ELG7113 Machine Learning for Adaptive and Intelligent Control Systems

Student: Derek Boase

Std Num: 300043860

e-mail: dboas065@uottawa.ca

assignment GitHub: git@github.com:derekboase/Adaptive_Control_Code.git

In [1]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import sympy as sp

from sympy.abc import r, alpha
from numpy import cos, sin, pi
```

Question 1: Optimal Control in Discrete Time

In [2]:
```
_uk, _lamk, _lamk1, _xk, _xk1, = sp.symbols('u_k,lambda_k,lambda_k+1,x_k,x_k+1')
Ham = sp.symbols('H^k')

xk1 = _xk * _uk + alpha
L = r/2*_uk ** 2

H_eq = sp.Eq(Ham, L + _lamk1*xk1)
state = sp.Eq(_xk1, sp.diff(H_eq.rhs, _lamk1))
costate = sp.Eq(_lamk, sp.diff(H_eq.rhs, _xk))
stationarity = sp.Eq(0, sp.diff(H_eq.rhs, _uk))
```

Q1.1 Hamiltonian Derivation

The Hamiltonian is found by implementing the equation,

$H^k = L^k(x_k, u_k) + \lambda_{k+1} f^k(x_k, u_k)$

where,

$L^k(x_k, u_k) = \frac{r}{2} u_k^2$

$f^k(x_k, u_k) = x_k u_k + \alpha$

Then,

$$H^k = \lambda_{k+1}(\alpha + u_k x_k) + \frac{r u_k^2}{2}$$

In [3]:
```
state
```

Out[3]: $x_{k+1} = \alpha + u_k x_k$

In [4]:
```
costate
```

Out[4]: $\lambda_k = \lambda_{k+1} u_k$

In [5]:
```
stationarity
```

Out[5]: $0 = \lambda_{k+1} x_k + r u_k$

Q1.2 Elimination of u_k

In [6]:
```
uk = sp.solve(stationarity, _uk)[0]
```

In [7]:
```
state_subs = state.subs(_uk, uk)
state_subs
```

Out[7]: $$x_{k+1} = \alpha - \frac{\lambda_{k+1} x_k^2}{r}$$

In [8]:
```
costate_subs = costate.subs(_uk, uk)
costate_subs
```

Out[8]: $$\lambda_k = -\frac{\lambda_{k+1}^2 x_k}{r}$$

Q1.3 Characteristic Equation

In [9]:
```
_x0, _x1, _x2, _lam0, _lam1, _lam2 = sp.symbols('x_0,x_1,x_2,lambda_0,lambda_1,lambda_2')

x1 = state_subs.subs([(_xk1, _x1),
                      (_lamk1, _lam1),
                      (_xk, _x0)])

lam1 = costate_subs.subs([(_lamk, _lam1),
                          (_lamk1, _lam2),
                          (_xk, _x1)])

x1_lam2 = sp.Eq(_x1, sp.solve(x1.subs(_lam1, lam1.rhs), _x1)[0])

x2 = state_subs.subs([(_xk1, _x2),
                      (_lamk1, _lam2),
                      (_xk, _x1)])
x2_x1 = sp.simplify(sp.expand(x2.subs([(_x1, x1_lam2.rhs), (_x2, 0)])))
num, denum = sp.fraction(x2_x1.lhs)
characteristic = sp.Eq(sp.simplify(sp.expand(num))/alpha, 0)
```

Then $x_1(\lambda_2)$ and $f(\lambda_2)$ are given as,

$$x_1 = \frac{\alpha r^2}{-\lambda_2^2 x_0^2 + r^2}$$

$$-\alpha\lambda_2 r^3 + \lambda_2^4 x_0^4 - 2\lambda_2^2 r^2 x_0^2 + r^4 = 0$$

where $f(\lambda_2)$ is the characteristic equation

Q1.4 Optimal Stragety

In [10]:
```
_u0star, _u1star, _x0star, _x1star, _x2star = sp.symbols('u_0{^*},u_1{^*},x_0{^*},x_1{^*},x_2{^*}')
sub_vals = [alpha, r, _lam2, _x0]

