```python
import numpy as np
import massParam as P

class controllerObsv:
    def __init__(self):
        self.observer_state = np.array([[0.0], [0.0]])

        self.K = P.K
        self.ki = P.ki2
        self.L = P.L

        self.A = P.A
        self.B = P.B
        self.C = P.C

        self.limit = P.F_max
        self.F_e = P.F_e
        self.Ts = P.Ts

        self.F_d1 = 0.0
        self.integrator = 0.0
        self.error_d1 = 0.0

    def update(self, z_r, y):
        x_hat = self.update_observer(y)
        z_hat = x_hat[0][0]
        error = z_r - z_hat

        # For state space, very little need to do anti-windup
        self.integrateError(error)
        self.error_d1 = error

        force_tilde = -self.K @ x_hat - self.ki * self.integrator
        force = self.saturate(self.F_e + force_tilde[0])
        self.F_d1 = force

        return force, x_hat

    def update_observer(self, y):
        # update the observer using RK4 integration
        F1 = self.observer_f(self.observer_state, y)
        F2 = self.observer_f(self.observer_state + self.Ts / 2 * F1, y)
        F3 = self.observer_f(self.observer_state + self.Ts / 2 * F2, y)
        F4 = self.observer_f(self.observer_state + self.Ts * F3, y)
        self.observer_state = self.observer_state + self.Ts / 6 * (F1 + 2 *
F2 + 2 * F3 + F4)
        return self.observer_state

    def observer_f(self, x_hat, y):
        # x_hat_dot = A*(x_hat-xe) + B*(u-ue) + L(y-C*x_hat)
        x_hat_dot = self.A @ x_hat + self.B * (self.F_d1 - self.F_e) + self.
L * (y - self.C @ x_hat)
        return x_hat_dot

    def integrateError(self, error):
        self.integrator = self.integrator + (P.Ts / 2.0) * (error + self.
error_d1)

    def saturate(self,u):
        if abs(u) > self.limit:
            u = self.limit*np.sign(u)
        return u
```