```python
# MRAC Adaptive controller

# Import libraries
import numpy as np
from base_controller import BaseController
from lqr_solver import dlqr, lqr
from scipy.linalg import solve_continuous_lyapunov, solve_lyapunov, solve_discrete_lyapunov
from math import cos, sin
import numpy as np
from scipy import signal

class AdaptiveController(BaseController):
    """ The LQR controller class.

    """

    def __init__(self, robot, lossOfThurst):
        """ MRAC adaptive controller __init__ method.

        Initialize parameters here.

        Args:
            robot (webots controller object): Controller for the drone.
            lossOfThrust (float): percent lost of thrust.

        """

        super().__init__(robot, lossOfThurst)

        # define integral error
        self.int_e1 = 0
        self.int_e2 = 0
        self.int_e3 = 0
        self.int_e4 = 0

        # flag for initializing adaptive controller
        self.have_initialized_adaptive = False
```

```python

        # reference model
        self.x_m = None

        # baseline LQR controller gain
        self.Kbl = None

        # Saved matrix for adaptive law computation
        self.A_d = None
        self.B_d = None
        self.Bc_d = None

        self.B = None
        self.Gamma = None
        self.P = None

        # adaptive gain
        self.K_ad = None

    def initializeGainMatrix(self):
        """ Calculate the LQR gain matrix and matrices for adaptive controller.

        """

        # ---------------|LQR Controller|-------------------------
        # Use the results of linearization to create a state-space model

        # Given parameters
        n_p = 12  # number of states
        m = 4  # number of integral error terms

        # robot parameter
        self.m = 0.4
        self.d1x = 0.1122
        self.d1y = 0.1515
        self.d2x = 0.11709
        self.d2y = 0.128
        self.Ix = 0.000913855
```

```python
        self.Iy = 0.00236242
        self.Iz = 0.00279965

        # constants
        self.g = 9.81
        self.ct = 0.00026
        self.ctau = 5.2e-06
        self.U1_max = 10
        self.pi = 3.1415926535

        # ----------------- Your Code Here ----------------- #
        # Compute the continuous A, B, Bc, C, D and
        # discretized A_d, B_d, Bc_d, C_d, D_d, for the computation of LQR gain

        # Matrix A logic
        # Initialize A matrix with zeros ( 16 x 16 )
        A = np.zeros((n_p + m, n_p + m))
        A[0, 6] = 1; A[1, 7] = 1; A[2, 8] = 1; A[3, 9] = 1; A[4, 10] = 1; A[5, 11] = 1
        A[6, 4] = self.g; A[7, 3] = -self.g
        A[12, 0] = 1; A[13, 1] = 1; A[14, 2] = 1; A[15, 5] = 1
        # A[12, 0] = 1; A[12, 12] = -1; A[13, 1] = 1; A[13, 13] = -1; A[14, 2] = 1; A[14, 14] = -1; A[15, 5] = 1; A[15, 15] = -1

        # Matrix B logic
        # Initialize B matrix with zeros ( 16 x 4 )
        B = np.zeros((n_p + m, m))
        B[8, 0] = 1 / self.m; B[9, 1] = 1 / self.Ix; B[10, 2] = 1 / self.Iy; B[11, 3] = 1 / self.Iz

        # Matrix Bc logic
        # Initialize Bc matrix with zeros ( 16  x 4 )
        Bc = np.zeros((n_p + m, m))
```

```
        Bc[12, 0] = -1; Bc[13, 1] = -1; Bc[14, 2] = -1; Bc[15, 3] = -1

        # Combine B and Bc into one matrix
        combined_B = np.hstack((B, Bc))

        # Matrix C logic
        # Initialize C matrix with zeros ( 4 x 16 )
        C = np.zeros((m, n_p + m))
        C[0, 0] = 1; C[1, 1] = 1; C[2, 2] = 1; C[3, 3] = 1

        # Matrix D logic
        # Zero matrix ( 4 x 4 )
        D = np.zeros((m, m))

        # Discretize the system
        sys_discrete = signal.cont2discrete((A, combined_B, C, D), self.delT, method='zoh')

        # Extract A_d, B_d, Bc_d, C_d, D_d
        A_d = sys_discrete[0]
        B_d = sys_discrete[1][:, :m]  # only take the first 4 columns
        Bc_d = sys_discrete[1][:, m:]  # only take the last 4 columns
        C_d = sys_discrete[2]
        D_d = sys_discrete[3]


        # ----------------- Your Code Ends Here ----------------- #

        # Record the matrix for later use
        self.B = B  # continuous version of B
        self.A_d = A_d  # discrete version of A
        self.B_d = B_d  # discrete version of B
        self.Bc_d = Bc_d  # discrete version of Bc

        # -----------------    Example code
```

