

```

1  # MRAC Adaptive controller
2
3  # Import libraries
4  import numpy as np
5  from base_controller import BaseController
6  from lqr_solver import dlqr, lqr
7  from scipy.linalg import solve_continuous_lyapunov
   , solve_lyapunov, solve_discrete_lyapunov
8  from math import cos, sin
9  import numpy as np
10 from scipy import signal
11
12 class AdaptiveController(BaseController):
13     """ The LQR controller class.
14
15     """
16
17     def __init__(self, robot, lossOfThrust):
18         """ MRAC adaptive controller __init__
19         method.
20
21         Initialize parameters here.
22
23         Args:
24             robot (webots controller object):
25             Controller for the drone.
26             lossOfThrust (float): percent lost of
27             thrust.
28
29         """
30
31         super().__init__(robot, lossOfThrust)
32
33         # define integral error
34         self.int_e1 = 0
35         self.int_e2 = 0
36         self.int_e3 = 0
37         self.int_e4 = 0
38
39         # flag for initializing adaptive controller
40         self.have_initialized_adaptive = False

```

```

38
39     # reference model
40     self.x_m = None
41
42     # baseline LQR controller gain
43     self.Kbl = None
44
45     # Saved matrix for adaptive law computation
46     self.A_d = None
47     self.B_d = None
48     self.Bc_d = None
49
50     self.B = None
51     self.Gamma = None
52     self.P = None
53
54     # adaptive gain
55     self.K_ad = None
56
57     def initializeGainMatrix(self):
58         """ Calculate the LQR gain matrix and
59         matrices for adaptive controller.
60         """
61
62         # -----|LQR Controller
63         |-----
64         # Use the results of linearization to
65         create a state-space model
66
67         # Given parameters
68         n_p = 12 # number of states
69         m = 4 # number of integral error terms
70
71         # robot parameter
72         self.m = 0.4
73         self.d1x = 0.1122
74         self.d1y = 0.1515
75         self.d2x = 0.11709
76         self.d2y = 0.128
77         self.Ix = 0.000913855

```

```

76         self.Iy = 0.00236242
77         self.Iz = 0.00279965
78
79         # constants
80         self.g = 9.81
81         self.ct = 0.00026
82         self.ctau = 5.2e-06
83         self.U1_max = 10
84         self.pi = 3.1415926535
85
86         # ----- Your Code Here
87         ----- #
88         # Compute the continuous A, B, Bc, C, D
89         and
90         # discretized A_d, B_d, Bc_d, C_d, D_d,
91         for the computation of LQR gain
92
93         # Matrix A logic
94         # Initialize A matrix with zeros ( 16 x 16
95         )
96         A = np.zeros((n_p + m, n_p + m))
97         A[0, 6] = 1; A[1, 7] = 1; A[2, 8] = 1; A[3
98         , 9] = 1; A[4, 10] = 1; A[5, 11] = 1
99         A[6, 4] = self.g; A[7, 3] = -self.g
100        A[12, 0] = 1; A[13, 1] = 1; A[14, 2] = 1;
101        A[15, 5] = 1
102        # A[12, 0] = 1; A[12, 12] = -1; A[13, 1
103        ] = 1; A[13, 13] = -1; A[14, 2] = 1; A[14, 14] = -
104        1; A[15, 5] = 1; A[15, 15] = -1
105
106        # Matrix B logic
107        # Initialize B matrix with zeros ( 16 x 4
108        )
109        B = np.zeros((n_p + m, m))
110        B[8, 0] = 1 / self.m; B[9, 1] = 1 / self.
111        Ix; B[10, 2] = 1 / self.Iy; B[11, 3] = 1 / self.Iz
112
113        # Matrix Bc logic
114        # Initialize Bc matrix with zeros ( 16 x
115        4 )
116        Bc = np.zeros((n_p + m, m))

```

```

106         Bc[12, 0] = -1; Bc[13, 1] = -1; Bc[14, 2
    ] = -1; Bc[15, 3] = -1
107
108         # Combine B and Bc into one matrix
109         combined_B = np.hstack((B, Bc))
110
111         # Matrix C logic
112         # Initialize C matrix with zeros ( 4 x 16
    )
113         C = np.zeros((m, n_p + m))
114         C[0, 0] = 1; C[1, 1] = 1; C[2, 2] = 1; C[3
    , 3] = 1
115
116         # Matrix D logic
117         # Zero matrix ( 4 x 4 )
118         D = np.zeros((m, m))
119
120         # Discretize the system
121         sys_discrete = signal.cont2discrete((A,
    combined_B, C, D), self.delT, method='zoh')
122
123         # Extract A_d, B_d, Bc_d, C_d, D_d
124         A_d = sys_discrete[0]
125         B_d = sys_discrete[1][:, :m] # only take
    the first 4 columns
126         Bc_d = sys_discrete[1][:, m:] # only take
    the last 4 columns
127         C_d = sys_discrete[2]
128         D_d = sys_discrete[3]
129
130
131         # ----- Your Code Ends Here
    ----- #
132
133         # Record the matrix for later use
134         self.B = B # continuous version of B
135         self.A_d = A_d # discrete version of A
136         self.B_d = B_d # discrete version of B
137         self.Bc_d = Bc_d # discrete version of Bc
138
139         # ----- Example code

