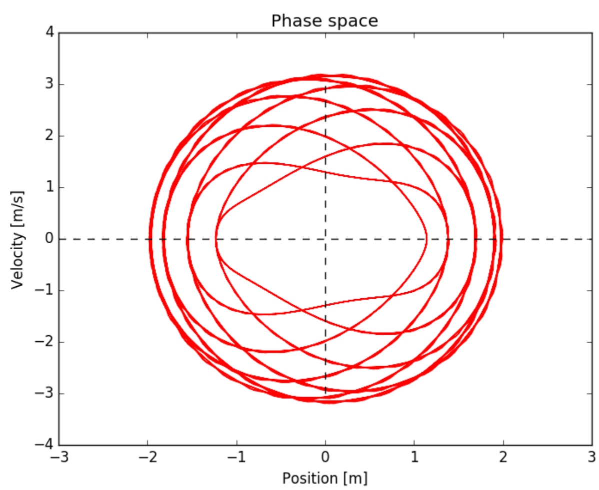

Figure 5: Phase diagram of a harmonic oscilator.

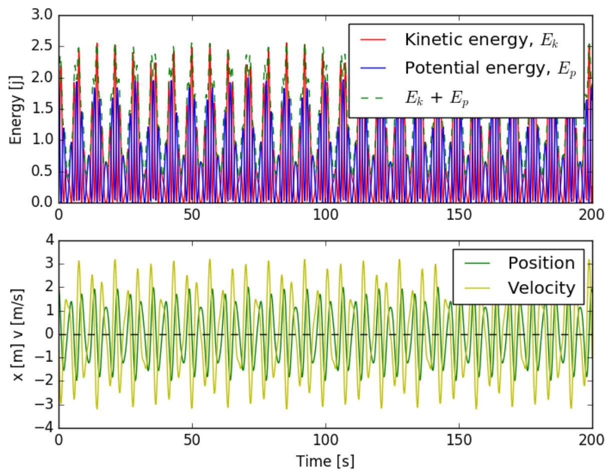

Figure 6: Energy for damped harmonic oscillator.

Problem 4

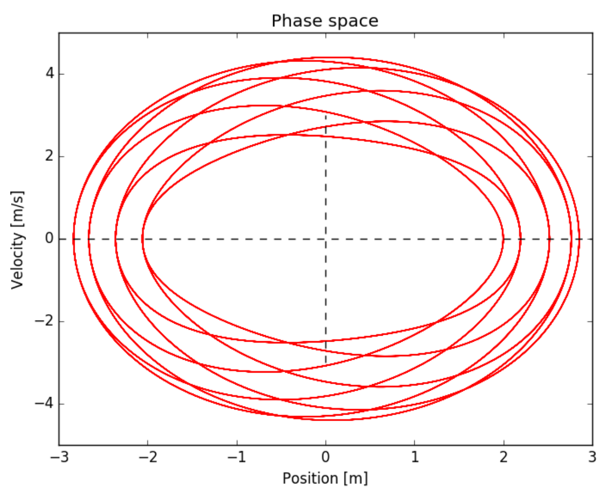

Figure 7: Phase diagram of a harmonic oscilator.

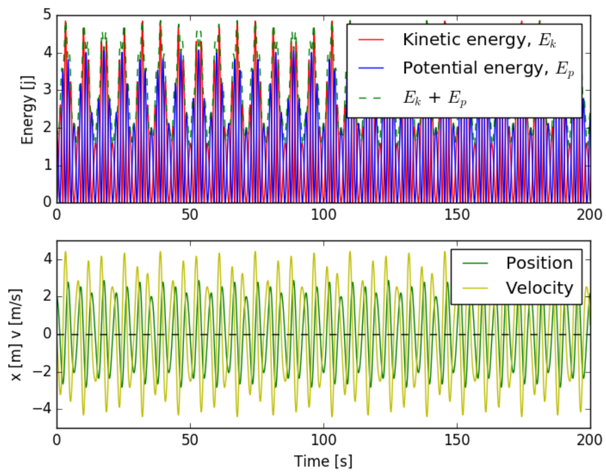

Figure 8: Energy for damped harmonic oscillator.

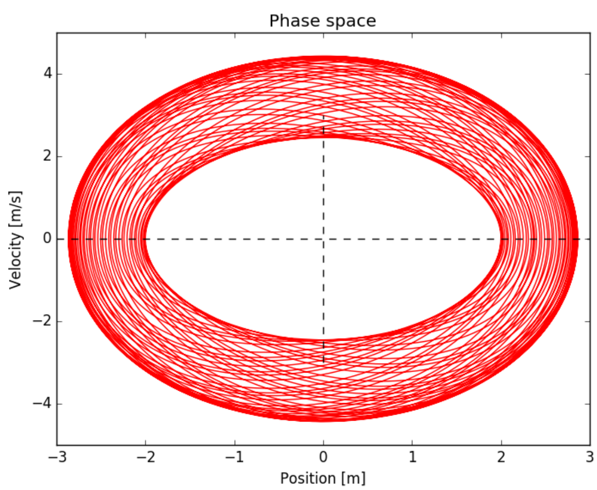

Figure 9: Phase diagram of a harmonic oscilator.