```
    ----------------- #
        # max_pos = 15.0
        # max_ang = 0.2 * self.pi
        # max_vel = 6.0
        # max_rate = 0.015 * self.pi
        # max_eyI = 3.

        # max_states = np.array([0.1 * max_pos, 0.1 * max_pos, max_pos,
        #                     max_ang, max_ang, max_ang,
        #                     0.5 * max_vel, 0.5 * max_vel, max_vel,
        #                     max_rate, max_rate, max_rate,
        #                     0.1 * max_eyI, 0.1 * max_eyI, 1 * max_eyI, 0.1 * max_eyI])

        # max_inputs = np.array([0.2 * self.U1_max, self.U1_max, self.U1_max, self.U1_max])

        # Q = np.diag(1/max_states**2)
        # R = np.diag(1/max_inputs**2)
        # -----------------  Example code Ends ----------------- #
        # ----------------- Your Code Here ----------------- #
        # Come up with reasonable values for Q and R (state and control weights)
        # The example code above is a good starting point, feel free to use them or write you own.
        # Tune them to get the better performance

        # referencing the example code above
        max_pos = 15.0
        max_ang = 0.2 * self.pi
        max_vel = 6.0
        max_rate = 0.015 * self.pi
        max_eyI = 3.0

```

```
        max_states = np.array([0.1 * max_pos, 0.1 * max_pos, max_pos,
                               max_ang, max_ang, max_ang,
                               0.5 * max_vel, 0.5 * max_vel, max_vel,
                               max_rate, max_rate, max_rate,
                               0.1 * max_eyI, 0.1 * max_eyI, 1 * max_eyI, 0.1 * max_eyI])

        max_inputs = np.array([0.2 * self.U1_max, self.U1_max, self.U1_max, self.U1_max])

        Q = np.diag(1 / max_states ** 2)
        R = np.diag(1 / max_inputs ** 2)

        # ----------------- Your Code Ends Here ----------------- #

        # solve for LQR gains
        [K, _, _] = dlqr(A_d, B_d, Q, R)
        self.Kbl = -K

        [K_CT, _, _] = lqr(A, B, Q, R)
        Kbl_CT = -K_CT

        # initialize adaptive controller gain to baseline LQR controller gain
        self.K_ad = self.Kbl.T

        # -----------------    Example code    ----------------- #
        # self.Gamma = 3e-3 * np.eye(16)

        # Q_lyap = np.copy(Q)
        # Q_lyap[0:3,0:3] *= 30
        # Q_lyap[6:9,6:9] *= 150
        # Q_lyap[14,14] *= 2e-3
        # -----------------  Example code Ends ----------------- #
```

```
        # ----------------- Your Code Here ----------------- #
        # Come up with reasonable value for Gamma matrix and Q_lyap
        # The example code above is a good starting point, feel free to use them or write you own.
        # Tune them to get the better performance

        # referencing the example code above
        self.Gamma = 3e-3 * np.eye(16)

        Q_lyap = np.copy(Q)
        Q_lyap[0:3,0:3] *= 30
        Q_lyap[6:9,6:9] *= 150
        Q_lyap[14,14] *= 2e-3
        # ----------------- Your Code Ends Here ----------------- #

        A_m = A + self.B @ Kbl_CT
        self.P = solve_continuous_lyapunov(A_m.T, -Q_lyap)

    def update(self, r):
        """ Get current states and calculate desired control input.

        Args:
            r (np.array): reference trajectory.

        Returns:
            np.array: states. information of the 16 states.
            np.array: U. desired control input.

        """

        U = np.array([0.0, 0.0, 0.0, 0.0]).reshape(-1,1)

```

```
        # Fetch the states from the BaseController method
        x_t = super().getStates()

        # update integral term
        self.int_e1 += float((x_t[0]-r[0])*(self.delT))
        self.int_e2 += float((x_t[1]-r[1])*(self.delT))
        self.int_e3 += float((x_t[2]-r[2])*(self.delT))
        self.int_e4 += float((x_t[5]-r[3])*(self.delT))

        # Assemble error-based states into array
        error_state = np.array([self.int_e1, self.int_e2, self.int_e3, self.int_e4]).reshape((-1,1))
        states = np.concatenate((x_t, error_state))

        # initialize adaptive controller
        if self.have_initialized_adaptive == False:
            print("Initialize adaptive controller")
            self.x_m = states
            self.have_initialized_adaptive = True
        else:
            # ----------------- Your Code Here ----------------- #
            # adaptive controller update law
            # Update self.K_ad by first order approximation:
            # self.K_ad = self.K_ad + rate_of_change * self.delT

            # error term
            e = states - self.x_m

            # adaptive rate of the control gain
            rate_of_change = -self.Gamma @ states
```

```
 @ e.T @ self.P @ self.B
            self.K_ad = self.K_ad + rate_of_change * self.delT

            # ----------------- Your Code Ends Here ----------------- #

            # compute x_m at k+1
            self.x_m = self.A_d @ self.x_m + self.B_d @ self.Kbl @ self.x_m + self.Bc_d @ r
            # Compute control input
            U = self.K_ad.T @ states

        # calculate control input
        U[0] += self.g * self.m

        # Return all states and calculated control inputs U
        return states, U
```