u0star = sp.Eq(_u0star,
               sp.simplify(sp.expand(uk.subs([(_lamk1, lam1.rhs),
                                              (_xk, _x0),
                                              (_x1, x1_lam2.rhs)]))))
u0star_func = sp.lambdify(sub_vals, u0star.rhs)

u1star = sp.Eq(_u1star,

               sp.simplify(sp.expand(uk.subs([(_lamk1, _lam2),
                                              (_xk, x1_lam2.rhs)]))))
u1star_func = sp.lambdify(sub_vals, u1star.rhs)

x0star = sp.Eq(_x0star, _x0)
x0star_func = sp.lambdify(sub_vals, x0star.rhs)

x1star = sp.Eq(_x1star, x1_lam2.rhs)
x1star_func = sp.lambdify(sub_vals, x1star.rhs)

x2star = sp.Eq(_x2star, 0)
x2star_func = sp.lambdify(sub_vals, x2star.rhs)

display(u0star)
display(u1star)
display(x0star)
display(x1star)
display(x2star)
```

$$u_0{}^* = \frac{\alpha\lambda_2^2 x_0}{-\lambda_2^2 x_0^2 + r^2}$$

$$u_1{}^* = \frac{\alpha\lambda_2 r}{\lambda_2^2 x_0^2 - r^2}$$

$$x_0{}^* = x_0$$

$$x_1{}^* = \frac{\alpha r^2}{-\lambda_2^2 x_0^2 + r^2}$$

$$x_2{}^* = 0$$

Q1.5 Implementation

In [11]:
```
vals = [(alpha, 2), (r, 1), (_x0, 1.5)]
char_subs = characteristic.subs(vals)
char_subs_poly = sp.Poly(char_subs, _lam2)
char_coeffs = char_subs_poly.coeffs()
char_coeffs.insert(1, 0.0)
sols = np.roots(char_coeffs)

optimal = []
for idx in [2, 3]:
    optimal.append([u0star_func(2, 1, np.real(sols[idx]), 1.5),
                    u1star_func(2, 1, np.real(sols[idx]), 1.5),
                    x0star_func(2, 1, np.real(sols[idx]), 1.5),
                    x1star_func(2, 1, np.real(sols[idx]), 1.5),
                    x2star_func(2, 1, np.real(sols[idx]), 1.5)])
optimal = np.array(optimal).T
```

In [12]:
```
col_names = [f'$\lambda_2$ = {np.round(np.real(sols[2]), 4)}',
             f'$\lambda_2$ = {np.round(np.real(sols[3]), 4)}']
idx_names = ['$u_0^*$', '$u_1^*$', '$x_0$', '$x_1^*$', '$x_2$']

answer = pd.DataFrame(optimal, index=idx_names, columns=col_names)
answer
```

Out[12]:

	λ_2 = 1.0422	λ_2 = 0.3086
u_0^*	-2.256875	0.363732
u_1^*	1.443717	-0.785670
x_0	1.500000	1.500000
x_1^*	-1.385313	2.545598
x_2	0.000000	0.000000

Clearly in both cases for the real solutions to the characteristic equation $x_0 = 1.5$ and $x_2 = 0$, as required

Question 2: Dynamic Programming

In [13]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import sympy as sp

from sympy.abc import r, alpha
from numpy import cos, sin, pi
```