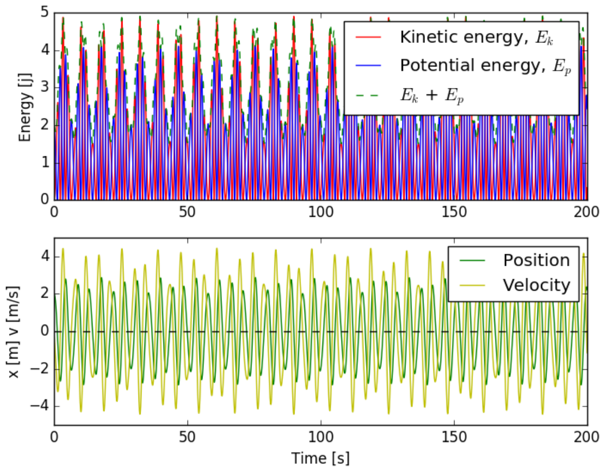

Figure 10: Energy for damped harmonic oscillator.

Problem 5

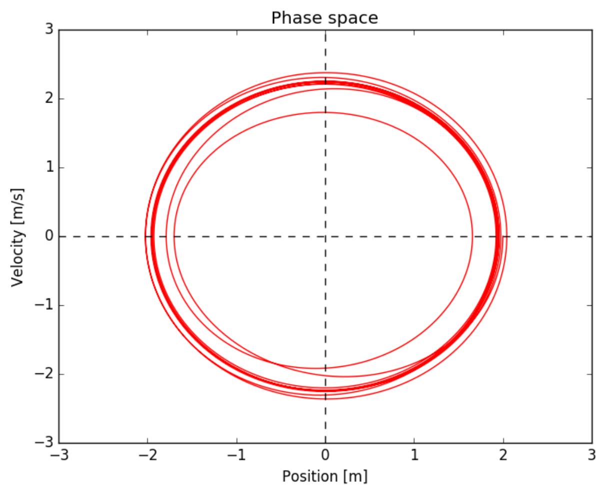

Figure 11: Phase diagram of a harmonic oscilator.

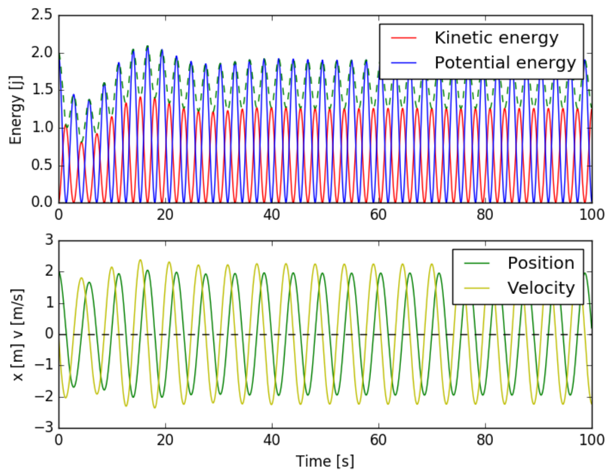

Figure 12: Energy for damped harmonic oscillator.

