

# controlsTest

September 6, 2023

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy.linalg import expm

import control as ct
import control.optimal as obc

import time
```

```
[2]: import platform
print('Platform Info: ' + str(platform.system_alias(platform.system(),platform.
↪release()),platform.version()))
print('Platform Info: ' + platform.platform(aliased=0, terse=0))
print('Python Version: ' + platform.python_version())
print('Numpy Version: ' + np.__version__)
print('Scipy Version: ' + scipy.__version__)
print('Control Version: ' + ct.__version__)
```

Platform Info: ('Linux', '4.18.0-477.15.1.el8\_8.x86\_64', '#1 SMP Wed Jun 28 15:04:18 UTC 2023')

Platform Info: Linux-4.18.0-477.15.1.el8\_8.x86\_64-x86\_64-with-glibc2.36

Python Version: 3.10.12

Numpy Version: 1.25.2

Scipy Version: 1.11.2

Control Version: 0.9.4

```
[3]: M = 0.5
m = 0.2
l = 0.3
I = 0.006
b = 0.2
g = 9.81

Ac = np.array([[0, 1, 0, 0],
               [0, b*(m*l*l - I)/(I*(M+m)-M*m*l*l), -m*m*l*l*g/(I*(M+m)-M*m*l*l), 0],
               [0, 0, 0, 1],
```

```

[0, -b*m*1/(I*(M+m)-M*m*1*1), m*g*1*(M+m)/(I*(M+m)-M*m*1*1), 0]])

Bc = np.array([[0],
               [(I-m*1*1)/(I*(M+m)-M*m*1*1)],
               [0],
               [m*1/(I*(M+m)-M*m*1*1)]])

```

```

[4]: def convertToDiscreteTime(Ac,Bc,Ts):
      AB = expm(np.concatenate((np.concatenate((Ac, Bc),axis=1),np.
      ↪zeros((1,5))),axis=0) * Ts)
      A = np.array(AB[0:4,0:4])
      B = np.transpose(np.array([AB[0:4,-1]]))
      return A, B

```

```

[5]: # sampling rate
Ts = 0.1

# convert to discrete time
A, B = convertToDiscreteTime(Ac,Bc,Ts)

# grab number of states
numState = len(A)

```

```

[6]: # initial conditions
x_0 = np.array([[2],[0],[0],[0]])

# target state reference
xstar = np.array([[10],[0],[0],[0]])

```

```

[7]: # include state constraints, set min max velocity and acceleration, assume
maxCranePosition = 10 # m
minCranePosition = -5 # m
maxVelocity = 3 # m/s
minVelocity = -maxVelocity # m/s
maxCraneAngle = 0.1 # rad
minCraneAngle = -maxCraneAngle # rad
maxCraneAngledot = 10
minCraneAngledot = -maxCraneAngledot

maxForce = 100 # N
minForce = -maxForce # N

```

```

[8]: # Choose the finite horizon
finiteHorizon = 1 # seconds
# convert to discrete-time steps
finiteHorizon = finiteHorizon/Ts

```

```
[9]: # Specify Q and R matrices
Q = np.eye(numState)
R = np.eye(1)*1
```

```
[10]: # create system model (used to mimic the system)
system = ct.ss2io(ct.ss(A, B, np.eye(numState), 0, Ts))

# Assume full state knowledge
C = np.eye(numState)
# create mathematical model that will be the basis of the controller design
model = ct.ss2io(ct.ss(A, B, C, 0, Ts))

# create constraint vector
constraints = [obc.state_range_constraint(system, [minCranePosition,
    ↪minVelocity, minCraneAngle, minCraneAngledot], [maxCranePosition,
    ↪maxVelocity, maxCraneAngle, maxCraneAngledot]), obc.
    ↪input_range_constraint(system, [minForce], [maxForce])]

# create stage cost function on the model, with given desired error signal and
    ↪zero control input
cost = obc.quadratic_cost(model, Q, R, x0=xstar.flatten(), u0=0)
# note: this uses the 'model' (which could be different to the system)

# online MPC controller object is constructed with finite horizon, cost and
    ↪constraints
ctrl = obc.create_mpc_iosystem(model, np.arange(0, finiteHorizon) * Ts, cost,
    ↪constraints)

# create closed-loop feedback 'loop' object using the feedback function to link
    ↪the controller to the system
loop = ct.feedback(system, ctrl, 1)
```

```
[11]: res = obc.solve_ocp(model, np.arange(0, finiteHorizon) * Ts, x_0.flatten(),
    ↪cost, constraints, squeeze=True)

print(res)
```

Summary statistics:

```
* Cost function calls: 105
* Constraint calls: 126
* System simulations: 215
* Final cost: 568.294958426968
message: Optimization terminated successfully
success: True
status: 0
    fun: 568.294958426968
      x: [ 8.085e-01  1.464e-02  4.912e-01  7.621e-01  5.672e-01
```

```

        5.080e-01  4.151e-01  2.398e-01  0.000e+00  0.000e+00]
nit: 9
jac: [-3.049e+00 -4.285e+00 -1.991e+00 -2.014e-03 -8.926e-04
      1.045e-03 -8.392e-04  1.541e-03  0.000e+00  0.000e+00]
nfev: 105
njev: 9
problem: <control.optimal.OptimalControlProblem object at 0x7f0e2d906650>
cost: 568.294958426968
time: [ 0.000e+00  1.000e-01  2.000e-01  3.000e-01  4.000e-01
       5.000e-01  6.000e-01  7.000e-01  8.000e-01  9.000e-01]
inputs: [ 8.085e-01  1.464e-02  4.912e-01  7.621e-01  5.672e-01
         5.080e-01  4.151e-01  2.398e-01  0.000e+00  0.000e+00]
states: [[ 2.000e+00  2.010e+00 ...  2.216e+00  2.261e+00]
        [ 0.000e+00  1.856e-01 ...  4.615e-01  4.397e-01]
        [ 0.000e+00 -4.624e-02 ... -1.408e-02  1.761e-02]
        [ 0.000e+00 -8.497e-01 ...  2.412e-01  3.458e-01]]

```

```

[12]: # set length of simulation
Nsim = 20
# convert to number of discrete-time steps
Nsim = Nsim/Ts

# Simulate the closed-loop MPC
start = time.time()
tout, xout = ct.input_output_response(loop, np.arange(0, Nsim) * Ts, 0, x_0.
    ↪flatten())
end = time.time()
print("Computation time = %g seconds" % (end-start))

```

```

/opt/conda/lib/python3.10/site-packages/control/optimal.py:928: UserWarning:
unable to solve optimal control problem
scipy.optimize.minimize returned Positive directional derivative for linesearch
warnings.warn(

Computation time = 159.36 seconds

```

```

[13]: figlabels = ['Crane Cart Position [m]', 'Cart Velocity [m/s]', 'Object Degree_
    ↪from Centre [rad]', 'Object Degree rate [rad/s]']
for i, y in enumerate(xout):
    plt.figure()
    plt.plot(tout, y)
    plt.plot(tout, xstar[i] * np.ones(tout.shape), 'k--')
    plt.ylabel(figlabels[i])
    plt.xlabel('Time [sec]')

```