```

```

139         ----- #
140         # max_pos = 15.0
141         # max_ang = 0.2 * self.pi
142         # max_vel = 6.0
143         # max_rate = 0.015 * self.pi
144         # max_eyI = 3.
145
146         # max_states = np.array([0.1 * max_pos, 0.
147 1 * max_pos, max_pos,
148         #                               max_ang, max_ang,
149 max_ang,
150         #                               0.5 * max_vel, 0.5
151         * max_vel, max_vel,
152         #                               max_rate, max_rate,
153 max_rate,
154         #                               0.1 * max_eyI, 0.1
155         * max_eyI, 1 * max_eyI, 0.1 * max_eyI])
156
157         # max_inputs = np.array([0.2 * self.U1_max
158 , self.U1_max, self.U1_max, self.U1_max])
159
160         # Q = np.diag(1/max_states**2)
161         # R = np.diag(1/max_inputs**2)
162         # ----- Example code Ends
163         ----- #
164         # ----- Your Code Here
165         ----- #
166         # Come up with reasonable values for Q and
167 R (state and control weights)
168         # The example code above is a good
169 starting point, feel free to use them or write you
170 own.
171
172         # Tune them to get the better performance
173
174         # referencing the example code above
175 max_pos = 15.0
176 max_ang = 0.2 * self.pi
177 max_vel = 6.0
178 max_rate = 0.015 * self.pi
179 max_eyI = 3.0
180

```

```

169         max_states = np.array([0.1 * max_pos, 0.1
    * max_pos, max_pos,
170                                     max_ang, max_ang,
    max_ang,
171                                     0.5 * max_vel, 0.5
    * max_vel, max_vel,
172                                     max_rate, max_rate
    , max_rate,
173                                     0.1 * max_eyI, 0.1
    * max_eyI, 1 * max_eyI, 0.1 * max_eyI])
174
175         max_inputs = np.array([0.2 * self.U1_max,
    self.U1_max, self.U1_max, self.U1_max])
176
177         Q = np.diag(1 / max_states ** 2)
178         R = np.diag(1 / max_inputs ** 2)
179
180         # ----- Your Code Ends Here
    ----- #
181
182         # solve for LQR gains
183         [K, _, _] = dlqr(A_d, B_d, Q, R)
184         self.Kbl = -K
185
186         [K_CT, _, _] = lqr(A, B, Q, R)
187         Kbl_CT = -K_CT
188
189         # initialize adaptive controller gain to
    baseline LQR controller gain
190         self.K_ad = self.Kbl.T
191
192         # ----- Example code
    ----- #
193         # self.Gamma = 3e-3 * np.eye(16)
194
195         # Q_lyap = np.copy(Q)
196         # Q_lyap[0:3,0:3] *= 30
197         # Q_lyap[6:9,6:9] *= 150
198         # Q_lyap[14,14] *= 2e-3
199         # ----- Example code Ends
    ----- #

```

```

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

204
205      # referencing the example code above
206      self.Gamma = 3e-3 * np.eye(16)
207
208      Q_lyap = np.copy(Q)
209      Q_lyap[0:3,0:3] *= 30
210      Q_lyap[6:9,6:9] *= 150
211      Q_lyap[14,14] *= 2e-3
212      # ----- Your Code Ends Here
      ----- #
213
214      A_m = A + self.B @ Kbl_CT
215      self.P = solve_continuous_lyapunov(A_m.T
, -Q_lyap)
216
217      def update(self, r):
218          """ Get current states and calculate
desired control input.
219
220          Args:
221              r (np.array): reference trajectory.
222
223          Returns:
224              np.array: states. information of the
16 states.
225              np.array: U. desired control input.
226
227          """
228
229      U = np.array([0.0, 0.0, 0.0, 0.0]).reshape
(-1,1)
230

```

```

231         # Fetch the states from the BaseController
        method
232         x_t = super().getStates()
233
234         # update integral term
235         self.int_e1 += float((x_t[0]-r[0])*(self.
delT))
236         self.int_e2 += float((x_t[1]-r[1])*(self.
delT))
237         self.int_e3 += float((x_t[2]-r[2])*(self.
delT))
238         self.int_e4 += float((x_t[5]-r[3])*(self.
delT))
239
240         # Assemble error-based states into array
241         error_state = np.array([self.int_e1, self.
int_e2, self.int_e3, self.int_e4]).reshape((-1,1))
242         states = np.concatenate((x_t, error_state
))
243
244         # initialize adaptive controller
245         if self.have_initialized_adaptive == False
:
246             print("Initialize adaptive controller"
)
247             self.x_m = states
248             self.have_initialized_adaptive = True
249         else:
250             # ----- Your Code Here
            ----- #
251             # adaptive controller update law
252             # Update self.K_ad by first order
approximation:
253             # self.K_ad = self.K_ad +
rate_of_change * self.delT
254
255             # error term
256             e = states - self.x_m
257
258             # adaptive rate of the control gain
259             rate_of_change = -self.Gamma @ states

```



```

259 @ e.T @ self.P @ self.B
260         self.K_ad = self.K_ad + rate_of_change
        * self.deltT
261
262         # ----- Your Code Ends
        Here ----- #
263
264         # compute x_m at k+1
265         self.x_m = self.A_d @ self.x_m + self.
        B_d @ self.Kbl @ self.x_m + self.Bc_d @ r
266         # Compute control input
267         U = self.K_ad.T @ states
268
269         # calculate control input
270         U[0] += self.g * self.m
271
272         # Return all states and calculated control
        inputs U
273         return states, U

```