Problem 6

Problem 7

Problem 8

Problem 9

Appendix

Python code

```
import sys
import numpy as np
import matplotlib.pyplot as plt


class Solver():
    def __init__(self, time, dt):
        """
        Initialize solver. Calculated number of steps and
    initialize arrays.
        """
        self.time = time
        self.dt = dt
        self.N = int(self.time/self.dt)

        self.x = np.zeros(self.N)
        self.v = np.zeros(self.N)
        self.t = np.zeros(self.N)
```

```python
        self.m = np.zeros(self.N)

        self.dm = 1.0


    def set_initial_conditions(self, x0, v0, m0):
        """
        Set initial conditions.
        """
        self.x[0] = x0
        self.v[0] = v0
        self.m[0] = m0


    def set_time(self, time):
        """
        Set new time. Because the number of steps depends on the
    time. New arrays will also be initialized.
        """
        self.time = time
        self.N = int(self.time/self.dt)

        self.x = np.zeros(self.N)
        self.v = np.zeros(self.N)
        self.t = np.zeros(self.N)
        self.m = np.zeros(self.N)


    def set_dt(self, dt):
        """
        Set delta time, dt
        """
        self.dt = dt


    def set_dm(self, dm):
        """
        Set delta mass, dm
        """
        self.dm = dm


    def set_diff_eq(self, f):
        """
        Set the differential equation to be solved.
        """
        self.diff_eq = f;


    def rk4_step(self, x0, v0, t0, mi):
        """
        Solves one step using the Runge Kutta method
        """
        dt = self.dt

        a1 = self.diff_eq(x0, v0, t0, mi)
        v1 = v0
```

```
        x_half_1 = x0 + v1 * dt/2.0
        v_half_1 = v0 + a1 * dt/2.0

        a2 = self.diff_eq(x_half_1, v_half_1, t0+dt/2.0, mi)
        v2 = v_half_1

        x_half_2 = x0 + v2 * dt/2.0
        v_half_2 = v0 + a2 * dt/2.0

        a3 = self.diff_eq(x_half_2, v_half_2, t0+dt/2.0, mi)
        v3 = v_half_2

        x_end = x0 + v3 * dt
        v_end = v0 + a3 * dt

        a4 = self.diff_eq(x_end, v_end, t0 + dt, mi)
        v4 = v_end

        a_middle = 1.0/6.0 * (a1 + 2*a2 + 2*a3 + a4)
        v_middle = 1.0/6.0 * (v1 + 2*v2 + 2*v3 + v4)

        x_end = x0 + v_middle * dt
        v_end = v0 + a_middle * dt

        return x_end, v_end


    def solve(self):
        """
        Solver for the first 5 programming problems
        """
        x = self.x; v = self.v; t = self.t; N = self.N; dt = self.
    dt; m = self.m; dm = self.dm
        for i in range(N-1):
            [x[i+1], v[i+1]] = self.rk4_step(x[i], v[i], t[i], m[i]
    )
            t[i+1] = t[i] + dt
            m[i+1] = m[i] + dm

        return x, v, t, m


    def solve_2(self):
        """
        Solver for the last three programming problems
        """
        x = self.x; v = self.v; t = self.t; N = self.N; dt = self.
    dt; m = self.m; dm = self.dm
        for i in range(N-1):
            [x[i+1], v[i+1]] = self.rk4_step(x[i], v[i], t[i], m[i]
    )
            t[i+1] = t[i] + dt
            m[i+1] = m[i] + dm
            if( True ):
                m[i+1] = m[i] - dm
```

```python
        return x, v, t, m


def show_and_save_plots(i, x=0, v=0, t=0, k=1.0, m=1.0, ps=0, steps
    =20, lw=2):
    """
    Just a wrapper functions for plotting the PHASE SPACE, ENERGIES
     and POSITION/VELOCITIES.
    """
    # Plot phase space
    plt.plot(x, v, 'r', linewidth=lw)
    plt.plot([-ps,ps],[0,0],'--k')
    plt.plot([0,0],[-ps,ps],'--k')
    plt.title('Phase space')
    plt.ylabel('Velocity [m/s]')
    plt.xlabel('Position [m]')

    plt.savefig('problem_%s_1.png' % (i))

    plt.show()

    #
    Ek = 0.5*m*(v**2)
    Ep = 0.5*k*(x**2)

    #
    plt.subplot(2,1,1)
    plt.plot(t, Ek, 'r', linewidth=lw)
    plt.plot(t, Ep, 'b', linewidth=lw)
    plt.plot(t, Ek+Ep, '--g', linewidth=1)
    plt.plot([0,steps], [0,0], '--k')
    plt.legend(['Kinetic energy, $E_k$', 'Potential energy, $E_p$', '$E_k +
     E_p$'])
    plt.ylabel('Energy [j]')

    #
    plt.subplot(2,1,2)
    plt.plot(t, x, 'g', linewidth=lw)
    plt.plot(t, v, 'y', linewidth=lw)
    plt.plot([0,steps], [0,0], '--k')
    plt.legend(['Position', 'Velocity'])
    plt.ylabel('x [m] v [m/s]')
