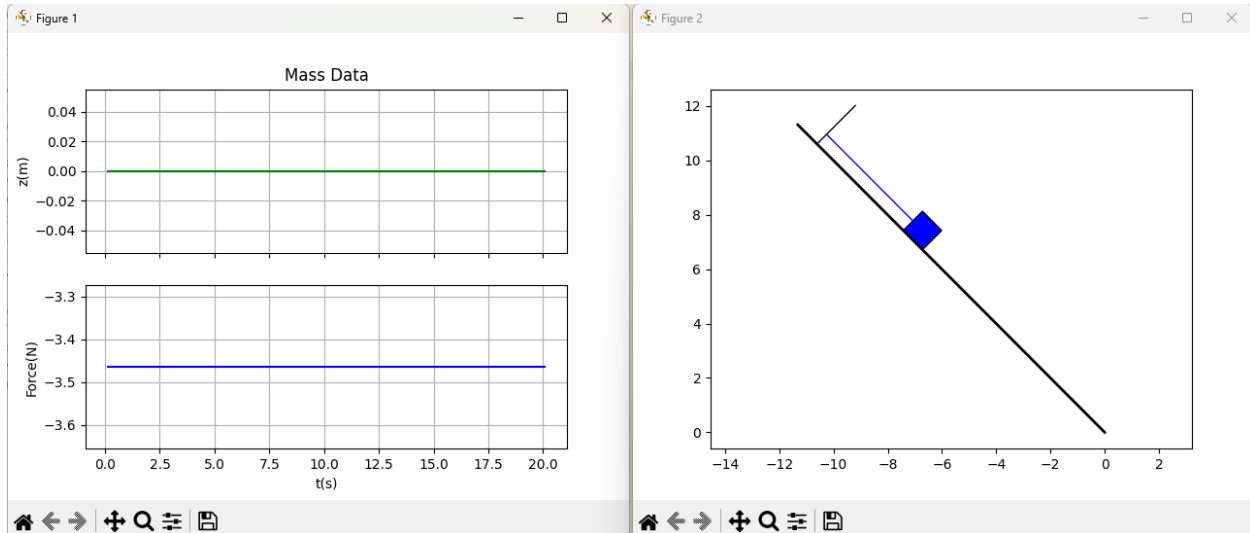


Part 2.2



Part 2.6:

```
import numpy as np
import massParam as P
class massDynamics:
    def __init__(self, alpha=0.0):
        self.state = np.array([
            [0.0], # initial position
            [0.0] # initial velocity
        ])
        self.g = P.g
        self.theta = 45 * np.pi / 180 * (1.+alpha*(2.*np.random.rand()-1.))
        self.m = P.m * (1.+alpha*(2.*np.random.rand()-1.))
        self.Fmax = P.F_max
        self.k1 = P.k1 * (1.+alpha*(2.*np.random.rand()-1.))
        self.k2 = P.k2 * (1.+alpha*(2.*np.random.rand()-1.))
        self.b = P.b * (1.+alpha*(2.*np.random.rand()-1.))
        self.Ts = P.Ts
        self.force_limit = P.F_max
    def update(self, u):
        u = self.saturate(u, self.force_limit)
        self.rk4_step(u)
        y = self.h()
        return y
    def f(self, state, F):
        # Return xdot = f(x,u), the system state update equations
        # re-label states for readability
        z = state.item(0)
        zdot = state.item(1)
        xdot = np.array([
            [zdot],
```

```

        [1/self.m * (F - self.b*zdot - self.k1*z - self.k2*z**3 +
self.m*self.g/np.sqrt(2.0))]
    ])
    return xdot
def h(self):
    # return the output equations
    # could also use input u if needed
    z = self.state.item(0)
    y = np.array([
        [z],
    ])
    return y
def rk4_step(self, u):
    # Integrate ODE using Runge-Kutta RK4 algorithm
    F1 = self.f(self.state, u)
    F2 = self.f(self.state + self.Ts / 2 * F1, u)
    F3 = self.f(self.state + self.Ts / 2 * F2, u)
    F4 = self.f(self.state + self.Ts * F3, u)
    self.state += self.Ts / 6 * (F1 + 2 * F2 + 2 * F3 + F4)
def saturate(self, u, limit):
    if abs(u) > limit:
        u = limit * np.sign(u)
    return u

```

Part 3.5

