```python
import numpy as np
from control import c2d, tf
import massParam as P
import loopshape_mass as L


class controllerLoop:
    def __init__(self, method="state_space"):
        if method == "state_space":
            self.prefilter = transferFunction(L.F_num, L.F_den, P.Ts)
            self.control = transferFunction(L.C_num, L.C_den, P.Ts)
        elif method == "digital_filter":
            self.prefilter = digitalFilter(L.F_num, L.F_den, P.Ts)
            self.control = digitalFilter(L.C_num, L.C_den, P.Ts)
        self.method = method

    def update(self, z_r, y):
        z = y[0][0]
        # prefilter the reference
        z_r_filtered = self.prefilter.update(z_r)
        # filtered error signal
        error = z_r_filtered - z
        # update controller
        force_tilde = self.control.update(error)
        # compute total torque
        force = saturate(P.F_e + force_tilde, P.F_max)
        return force


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


class transferFunction:
    def __init__(self, num, den, Ts):
        # expects num and den to be numpy arrays of
        # shape (1,m+1) and (1,n+1)
        m = num.shape[1]
        n = den.shape[1]
        # set initial conditions
        self.state = np.zeros((n - 1, 1))
        self.Ts = Ts
        # make the leading coef of den == 1
        if den.item(0) != 1:
            tmp = den.item(0)
            num = num / tmp
            den = den / tmp
        self.num = num
```

```
        self.den = den
        # set up state space equations in control canonic form
        self.A = np.zeros((n - 1, n - 1))
        self.B = np.zeros((n - 1, 1))
        self.C = np.zeros((1, n - 1))
        for i in range(0, n - 1):
            self.A[0][i] = - den.item(i + 1)
        for i in range(1, n - 1):
            self.A[i][i - 1] = 1.0
        if n > 1:
            self.B[0][0] = 1.0
        if m == n:
            self.D = num.item(0)
            for i in range(0, n - 1):
                self.C[0][i] = num.item(i + 1) \
                               - num.item(0) * den.item(i + 1)
        else:
            self.D = 0.0
            for i in range(n - m - 1, n - 1):
                self.C[0][i] = num.item(i)

    def update(self, u):
        x = self.rk4(u)
        y = self.C @ x + self.D * u
        return y.item(0)

    def f(self, state, u):
        xdot = self.A @ state + self.B * u
        return xdot

    def rk4(self, u):
        # Integrate ODE using Runge-Kutta 4 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)
        return self.state


class digitalFilter:
    def __init__(self, num, den, Ts):
        self.Ts = Ts
        sys = tf(num[0], den[0])
        sys_d = c2d(sys, Ts, method='tustin')
        self.den_d = sys_d.den[0][0]
        self.num_d = sys_d.num[0][0]
        self.prev_filt_output = np.zeros(len(self.num_d) - 1)
        self.prev_filt_input = np.zeros(len(self.den_d))
```

```
    def update(self, u):
        # update vector with filter inputs (u)
        self.prev_filt_input = np.hstack(([u], self.prev_filt_input[0:-1]))
        # use filter coefficients to calculate new output (y)
        y = self.num_d @ self.prev_filt_input - self.den_d[1:] @ self.
prev_filt_output
        # update vector with filter outputs
        self.prev_filt_output = np.hstack(([y], self.prev_filt_output[0:-1
]))
        return y
```