    plt.xlabel('Time [s]')

    plt.savefig('problem_%s_2.png' % (i))

    plt.show()


#
def diff_eq_1(x, v, t, m):
    """
    Differential equation for problem 1
    """
    m = 0.5      # [kg]
    k = 1.0      #
```

```
    a = -(1./.5)*x

    return a


#
def diff_eq_2(x, v, t, m):
    """
    Differential equation for problem 2
    """
    m = 0.5      # [kg]
    k = 1.0      # [N/m]
    b = 0.1      # [kg/s]

    a = -(k*x + b*v)/m

    return a


#
def prob_3_eq(k, m, steps):
    """
    Plots the analytical function found in problem 3
    """
    F_D = 0.7
    k = 1.0
    m = 0.5
    omega_0 = np.sqrt(k/m)
    omega_D = 13.0/8.0*omega_0
    t = np.linspace(0, steps, 1000)
    c1 = 2.0 - F_D/(m*(omega_D**2 - omega_0**2))
    c2 = 0.0

    x = c1*np.cos(omega_0*t) + c2*np.sin(omega_0*t) + F_D/(m*(
    omega_D**2 - omega_0**2))*np.cos(omega_D*t)

    v = -c1*omega_0*np.sin(omega_0*t) + c2*omega_0*np.cos(omega_0*t
    ) - omega_D*(F_D/(m*(omega_D**2 - omega_0**2)))*np.sin(omega_D*
    t)

    return x, v, t


#
def diff_eq_4_1(x, v, t, m):
    """
    Differential equation for problem 4
    """
    m = 0.5      # [kg]
    k = 1.0      # [N/m]
    F_D = 0.7    # N
    omega_0 = np.sqrt(k/m)
    omega_D = 13.0/8.0*omega_0

    a = (F_D*np.cos(omega_D*t)-k*x)/m

    return a
```

```
#
def diff_eq_4_2(x, v, t, m):
    """
    Differential equation for problem 4
    """
    m = 0.5       # [kg]
    k = 1.0       # [N/m]
    F_D = 0.7     # N
    omega_0 = np.sqrt(k/m)
    omega_D = 2.0/(np.sqrt(5) - 1)*omega_0

    a = (F_D*np.cos(omega_D*t)-k*x)/m

    return a


#
def diff_eq_5(x, v, t, m):
    """
    Differential equation for problem 5
    """
    m = 0.5       # []
    k = 1.0       # []
    b = 0.1       # []
    F_D = 0.7     # N
    w_0 = np.sqrt(k/m)
    omega_D = 13.0/(8.0*w_0)

    a = (F_D*np.cos(omega_D*t)-k*x-b*v)/m

    return a


#
def diff_eq_6(x, v, t, m):
    """
    Differential equation for problem 6
    """
    k = 0.475         # []
    b = 0.001         # []
    g = 9.81          #

    a = -(b*v + k*x + g)/m

    return a


#
def main(problem_to_solve):
    solver = Solver(20.0, 1e-2)

    if problem_to_solve == 1:
        #
        #solver.set_time(1.0)
        solver.set_diff_eq(diff_eq_1)
```

```
    solver.set_initial_conditions(1.0, 0.0, 0.5)
    [x, v, t, m] = solver.solve();

    show_and_save_plots(problem_to_solve, x, v, t, 1.0, m[0],
1.5, 20.0, 2)

elif problem_to_solve == 2:
    #
    #solver.set_time(1.0)
    solver.set_diff_eq(diff_eq_2)
    solver.set_initial_conditions(1.0, 0.0, 0.5)
    [x, v, t, m] = solver.solve()

    show_and_save_plots(problem_to_solve, x, v, t, 1.0, m[0],
1.5, 20.0, 2)

elif problem_to_solve == 3:
    #
    [x, v, t] = prob_3_eq(1.0, 0.5, 200)
    show_and_save_plots(problem_to_solve, x, v, t, 1.0, 0.5,
3.0, 200.0, 1)

elif problem_to_solve == 4:
    # Solve for omega_D =
    solver.set_diff_eq(diff_eq_4_1)
    solver.set_time(200.0)
    solver.set_initial_conditions(2.0, 0.0, 0.5)
    [x, v, t, m] = solver.solve()

    show_and_save_plots('4_1', x, v, t, 1.0, m[0], 3.0, 200.0,
1)

    # Solve for omega_D =
    solver.set_diff_eq(diff_eq_4_2)
    solver.set_time(200.0)
    solver.set_initial_conditions(2.0, 0.0, 0.5)
    [x, v, t, m] = solver.solve()

    show_and_save_plots('4_2', x, v, t, 1.0, m[0], 3.0, 200.0,
1)

elif problem_to_solve == 5:
    #
    solver.set_diff_eq(diff_eq_5)
    solver.set_time(100.0)
    solver.set_initial_conditions(2.0, 0.0, 0.5)
    [x, v, t, m] = solver.solve();

    show_and_save_plots(problem_to_solve, x, v, t, 1.0, m[0],
3.0, 100.0, 1)

elif problem_to_solve == 6:
    solver.set_diff_eq(diff_eq_6)
    solver.set_time(3.0)
    solver.set_dt(1e-4)
    solver.set_dm(0.00055)
    solver.set_initial_conditions(0.001, 0.001, 0.00001)
```

```python
        [x, v, t, m] = solver.solve();

        show_and_save_plots(problem_to_solve, x, v, t, 1.0, m[0],
    3.0, 100.0, 1)

    else:
        print "Please input a valid problem numer: [1,2,3,4,5,6]"


if __name__ == "__main__":
    if len(sys.argv) <= 1:
        main(1)
        main(2)
        main(3)
        main(4)
        main(5)
        main(6)
    else:
        main(int(sys.argv[1]))
```