Q2.1 Derivation of the equations

In [14]:
```
syms = 'J_2^{*},J_1^{*},J_0^{*},u_k,u_1,u_0,x_{k},x_{k+1},x_2,x_1,x_0'
_J2, _J1, _J0, _uk, _u1, _u0, _xk, _xk1, _x2, _x1, _x0 = sp.symbols(syms)
_u0star, _u1star, _x0star, _x1star, _x2star = sp.symbols('u_0{^*},u_1{^*},x_0{^*},x_1{^*},x_2{^*}')

xk1_eq = sp.Eq(_xk1, _xk*_uk + alpha)

# For k = N = 2
J2_eq = sp.Eq(_J2, 1/2*_x2**2)


# For k = N-1 = 1
J1_eq = sp.Eq(_J1, J2_eq.rhs + r/2*(_u1**2))
x2 = (xk1_eq.rhs).subs([(_uk, _u1),
                        (_xk, _x1)])
J1_subs = sp.Eq(_J1, sp.collect(sp.expand(J1_eq.subs(_x2, x2)).rhs, _u1))
J1_diff_eq = sp.diff(J1_subs, _u1)
J1_diff = sp.diff(J1_subs.rhs, _u1)
u1_star_func = sp.solve(J1_diff, _u1)[0]
u1_star_eq = sp.Eq(_u1star, u1_star_func)


# For k = N-2 = 0
J0_eq = sp.Eq(_J0, sp.collect(J1_eq.rhs + r/2*(_u0**2), r))
x1 = (xk1_eq.rhs).subs([(_uk, _u0),
                        (_xk, _x0)])
# J0_subs = sp.Eq()
J0_test = sp.Eq(_J0, sp.simplify(sp.expand(J1_subs.rhs+ r/2*(_u0)**2)))
J0_test = J0_test.subs([(_u1, u1_star_func),
                        (_x1, x1)])
J0_test = sp.simplify(J0_test)

J0_diff_test = sp.simplify(sp.diff(J0_test.rhs, _u0))
num, denum = sp.fraction(J0_diff_test)
tit = sp.collect(sp.expand(num), _u0).subs([(alpha, 2),
                                            (r, 1),
                                            (_x0, 1.5)])
tit_poly = sp.Poly(tit, _u0)
coeffs_lst = np.roots(tit_poly.coeffs())
vals = []
for n in coeffs_lst:
    if np.imag(n) == 0 and np.real(n) > 0:
        vals.append(np.real(n))

r1 = [1, 20]
dec = 4
sols = []
for r_val in r1:
    x0_star = sp.Eq(_x0star, 1.5)
    u0_star = sp.Eq(_u0star, vals[0])
    x1_star = sp.Eq(_x1star, x1.subs([(alpha, 2), (r, r_val), (_x0, x0_star.rhs), (_u0, u0_star.rhs)]))
    u1_star = sp.Eq(_u1star, u1_star_func.subs([(alpha, 2), (r, r_val), (_x1, x1_star.rhs)]))
    x2_star = sp.Eq(_x2star, x2.subs([(alpha, 2), (r, r_val), (_u1, u1_star.rhs), (_x1, x1_star.rhs)]))
    sols.append([u0_star.rhs, u1_star.rhs, x0_star.rhs, x1_star.rhs, x2_star.rhs])
ans_lst = np.array(sols).T
```

Given the system dynamics, $x_{k+1} = \alpha + u_k x_k$ we find the optimal control using dynamic programming working backwards. Then,

$J_2^* = 0.5x_2^2$

Knowing this and that $J_1^* = J_2^* + \frac{r}{2}u_1^2$ we find,

$$J_1^* = \frac{r u_1^2}{2} + 0.5x_2^2$$

To find we take the derivative of $J_1^* = \dfrac{r u_1^2}{2} + 0.5x_2^2$ and solve for

$$u_1{}^* = -\frac{\alpha x_1}{r + x_1^2}$$

Following the same approach (as outlined in the code) we solve for the derivative of J_0^*. This yields a 5th order equation that is solved numerically for the implementation. u_0^* is then found by solving the polynomial,

$$5.0625u_0^5 + 27.0u_0^4 + 58.5u_0^3 + 60.0u_0^2 + 16.0u_0 - 12.0 = 0$$

Q2.2 Implementation

In [15]:
```
index_names = ['$u_0^*$', '$u_1^*$', '$x_0^*$', '$x_1^*$', '$x_2^*$']
column_names = ['$r = 1$', 'r=20']
ans = pd.DataFrame(ans_lst, index=index_names, columns=column_names)
ans
```

Out[15]:

	$r = 1$	r=20
u_0^*	0.299851871344322	0.299851871344322
u_1^*	-0.699795427160291	-0.188434218312075
x_0^*	1.50000000000000	1.50000000000000
x_1^*	2.44977780701648	2.44977780701648
x_2^*	0.285656693091098	1.53837803389658

Here the difference in the responses is that the policy derived with $r = 20$ seems to tend to 0 much quicker than the other. This is due to the fact that the weighting term that corresponds to the policy is much larger. The performance index is then more concerned with reducing the policy than with reducing the final state.

In [ ]:

In [ ]: