OpenControl

VNOpenAI

Jul 16, 2021

CONTENTS

The OpenControl is a python package that implement basic algorithms for analysis and design of optimal feedback controllers.

Features:

- Classical control methods
- Linear quadratic regulator (`LQR`) computation
- Robust Adaptive Dynamic Programming (`ADP`) for optimal linear/nonlinear control systems
- Off-policy, On-policy learning algorithms for linear/nonlinear systems
- Experience replay algorithms for linear/nonlinear systems (TODO)

Documentation

CHAPTER

ONE

INTRODUCTION

OpenControl is an open-source library that support rapid development of algorithms in modern control theory. This source code is written by python for robotic and control system simulation purposes. We welcome contributions from the open-source community.

1.1 Over view of the toolbox

The package contains a list of algorithms that cover the topic of classical feedback and adaptive optimal control for continuous system. The initial goal is to provide an analysis of the feedback control systems throung simulations. In particular, we focus on the **Adaptive/Approximate Dynamic Programming**, a powerful methodology that integrates the idea of **Reinforcement Learning** (RL) and with **Control Theory** in a model-free, data-driven approaches.

1.1.1 With OpenControl you can design the control system through simulating:

- Classic feedback control design
- Policy Iteration (PI) algorithms for both linear (LTI) and non-linear systems.
- Compare with LQR (in case of LTI) results.
- Analysis On-policy Learning versus Off-policy Learning.
- Customize the value function and time-interval for learning
- Customize the basis function of neural network used for approximating actor/critic
- Reproduce trending papers' result, explore new control algorithms
- Visualize the simulation results by Tensorboard

1.2 Installation

The *OpenControl* package can be installed using pip, support Linux, MacOS and Window 10 (64-bit) with python 3.8

```
pip install OpenControl
```

1.3 Getting started

```
>>> import OpenControl
```

CHAPTER

TWO

CLASSIC CONTROL

2.1 About

module classiccontrol is the fundamental package needed for implementation control algorithms in with Python.

the main object of module is LTI system and relative algorithms, controller,... The Linear time-invariant system formulate in:

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

This package contains:

- **linearsystem.py: implement class LTI and basic attributes of system,**
  - system matrix
  - dimension
  - poles of system
  - **controllability and observability:**
    - Kalman standard
    - Hatus standard
  - simulation with controller and observer
- **bibo.py: implement algorithms for determining stability of LTI system**
  - Gerschgorin
  - Lyapunov
  - Hurwitz
- **controller.py: Design controller for LTI**
  - **Pole statement:**
    - Roppernecker method
    - Arckerman method
    - Modal method
  - **Optimal control:**
    - LQR
- **observer.py: Design observer for LTI**

  - Luenberger observer

- **utils.py**: collection of small and common Python functions which re-use a lots in difference package

This manual contains many examples of use, usually prefixed with the Python prompt >>> (which is not a part of the example code). The examples assume that you have first entered:

```
from OpenControl import classiccontrol
```

Declare LTI system:

```
import numpy as np

A = np.array([[0,          1,    0,     0],
              [28.0286,    0,    0,     0],
              [0,          0,    0,     1],
              [-1.4014,    0,    0,     0]])

B = np.array([[    0     ],
              [-1.5873   ],
              [    0     ],
              [ 0.6349   ]])

C = np.array([[1,    0,    0,     0],
              [0,    1,    0,     0],
              [0,    0,    1,     0],
              [0,    0,    0,     1]])

sys = classiccontrol.linearsystem.LTI(A=A, B=B, C=C)
```

2.2 Controllability and observability

Kalman standard: **Theorem**: The linear continuous-time system is controllable if and only if the controllability matrix has full rank.

The observability matrix, in this case, defined by:

Theorem: The linear continuous-time system is controllable if and only if the controllability matrix has full rank.

The controllability matrix, in this case, defined by:

Hatus standard: System is controllable if matrix: is full rank with any s.

$$Rank(sI - A, B) = n$$
$$Systemisobservableifmatrix : isfullrank$$

2.3 Stability

Gerschgorin: A is system matrix. Elements of A, aij.

$$Ri(A) = \sum_{j=1, j \neq i} |a_{ij}$$

Hurwitz: system is stable if all the eigen values of system matrix on the left of complex coordinate

Lyapunov:

a) A is hurwitz, If there exist a positive definite squad matrix $Q = Q^T$ such that $P = -(QA + A^T Q)$ is a positive definite matrix

b) A is hurwitz ,If there exist a positive definite squad matrix $P = P^T$ such that equation $P = -(QA + A^T Q)$ got a solution Q and Q is positive definite squad matrix

Stability of system:

```
#select the algorithms

sys.is_stable(algorimth='gerschgorin')
sys.is_stable(algorimth='hurwitz')
sys.is_stable(algorimth='lyapunov')
```

2.4 Code examples 1

This experiment from the paper 'Observer-Based Controllers for Two-Wheeled Inverted Robots with Unknown Input Disturbance and Model Uncertainty' Declare LTI system:

```
import numpy as np

A = np.array([[0,          1,    0,     0],
              [28.0286,    0,    0,     0],
              [0,          0,    0,     1],
              [-1.4014,    0,    0,     0]])

B = np.array([[    0     ],
              [-1.5873   ],
              [    0     ],
              [ 0.6349   ]])

C = np.array([[1,    0,    0,     0],
              [0,    1,    0,     0],
              [0,    0,    1,     0],
              [0,    0,    0,     1]])

sys = classiccontrol.linearsystem.LTI(A=A, B=B, C=C)
```

Basic attribute of LTI system.

```
stability = sys.is_stable()
controllability =sys.controllable()
observability = sys.observable()
poles = sys.eigvals()
```

simulate Open-loop system.

```
sys.setup_simulink(max_step=1e-3, algo='RK45', t_sim=(0,10), x0=None, sample_time = 1e-2,
↪z0=None)
time_array,state,output = sys.step_response()
```

Design controller

```
controller = classiccontrol.controller.PoleStatement(pole=[-3,-4,-5,-6], system=sys)
R = controller.compute()
```

Design observer

```
observer = classiccontrol.observer.Luenberger(pole=[-3,-4,-5,-6], system=sys)
L = observer.compute()
```

simulate Closed-loop system with state-feedback controller R

```
sys.setup_simulink()
time_array,state,output,state_obs = sys.apply_state_feedback(R)
```

simulate Closed-loop system with output-feedback L-R

```
sys.setup_simulink()
time_array,state,output,state_obs = sys.apply_output_feedback(L,R)
```

2.5 Code examples 2

simulate a Two-Wheeled Inverted Rotbots. cited in the paper 'Observer-Based Controllers for Two-Wheeled Inverted Robots with Unknown Input Disturbance and Model Uncertainty'

object got the folowing parameters:

$$A_1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 0 & -14.50 & 0 & -32.47 & 1.07 & 0 \\ 0 & 162.07 & 0 & 242.88 & -8.01 & 0 \\ 0 & 0 & 0 & 0 & 0 & -5.66 \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad B_2 = \begin{bmatrix} 107.159 & 107.159 \\ -801.536 & -801.536 \\ -191.648 & 191.648 \end{bmatrix}$$

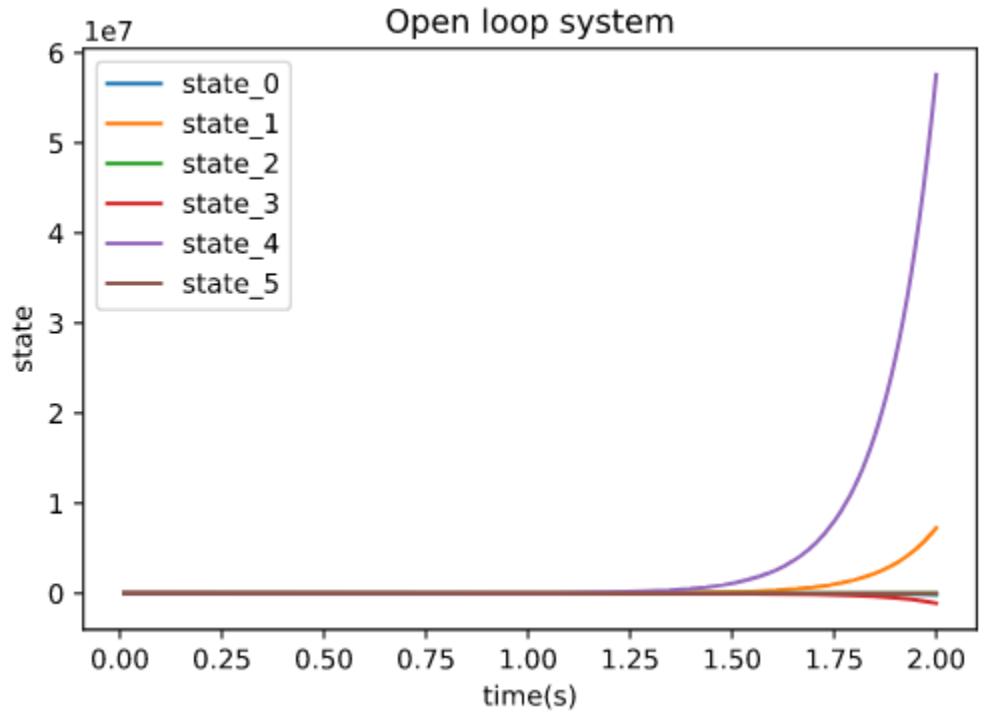
$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

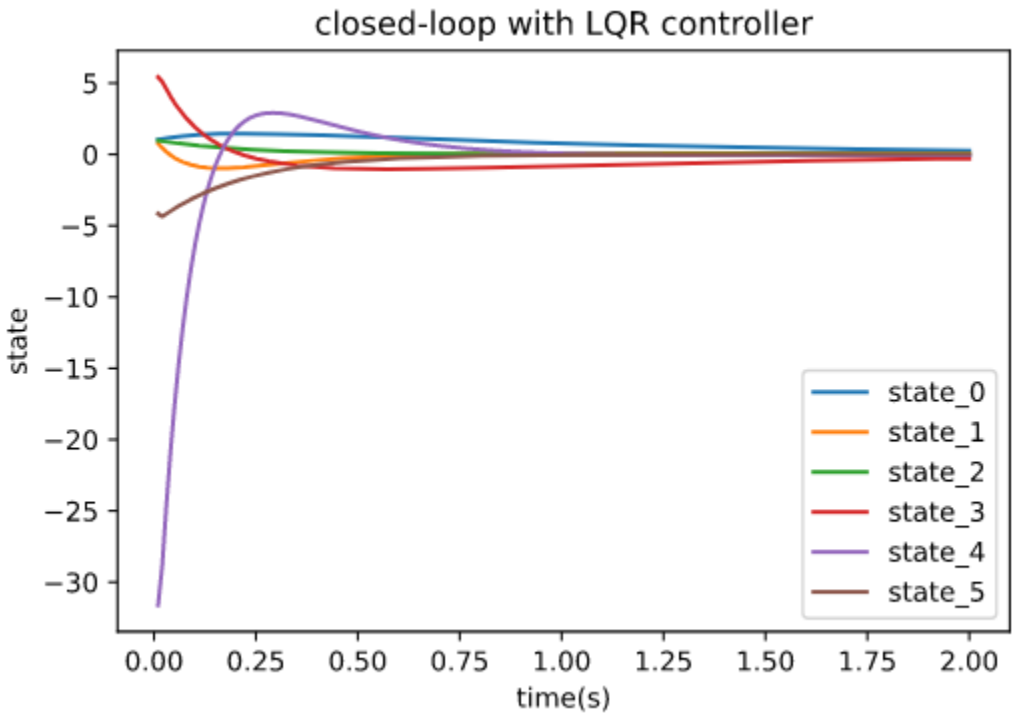
$$R_c = \text{diag}([1, 1])$$

$$Q_c = \text{diag}([4, 3, 20, 3.5, 0.001, 0.9])$$

Simulate behavior of open-loop system Design LQR controller Simulate behavior of closed-loop system with LQR controller

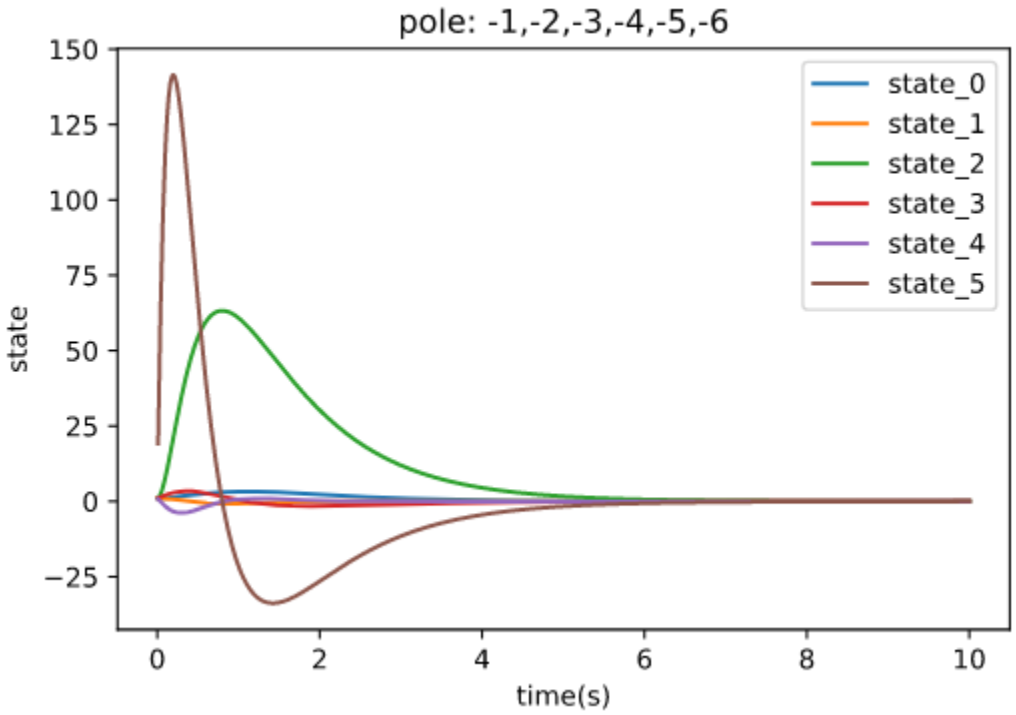
```
import numpy as np
from OpenControl import classiccontrol
A = np.array([0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,
        0,0,1,0,-14.5,0,-32.47,1.07,0,0,162.07,
        0,242.88,-8.01,0,0,0,0,0,0,-5.66]).reshape(6,6)
B = np.array([0,0,0,0,0,0,107.159,107.159,-801.536,
        -801.536,-191.648,191.648]).reshape(6,2)
C = np.eye(6)
Q = np.diag([4,3,20,3.5,0.001,0.9])
R = np.diag([1,1])
sys = classiccontrol.linearsystem.LTI(A=A,B=B,C=C)
ctl = classiccontrol.controller.LQR(sys,Q,R)
R = ctl.compute()
sys.setup_simulink(t_sim=(0,2))
time_array, state,output = sys.step_response()
time_array2, state2,output2 = sys.apply_state_feedback(R)
```

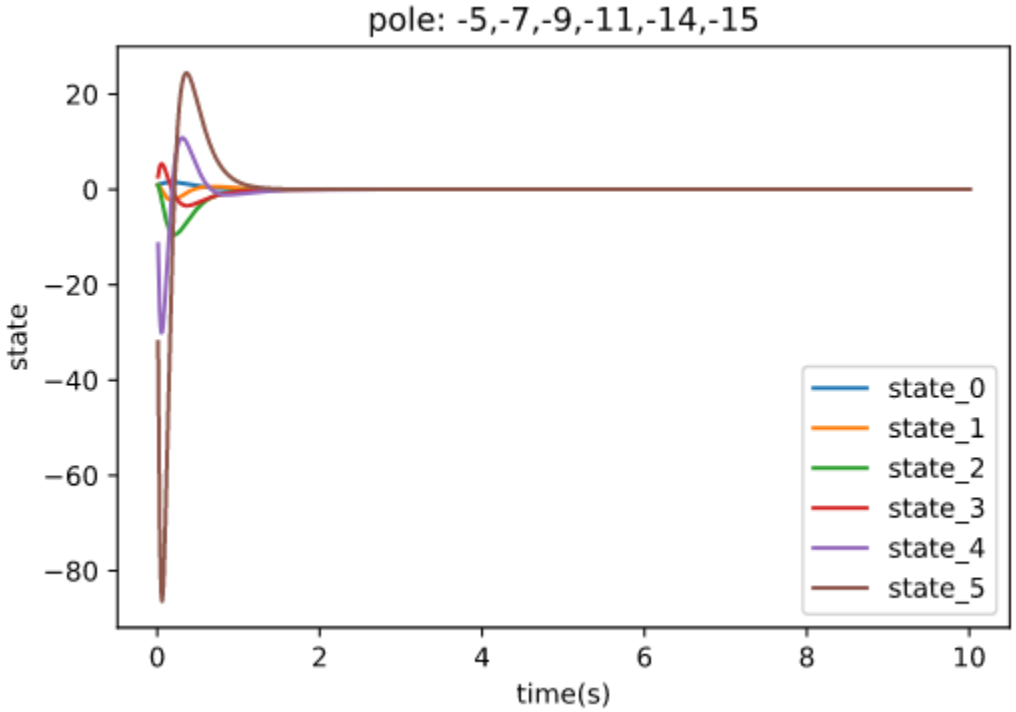


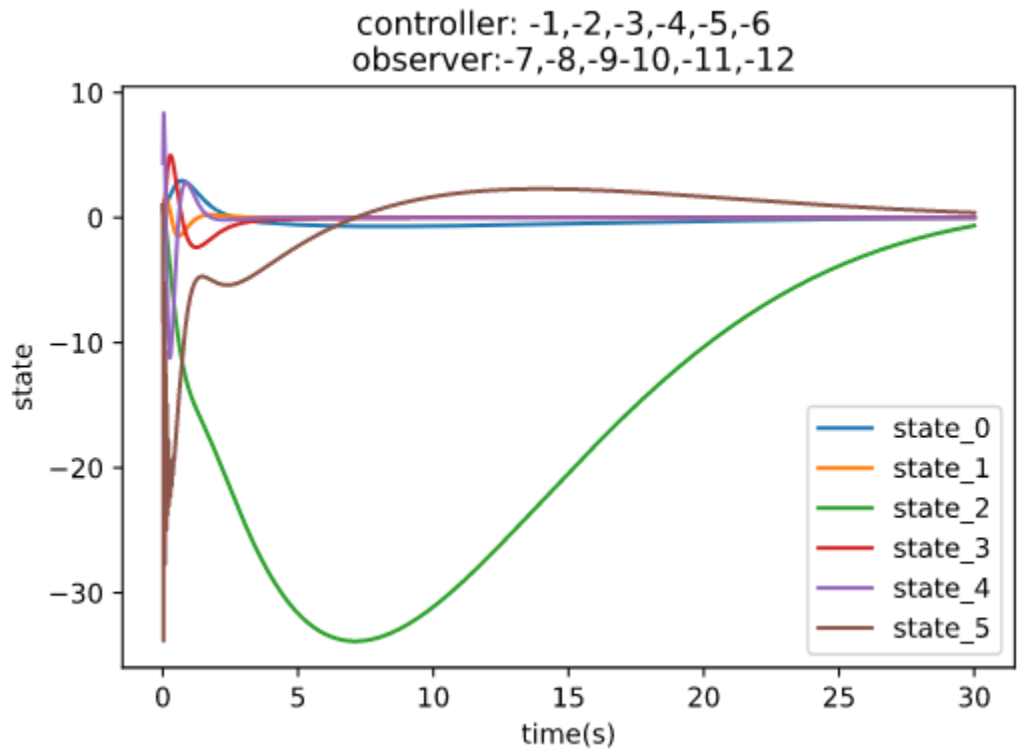


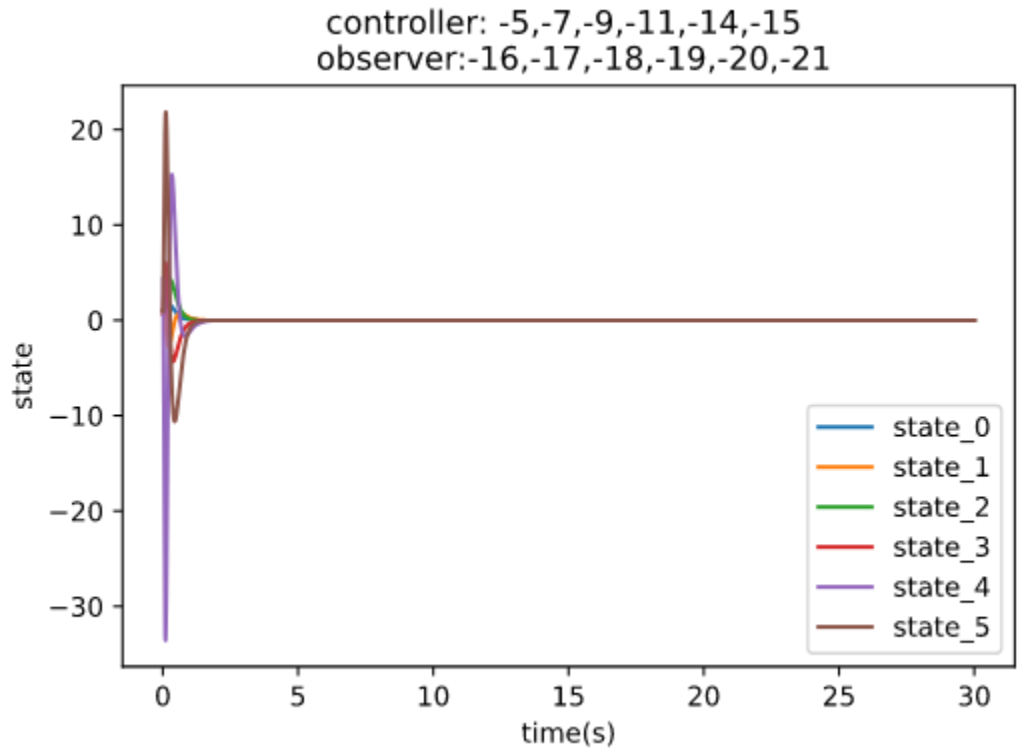
Design and simulate pole-statement controller 1 with pre-define poles : -1,-2,-3,-4,-5,-6 Design and simulate pole-statement controller 2 with pre-define poles : -5,-7,-9,-11,-14,-15 Design Luenberger observer with pre-define poles : -7,-8,-9,-10,-11,-12 Design Luenberger observer with pre-define poles : -16,-17,-18,-19,-20,-21

```
ctl_1 = classiccontrol.controller.PoleStatement(pole=[-1,-2,-3,-4,-5,-6],system=sys)
R1 =ctl_1.compute()
obs_1 = classiccontrol.observer.Luenberger(pole=[-7,-8,-9,-10,-11,-12],system=sys)
L1 = obs_1.compute()
ctl_2 = classiccontrol.controller.PoleStatement(pole=[-5,-7,-9,-11,-14,-15],system=sys)
R2 =ctl_2.compute()
obs_2 = classiccontrol.observer.Luenberger(pole=[-16,-17,-18,-19,-20,-21],system=sys)
L2 = obs_2.compute()
sys.setup_simulink(t_sim=(0,50))#,x0=np.random.randint(-1,1,(6,1)))
time_array1, state1,output1 = sys.apply_output_feedback(L1,R1)
time_array2, state2,output2 = sys.apply_output_feedback(L1,R1)
time_array3, state3,output3,z0 = sys.apply_output_feedback(L1,R1)
time_array4, state4,output4,z1 = sys.apply_output_feedback(L2,R2)
```


pole: -1,-2,-3,-4,-5,-6
state_0
state_1
state_2
state_3
state_4
state_5
state
time(s)


pole: -5,-7,-9,-11,-14,-15
state_0
state_1
state_2
state_3
state_4
state_5
state
time(s)


controller: -1,-2,-3,-4,-5,-6
observer:-7,-8,-9-10,-11,-12
state
time(s)
state_0
state_1
state_2
state_3
state_4
state_5


controller: -5,-7,-9,-11,-14,-15
observer:-16,-17,-18,-19,-20,-21
state
time(s)
state_0
state_1
state_2
state_3
state_4
state_5

CHAPTER

THREE

SYSTEM REPRESENTATION

3.1 LTI system

Consider a continuous linear time invariant system described by:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{3.1}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the control input, $A \in \mathbb{R}^{nn}$ is the state matrix, $B \in \mathbb{R}^{nm}$ is the control matrix

To create a LTI system use *OpenControl.ADP_control.LTI*

```
import numpy as np
from ADP_control import LTI

A = np.eye(3); B = np.ones((3,1))
sys = LTI(A, B) # or sys = LTI(A, B, C, D)
```

then setup a simulation section, use *OpenControl.ADP_control.LTI.setSimulationParam()*

```
t_start = 0; t_stop = 10
x0 = np.array([1,2,3])
sample_time = 0.01

sys.setupSimulationParam(t_sim=(t_start, t_stop), x0=x0, sample_time=sample_time)
```

3.2 Non-linear System

Consider a continuous-time affine nonlinear dynamical system described by:

$$\dot{x} = f(x) + g(x)u \tag{3.2}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the control input, $f : \mathbb{R}^n \to \mathbb{R}^n$ and $g : \mathbb{R}^n \to \mathbb{R}^{nm}$ are locally Lipschitz mappings with $f(0) = 0$

To create a nonlinear system use *OpenControl.ADP_control.NonLin*

```
from ADP_control import nonLin

dot_x = lambda t,x,u: 3x + np.array([0 0 1]).dot(u)
dimension = (3,1)
sys = NonLin(dot_x, dimension)
```

then setup a simulation section, see *setup a simulation section*

CHAPTER

FOUR

LINEAR ADP CONTROLLER

The below algorithms is given in the book Robust Adaptive Dynamic Programming.

4.1 Problem statements

Given the linear system equation (3.1), design a linear quadratic regulator (LQR) in the form of $u = -Kx$ that minimizes the following cost function

$$V(x) = \int_t^\infty (x^T Q x + u^T R u) d\tau = x^T P x$$

where $Q = Q^T \geq 0, R = R^T > 0$.

The solution of this problem is $K*$ which is also the solution of the Riccati equation, can be obtained by *OpenControl.ADP_control.LTIController.LQR()*. However, this approach requires the knowledge of the system dynamic (the matrix `A` and `B`). The **model-free** approach below will resolve this problem.

Lets begin with another control policy $u = -K_k x + e$ where the time-varying signal e denotes an artificial noise, known as the `exploration noise`. Taking time derivative of the cost function and integrate it within interval $[t, t + \delta t]$ to obtain:

$$x^T(t+\delta t) P_k x(t+\delta t) - x^T(t) P_k x(t) - 2 \int_t^{t+\delta t} e^T R K_{k+1} x d\tau = -\int_t^{t+\delta t} x^T Q_k x d\tau \tag{4.1}$$

where $Q_k = Q + K_k^T R K_k$.
We can see that this is the fix point equation, meaning that the K matrix can be solve by finite number of iteration.
And because of no requirement of the system dynamic, it is a **data-driven** approach.

4.2 On-policy learning

For computational simplicity, we rewrite (4.1) in the following matrix form:

$$\Theta_k \begin{bmatrix} vec(P_k) \\ vec(K_{k+1}) \end{bmatrix} = \Xi_k \tag{4.2}$$

where

$$\Theta_k = \begin{bmatrix} x^T \otimes x^T|_{t_{k,1}}^{t_{k,1+\delta t}} & -2\int_{t_{k,1}}^{t_{k,1+\delta t}} (x^T \otimes e^T R)dt \\ x^T \otimes x^T|_{t_{k,2}}^{t_{k,2+\delta t}} & -2\int_{t_{k,2}}^{t_{k,2+\delta t}} (x^T \otimes e^T R)dt \\ \vdots & \vdots \\ x^T \otimes x^T|_{t_{k,l}}^{t_{k,l+\delta t}} & -2\int_{t_{k,l}}^{t_{k,l+\delta t}} (x^T \otimes e^T R)dt \end{bmatrix},$$

$$\Xi_k = \begin{bmatrix} -\int_{t_{k,1}}^{t_{k,1+\delta t}} x^T Q_k x dt \\ -\int_{t_{k,2}}^{t_{k,2+\delta t}} x^T Q_k x dt \\ \vdots \\ -\int_{t_{k,l}}^{t_{k,l+\delta t}} x^T Q_k x dt \end{bmatrix}$$

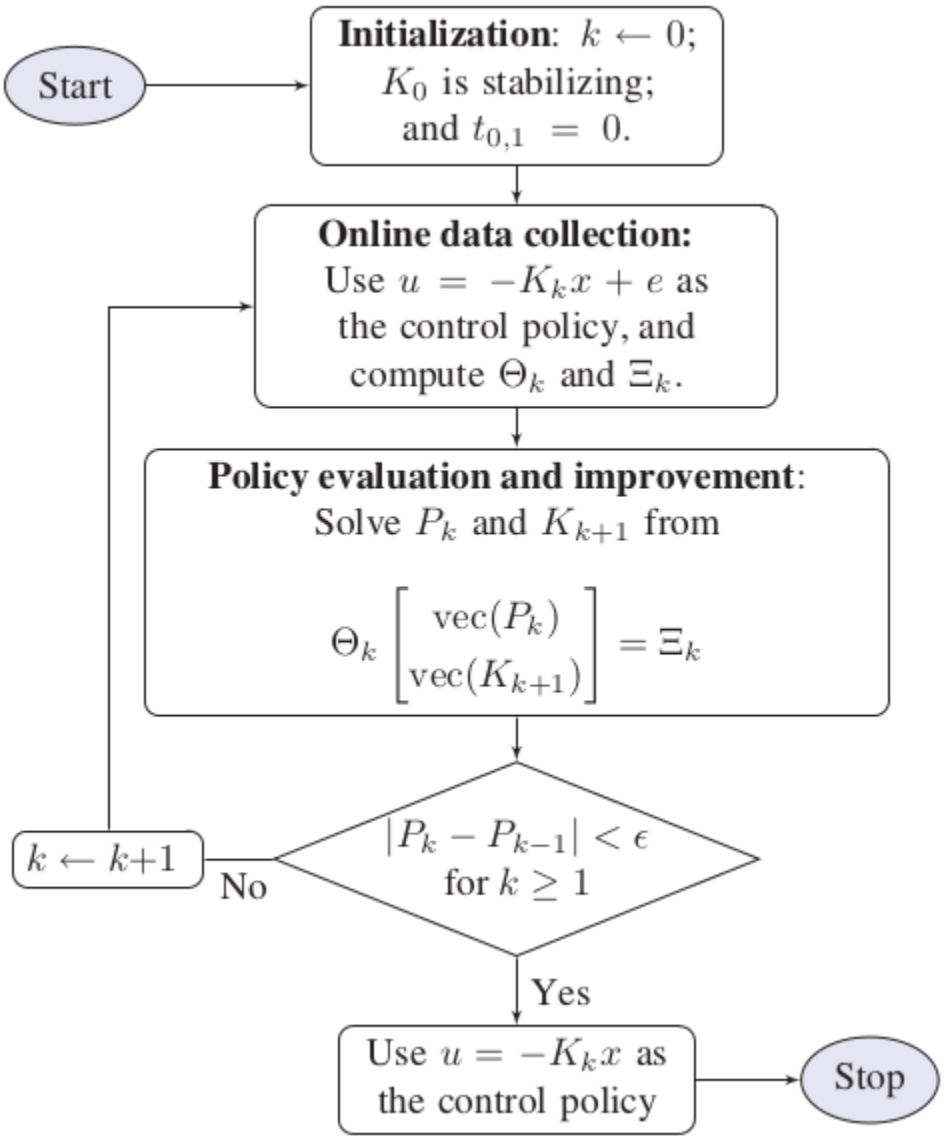
Note:

- To adopt the *persistent excitation* condition, one must choose a sufficiently large `number of data` $l > 0$ to satisfy the following condition:

$$rank(\Theta_k) = \frac{n(n+1)}{2} + mn$$

- In practical, $l \geq n(n+1) + 2mn$
- The time interval for each learning section is $l \cdot \delta t$, so properly set *OpenControl.ADP_control.LTIController.num_data* and *OpenControl.ADP_control.LTIController.data_eval* to make sure the learning section not too slow

4.2.1 Algorithm



4.2.2 Library Usage

Setup a simulation section with *OpenControl.ADP_control.LTIController* and *OpenControl.ADP_control.LTIController.setPolicyParam()* then perform simulation by *OpenControl.ADP_control.LTIController.onPolicy()*

```
from OpenControl.ADP_control import LTIController

Ctrl = LTIController(sys)
# set parameters for policy
Q = np.eye(3); R = np.array([[1]]); K0 = np.zeros((1,3))
explore_noise=lambda t: 2*np.sin(10*t)
```

(continues on next page)

(continued from previous page)

```
data_eval = 0.1; num_data = 10

Ctrl.setPolicyParam(K0=K0, Q=Q, R=R, data_eval=data_eval, num_data=num_data, explore_
↪noise=explore_noise)
# take simulation and get the results
K, P = Ctrl.onPolicy()
```

4.3 Off-policy learning

Let define some new matrices

$$\delta_{xx} = \left[x \otimes x|_{t_1}^{t_1+\delta t}, \quad x \otimes x|_{t_2}^{t_2+\delta t}, \quad ..., \quad x \otimes x|_{t_l}^{t_l+\delta t}\right]^T$$

$$I_{xx} = \left[\int_{t_1}^{t_1+\delta t} x \otimes x d\tau, \quad \int_{t_2}^{t_2+\delta t} x \otimes x d\tau, \quad ..., \quad \int_{t_l}^{t_l+\delta t} x \otimes x d\tau\right]^T$$

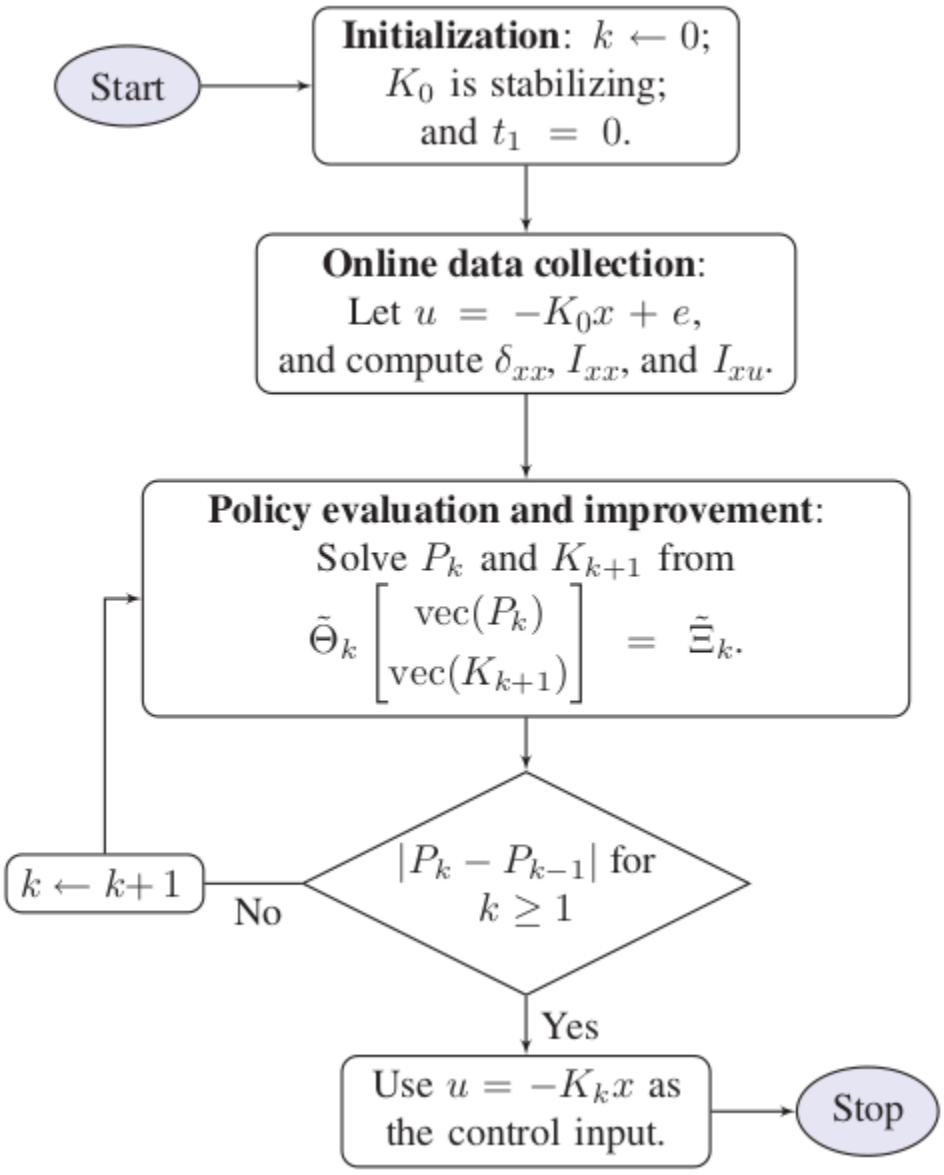
$$I_{xu} = \left[\int_{t_1}^{t_1+\delta t} x \otimes u_0 d\tau, \quad \int_{t_2}^{t_2+\delta t} x \otimes u_0 d\tau, \quad ..., \quad \int_{t_l}^{t_l+\delta t} x \otimes u_0 d\tau\right]^T$$

and $\Theta_k \in \mathbb{R}^{l\times(nn+mn)}, \Xi_k \in \mathbb{R}^l$ defined by:

$$\Theta_k = \left[\delta_{xx}, \quad -2I_{xx}(I_n \otimes K_k^T R) - 2I_{xu}(I_n \otimes R)\right]$$
$$\Xi_k = -I_{xx} vec(Q_k)$$

then for any given stabilizing gain matrix K_k, (4.1) implies the same matrix form as (4.2)

4.3.1 Algorithm



4.3.2 Library Usage

Setup a simulation section the same as the *section* then perform simulation by *OpenControl.ADP_control.LTIController.offPolicy()*

```
K, P = Ctrl.offPolicy()
```

CHAPTER

FIVE

NON-LINEAR ADP CONTROLLER

The below algorithms is given in the book Robust Adaptive Dynamic Programming.

5.1 Problem statements

Given the nonlinear system equation (3.2), design a control policy $u(t)$ that minimize the following cost function:

$$V(x) = \int_0^\infty r(x(t), u(t))dt, \qquad x(0) = x_0 \tag{5.1}$$

where $r(x, u) = q(x) + u^T R(x)u$ with $q(x)$ a positive definite function, and $R(x)$ is symmetric and positive definite for all $x \in \mathbb{R}^n$

If there exists a feedback control policy $u_0 : \mathbb{R}^n \to \mathbb{R}^m$ that globally asymtotically stabilizes the system (3.2) at the origin and the associated cost as defined in (5.1) and there exists a continous differentiable function V^* such that the Hamilton-Jacobi-Bellman (HJB) equation holds $H(V^*) = 0$ then the control policy

$$u^*(x) = -\frac{1}{2}R^{-1}(x)g^T(x)\nabla V^*(x) \tag{5.2}$$

globally asymptotically stabilizes (3.2) at $x = 0$, and u^* is also the optimal control policy

$$V^*(x) = min_u V(x, u) \qquad \forall x \in \mathbb{R}^n$$

5.2 Off Policy Learning

Consider the system with the folowing control policy

$$\dot{x} = f(x) + g(x)(u_0 + e) \tag{5.3}$$

where `u0` is the initial admissible control policy and e is the `exploration noise` then rewrite it as

$$\dot{x} = f(x) + g(x)u_i(x) + g(x)v_i \tag{5.4}$$

where $v_i = u_0 - u_i + e$.

Take the time derivative of $V_i(x)$ along (5.4) and integrate the result within interval $[t, t + T]$ to obtain:

$$V_i(x(t + T)) - V_i(x(t)) = -\int_t^{t+T} [q(x) + u_i^T R u_i + 2u_{i+1}^T R v_i]d\tau \tag{5.5}$$

Because the formula (5.2) requires the knowledge of the dynamic of the system, it makes practical application limited. One may use `neural networks` to approximate the control policy u_i (the `actor`)and the unknown cost function $V_i(x)$ (the `critic`).

$$V_i(x) \approx \sum_{j=1}^{N_1} Wc_{i,j}\phi_j(x)$$

$$u_{i+1}(x) \approx \sum_{j=1}^{N_2} Wa_{i,j}\psi_j(x)$$

where $\phi_j : \mathbb{R}^n \to \mathbb{R}$ and $\psi_j : \mathbb{R}^n \to \mathbb{R}^m$ are two sequences of linearly independent smooth basis function, Wc, Wa is the weight of the neural networks.

Replacing them to the (5.5) and transform the result into the matrix, we have

$$\begin{bmatrix} \Delta\phi^T & -2(I_{u\psi} - I_{\psi\psi}(Wa_i^T \otimes I_\phi)) \end{bmatrix} \begin{bmatrix} Wc_i^T \\ vec(Wa_{i+1}^T R) \end{bmatrix} = \begin{bmatrix} I_q + I_{\psi\psi} vec(Wa_i^T R Wa_i) \end{bmatrix} \tag{5.6}$$

where $Wc_i = \begin{bmatrix} Wc_{i,1}, & Wc_{i,2}, & ..., & Wc_{i,N_2} \end{bmatrix}^T$ and the same for $Wa_i, \phi(x), \psi(x)$ and

$$\Delta\phi = \phi(x(t+T)) - \phi(x(t)$$

$$I_q = \int_t^{t+T} q(x)d\tau$$

$$I_{u\psi} = \int_t^{t+T} (u_0 + e)^T \otimes \psi^T d\tau$$

$$I_{\psi\psi} = \int_t^{t+T} (\psi^T \otimes \psi^T) d\tau$$

$$I_\psi = np.eye(N_2)$$

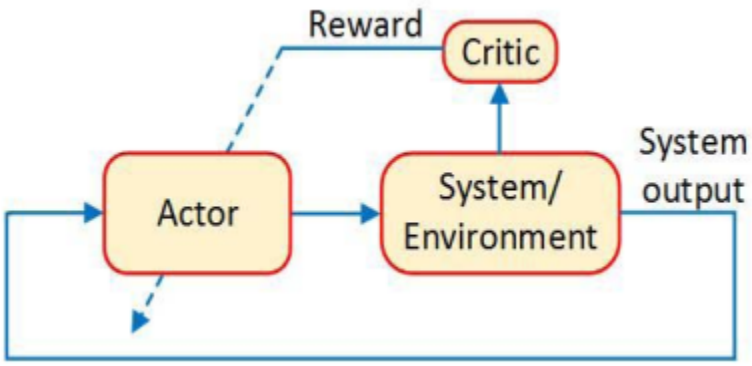
Finally, one can realize that the (5.6) is actually a fixpoint equation in the form

$$A(w_i)C(w_{i+1}) = B(w_i)$$

Note:

- the initial controller `u0` must be `admissible` controller
- the squences of basis functions $\phi_j(x), \psi_j(x)$ should be in the form of *linearly independent smooth*. Default basis functions is the `polynomial functions`, see *OpenControl.ADP_control.NonLinController.default_psi_func()* and *OpenControl.ADP_control.NonLinController.default_phi_func()*
- the time interval `T` for data collection must be larger than the sample time
- `number of data` $l >= n(n+1) + 2mn$
- the default function $q(x)$ is $x^T x$

5.2.1 Algorithm



5.2.2 Library Usage

Define a non-linear system like in *System representation*, then setup a simulation section by *OpenControl.ADP_control.NonLinController.setPolicyParam()* and perform simulation by *OpenControl.ADP_control.NonLinController.offPolicy()*

```
from OpenControl.ADP_control import NonLinController

##########define a controller##################
Ctrl = NonLinController(sys)
u0 = lambda x: 0      # the system is already globally stable
data_eval = 0.01; num_data = 80       # at leats n_phi+n_psi
explore_noise = lambda t: 0.2*np.sum(np.sin(np.array([1, 3, 7, 11, 13, 15])*t))

###############setup policy parameter############
Ctrl.setPolicyParam(data_eval=data_eval, num_data=num_data, explore_noise=explore_noise,␣
↪u0=u0)
################take simulation step##############
Wc, Wa = Ctrl.offPolicy()
```

then the optimal control policy is given by

```
uopt = lambda t,x: Wa.dot(Ctrl.psi_func(x))
```

CHAPTER

SIX

VISUALIZE

This class wrap the Pytorch Tensorboard class.

Present the simulation results on *Tensorboard* is so easy and interactive with

6.1 Create log file and log signal

```
from OpenControl.visualize import Logger

logX = Logger(log_dir='results')
for i in range(100):
    logX.log('euler', [i*np.cos(i), i*np.sin(i)], i)
logX.end_log()
```

6.2 Analysis simulation on Tensorboard

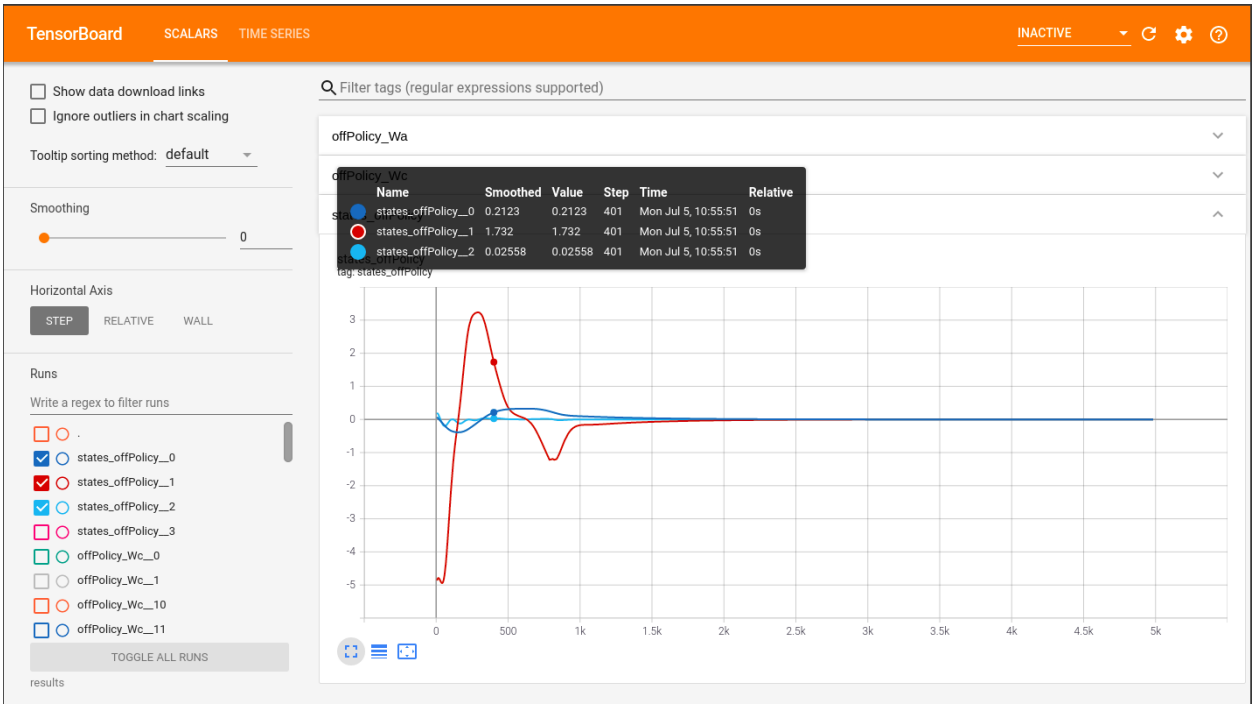
Note: If you run the code on Google Colab shell, just load Tensorboard first

```
%load_ext tensorboard
```

then run this shell to show

```
%tensorboard --logdir results
```

6.3 Usage



class OpenControl.visualize.**Logger**(*log_dir='results'*, *filename_suffix=''*)

Bases: `object`

Real-time visualize simulation results, using Tensorboard

writer

the same as SummaryWriter

Type class

__init__(*log_dir='results'*, *filename_suffix=''*)

The same as SummaryWriter

Parameters

- **log_dir** (*str, optional*) – the direction to save the log file. Defaults to 'results' folder.
- **filename_suffix** (*str, optional*) – suffix added to the name of log file. Defaults to ''.

end_log()

Call this function to end logging

log(*section*, *signals*, *step*)

Log the signals with the name of section in the step time

Parameters

- **section** (*str*) – name of signals
- **signals** (*array*) – signals
- **step** (*int*) – only *int* type is acceptable. Please convert *float* timestep to *int*

CHAPTER

SEVEN

EXAMPLES

The below examples is given in the paper *Neural network approach to continuous-time direct adaptive optimal control for partially unknown nonlinear systems*, the full source code is provided in here

Check out the other examples at https://github.com/VNOpenAI/OpenControl/tree/master/examples

Note: In this python implementation, we dont use the algorithm in the paper because the original one requires knowledge the system dynamics. Instead, we simulate with OpenControl, Off Policy ADP Controller. The results that is the same as the paper's results demonstrate our python packages.

7.1 Problem statements & Implementation

Consider the nonlinear system described be the equations:

$$\dot{x_1} = -x_1 + x_2$$
$$\dot{x_2} = -\frac{1}{2}(x_1 + x_2) + \frac{1}{2}x_2 sin^2(x_1) + sin(x_1)u$$

Define the system by code

```
# define system
def dot_x(t,x,u):
# dynamic of the system
x1 = x[0]
x2 = x[1]

# coefficient

# system dynamics
dx1 = -x1 + x2
dx2 = -0.5*(x1+x2)+0.5*x2*np.sin(x1)**2 + np.sin(x1)*u

dx = [dx1, dx2]
return dx

dimension = (2,1)
sys = NonLin(dot_x, dimension)

# setup simulation
```

(continues on next page)

(continued from previous page)

```
t_start = 0; t_stop = 20
x0 = np.array([-1, 1])
sample_time = 0.01

sys.setSimulationParam(t_sim=(t_start, t_stop), sample_time=sample_time, x0=x0)
```

The cost function $V(x) = \int_t^\infty (x_1^2 + x_2^2 + u^2) d\tau$ is approximated by the neural network which has basis function is $\phi(x) = [x_1^2 + x_1 x_2 + x_2^2]^T$. The initial stabilizing controller was taken as $u_0(x) = -\frac{3}{2} sin(x_1)(x_1 + x_2)$ applied for 30 time intervals $T = 0.1s$.

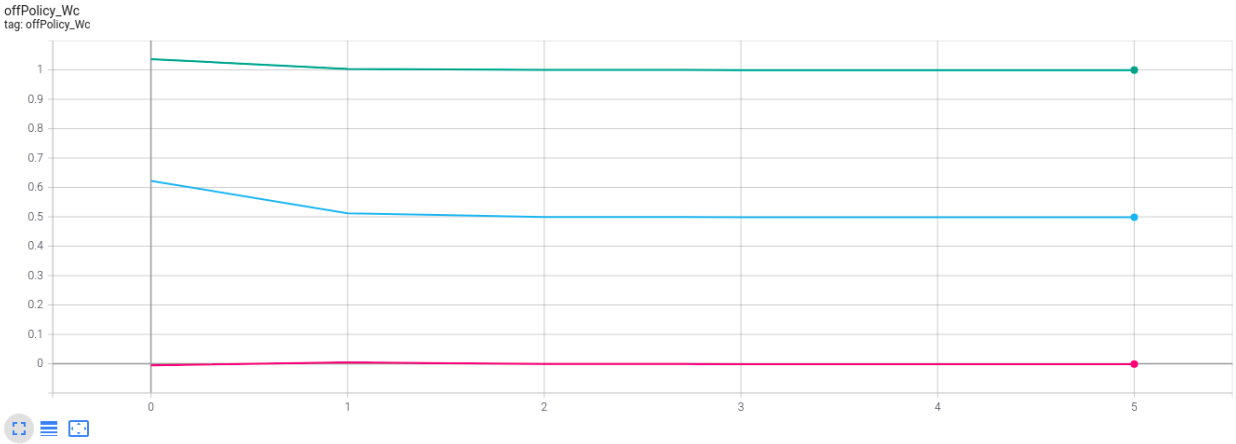
Design controller in code:

```
# define and setup controller
ctrl = NonLinController(sys)
u0 = lambda x: -1.5*np.sin(x[0])*(x[0] + x[1])
data_eval = 0.1; num_data = 30
explore_noise = lambda t: 0.2*np.sum(np.sin(np.array([1, 3, 7, 11, 13, 15])*t))

phi_func = lambda x: np.array([x[0]**2, x[0]*x[1], x[1]**2])

ctrl.setPolicyParam(data_eval=data_eval, num_data=num_data, explore_noise=explore_noise,␣
↪u0=u0, phi_func=phi_func)
```

Then run the simulation:

```
# run simuilation
Wc, Wa = ctrl.offPolicy()
print("The optimal weight of the critic")
print(Wc)
# for visualizing the results, using tensorboard
```

Analysis simulation with Tensorboard



For quick run, see the notebook file

CHAPTER

EIGHT

API REFERENCE

This page contains auto-generated API reference documentation[1].

8.1 OpenControl

8.1.1 Subpackages

OpenControl.ADP_control

Submodules

OpenControl.ADP_control.controller

Module Contents

Classes

LTIController	This present continuous controller for LTI System
NonLinController	This present continuous controller for Non-Linear System

class OpenControl.ADP_control.controller.**LTIController**(*system*, *log_dir='results'*)

This present continuous controller for LTI System

system

the object of LTI class

Type LTI class

log_dir

the folder include all log files. Defaults to 'results'.

Type string, optional

logX

the object of Logger class, use for logging state signals

Type Logger class

[1] Created with sphinx-autoapi

K0
The initial value of K matrix. Defaults to np.zeros((m,n)).

Type mxn array, optional

Q
The Q matrix. Defaults to 1.

Type nxn array, optional

R
The R matrix. Defaults to 1.

Type mxm array, optional

data_eval
data_eval x num_data = time interval for each policy updation. Defaults to 0.1.

Type float, optional

num_data
the number of data for each learning iteration. Defaults to 10.

Type int, optional

explore_noise
The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Type func(t), optional

logK
logger of the K matrix

Type Logger class

logP
logger of the P matrix

Type Logger class

t_plot, x_plot
use for logging, plotting simulation result

Type float, array

viz
True for visualize results on `Tensorboard`. Default to True

Type boolean

step(*self*, *x0*, *u*, *t_span*)
Step respond of the system.

Parameters

- **x0** (*1D array*) – initial state for simulation
- **u** (*1D array*) – the value of input within t_span
- **t_span** (*list*) – (t_start, t_stop)

Returns t_span, state at t_span (x_start, x_stop)

Return type list, 2D array

LQR(*self*, *Q=None*, *R=None*)
This function solve Riccati function with defined value function

Parameters

- **Q** (*nxn array optional*) – the Q matrix. Defaults to 1.
- **R** (*mxm arary, optional*) – the R matrix. Defaults to 1.

Returns the K, P matrix

Return type mxn array, nxn array

_isStable(*self*, *A*)

onPolicy(*self*, *stop_thres=0.001*, *viz=True*)

Using On-policy approach to find optimal adaptive feedback controller, requires only the dimension of the system

Parameters

- **stop_thres** (*float, optional*) – threshold value to stop iteration. Defaults to 1e-3.
- **viz** (*bool, optional*) – True for logging data. Defaults to True.

Raises **ValueError** – raise when the user-defined number of data is not enough, make rank condition unsatisfied

Returns the optimal K, P matrix

Return type mxn array, nxn array

_afterGainKopt(*self*, *t_plot*, *x_plot*, *Kopt*, *section*)

_rowGainOnPloicy(*self*, *K*, *x_sample*, *t_sample*)

setPolicyParam(*self*, *K0=None*, *Q=None*, *R=None*, *data_eval=0.1*, *num_data=10*, *explore_noise=lambda t: ...*)

Setup policy parameters for both the On (Off) policy algorithms. Initalize logger for K, P matrix

Parameters

- **K0** (*mxn array, optional*) – The initial value of K matrix. Defaults to np.zeros((m,n)).
- **Q** (*nxn array, optional*) – The Q matrix. Defaults to 1.
- **R** (*mxm array, optional*) – The R matrix. Defaults to 1.
- **data_eval** (*float, optional*) – data_eval x num_data = time interval for each policy updation. Defaults to 0.1.
- **num_data** (*int, optional*) – the number of data for each learning iteration. Defaults to 10.
- **explore_noise** (*func(t), optional*) – The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Raises **ValueError** – raise when the initial value of the K matrix is not admissible

---

Note:

- The K0 matrix must be admissible
- data_eval must be larger than the sample_time
- num_data >= n(n+1) + 2mn

---

offPolicy(*self*, *stop_thres=0.001*, *max_iter=30*, *viz=True*)

Using Off-policy approach to find optimal adaptive feedback controller, requires only the dimension of the system

Parameters

- **stop_thres** (*float, optional*) – threshold value to stop iteration. Defaults to 1e-3.
- **viz** (*bool, optional*) – True for logging data. Defaults to True.
- **max_iter** (*int, optional*) – the maximum number of policy iterations. Defaults to 30.

Raises **ValueError** – raise when the user-defined number of data is not enough, make rank condition unsatisfied

Returns the optimal K, P matrix

Return type mxn array, nxn array

_policyEval(*self*, *dxx*, *Ixx*, *Ixu*)

_getRowOffPolicyMatrix(*self*, *t_sample*, *x_sample*)

class OpenControl.ADP_control.controller.**NonLinController**(*system*, *log_dir='results'*)

This present continuous controller for Non-Linear System

system

the object of nonLin class

Type nonLin class

log_dir

the folder include all log files. Defaults to 'results'.

Type string, optional

logX

the object of Logger class, use for logging state signals

Type Logger class

u0

The initial feedback control policy. Defaults to 0.

Type func(x), optional

q_func

the function $q(x)$. Defaults to nonLinController.default_q_func.

Type func(x), optional

R

The R matrix. Defaults to 1.

Type mxm array, optional

phi_func

the sequences of basis function to approximate critic, $\phi_j(x)$. Defaults to nonLinController.default_phi_func

Type list of func(x), optional

psi_func

the sequences of basis function to approximate actor, $\psi_j(x)$. Defaults to nonLinController.default_psi_func

Type list of func(x), optional

data_eval
data_eval x num_data = time interval for each policy updation. Defaults to 0.1.

Type float, optional

num_data
the number of data for each learning iteration. Defaults to 10.

Type int, optional

explore_noise
The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Type func(t), optional

logWa
logging to value of the weight of the actor

Type Logger class

logWc
logging to value of the weight of the critic

Type Logger class

t_plot, x_plot
use for logging, plotting simulation result

Type float, array

viz
True for visualize results on `Tensorboard`. Default to True

Type boolean

setPolicyParam(*self*, *q_func=None*, *R=None*, *phi_func=None*, *psi_func=None*, *u0=lambda x: ...*, *data_eval=0.1*, *num_data=10*, *explore_noise=lambda t: ...*)
Setup policy parameters for both the On (Off) policy algorithms. Initalize logger for K, P matrix

Parameters

- **q_func** (*func(x), optional*) – the function $q(x)$. Defaults to nonLinController.default_q_func
- **R** (*mxm array, optional*) – The R matrix. Defaults to 1.
- **phi_func** (*list of func(x), optional*) – the sequences of basis function to approximate critic, $\phi_j(x)$. Defaults to nonLinController.default_phi_func
- **psi_func** (*list of func(x), optional*) – the sequences of basis function to approximate actor, $\psi_j(x)$. Defaults to nonLinController.default_psi_func
- **u0** (*func(x), optional*) – The initial feedback control policy. Defaults to 0.
- **data_eval** (*float, optional*) – data_eval x num_data = time interval for each policy updation. Defaults to 0.1.
- **num_data** (*int, optional*) – the number of data for each learning iteration. Defaults to 10.
- **explore_noise** (*func(t), optional*) – The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Note:

- u0 must be admissible controller
- the squences of basis functions $\phi_j(x), \psi_j(x)$ should be in the form of *linearly independent smooth*
- data_eval must be larger than the sample_time
- num_data >= n(n+1) + 2mn

---

step(*self*, *dot_x*, *x0*, *t_span*)
Step respond of the no-input system

Parameters

- **dot_x** (*func(x)*) – no-input ODEs function
- **x0** (*1D array*) – the initial state
- **t_span** (*tuple*) – (t_start, t_stop)

Returns t_span, state at t_span (x_start, x_stop)

Return type list, 2D array

feedback(*self*, *viz=True*)
Check stability of the initial control policy u0

Parameters **viz** (*boolean*) – True for visualize results on Tensorboard. Default to True

Returns t_plot and x_plot

Return type list, 2D array

offPolicy(*self*, *stop_thres=0.001*, *max_iter=30*, *viz=True*)
Using Off-policy approach to find optimal adaptive feedback controller, requires only the dimension of the system

Parameters

- **stop_thres** (*float, optional*) – threshold value to stop iteration. Defaults to 1e-3.
- **viz** (*boolean*) – True for visualize results on Tensorboard. Default to True
- **unlearned_compare** (*boolean*) – True to log unlearned states data, for comparision purpose.
- **max_iter** (*int, optional*) – the maximum number of policy iterations. Defaults to 30.

Returns the final updated weight of critic, actor neural nets.

Return type array, array

_unlearn_controller(*self*, *t_plot*, *x_plot*, *section*)

_afterGainWopt(*self*, *t_plot*, *x_plot*, *Waopt*, *section*)

_policyEval(*self*, *dphi*, *Iq*, *Iupsi*, *Ipsipsi*)

_getRowOffPolicyMatrix(*self*, *t_sample*, *x_sample*)

static default_psi_func(*x*)
The default sequences of basis functions to approximate actor

Parameters **x** (*1xn array*) – the state vector

Returns the polynomial basis function. If $x = [x_1, x_2]^T$ then $\psi(x) = [x_1, x_2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3]^T$

Return type list func(x)

static default_phi_func(*x*)
The default sequences of basis functions to approximate critic

Parameters **x** (*1xn array*) – the state vector

Returns the polynomial basis function. If $x = [x_1, x_2]^T$ then $\phi(x) = [x_1^2, x_1x_2, x_2^2, x_1^4, x_1^2x_2^2, x_2^4]^T$

Return type list func(x)

static default_q_func(*x*)
The default function of the q(x) function

Parameters **x** (*1D array*) – the state vector

Returns $x^T x$

Return type float

OpenControl.ADP_control.system

Module Contents

Classes

LTI	This class present state-space LTI system.
NonLin	This class represent non-linear system by ODEs.

class OpenControl.ADP_control.system.**LTI**(*A*, *B*, *C=1*, *D=0*)
This class present state-space LTI system.

dimension
(n_state, n_input).

Type tuple

model
{A, B, C, D, dimension}.

Type dict

max_step
define max step for ODEs solver algorithms. Defaults to 1e-3.

Type float, optional

algo
RK45, RK23 or DOP853 . Defaults to 'RK45'.

Type str, optional

t_sim
time for simualtion (start, stop). Defaults to (0,10).

Type tuple, optional

x0
the initial state. Defaults to np.ones((n,)).

Type 1xn array, optional

sample_time
the sample time. Defaults to 1e-2.

Type float, optional

_check_model(*self*)

setSimulationParam(*self*, *max_step=0.001*, *algo='RK45'*, *t_sim=(0, 10)*, *x0=None*, *sample_time=0.01*)
Run this function before any simulations

Parameters

- **max_step** (*float, optional*) – define max step for ODEs solver algorithms. Defaults to 1e-3.
- **algo** (*str, optional*) – RK45, RK23 or DOP853 . Defaults to 'RK45'.
- **t_sim** (*tuple, optional*) – time for simualtion (start, stop). Defaults to (0,10).
- **x0** (*1xn array, optional*) – the initial state. Defaults to np.ones((n,)).
- **sample_time** (*float, optional*) – the sample time. Defaults to 1e-2.

integrate(*self*, *x0*, *u*, *t_span*)

class OpenControl.ADP_control.system.**NonLin**(*dot_x*, *dimension*)
This class represent non-linear system by ODEs.

dot_x
the dx/dt function, return 1D array output

Type func(t,x,u)

dimension
(n_state, n_input)

Type tuple

max_step
define max step for ODEs solver algorithms. Defaults to 1e-3.

Type float, optional

algo
RK45, RK23 or DOP853 . Defaults to 'RK45'.

Type str, optional

t_sim
time for simualtion (start, stop). Defaults to (0,10).

Type tuple, optional

x0
the initial state. Defaults to np.ones((n,)).

Type 1xn array, optional

sample_time
the sample time. Defaults to 1e-2.

Type float, optional

setSimulationParam(*self*, *max_step=0.001*, *algo='RK45'*, *t_sim=(0, 10)*, *x0=None*, *sample_time=0.01*)
Run this function before any simulations

Parameters

- **max_step** (*float, optional*) – define max step for ODEs solver algorithms. Defaults to 1e-3.
- **algo** (*str, optional*) – RK45, RK23 or DOP853 . Defaults to 'RK45'.
- **t_sim** (*tuple, optional*) – time for simualtion (start, stop). Defaults to (0,10).
- **x0** (*1xn array, optional*) – the initial state. Defaults to np.ones((n,)).
- **sample_time** (*float, optional*) – the sample time. Defaults to 1e-2.

integrate(*self*, *x0*, *u*, *t_span*, *t_eval=None*)

OpenControl.ADP_control.testcase_linearSystem

Module Contents

OpenControl.ADP_control.testcase_linearSystem.**A0**

OpenControl.ADP_control.testcase_linearSystem.**B0**

OpenControl.ADP_control.testcase_linearSystem.**A1**

OpenControl.ADP_control.testcase_linearSystem.**B1**

OpenControl.ADP_control.testcase_linearSystem.**A2**

OpenControl.ADP_control.testcase_linearSystem.**B2**

OpenControl.ADP_control.testcase_linearSystem.**A3**

OpenControl.ADP_control.testcase_linearSystem.**B3**

OpenControl.ADP_control.testcase_linearSystem.**A4**

OpenControl.ADP_control.testcase_linearSystem.**B4**

OpenControl.ADP_control.testcase_linearSystem.**A5**

OpenControl.ADP_control.testcase_linearSystem.**B5**

OpenControl.ADP_control.testcase_linearSystem.**A6**

OpenControl.ADP_control.testcase_linearSystem.**B6**

OpenControl.ADP_control.testcase_linearSystem.**A7**

OpenControl.ADP_control.testcase_linearSystem.**B7**

OpenControl.ADP_control.testcase_linearSystem.**A8**

OpenControl.ADP_control.testcase_linearSystem.**B8**

OpenControl.ADP_control.testcase_linearSystem.**A9**

OpenControl.ADP_control.testcase_linearSystem.**B9**

OpenControl.ADP_control.testcase_linearSystem.**A10**

OpenControl.ADP_control.testcase_linearSystem.**B10**

OpenControl.ADP_control.testcase_linearSystem.**A11**

OpenControl.ADP_control.testcase_linearSystem.**B11**

OpenControl.ADP_control.testcase_linearSystem.**all_test_case = [None, None, None, None, None, None, None, None, None, None, None, None]**

Package Contents

Classes

LTI	This class present state-space LTI system.
NonLin	This class represent non-linear system by ODEs.
LTIController	This present continuous controller for LTI System
NonLinController	This present continuous controller for Non-Linear System
Logger	Real-time visualize simulation results, using Tensorboard

class OpenControl.ADP_control.**LTI**(*A*, *B*, *C=1*, *D=0*)

This class present state-space LTI system.

dimension

(n_state, n_input).

Type tuple

model

{A, B, C, D, dimension}.

Type dict

max_step

define max step for ODEs solver algorithms. Defaults to 1e-3.

Type float, optional

algo

RK45, RK23 or DOP853 . Defaults to 'RK45'.

Type str, optional

t_sim

time for simualtion (start, stop). Defaults to (0,10).

Type tuple, optional

x0

the initial state. Defaults to np.ones((n,)).

Type 1xn array, optional

sample_time

the sample time. Defaults to 1e-2.

Type float, optional

_check_model(*self*)

setSimulationParam(*self*, *max_step=0.001*, *algo='RK45'*, *t_sim=(0, 10)*, *x0=None*, *sample_time=0.01*)

Run this function before any simulations

Parameters

- **max_step** (*float, optional*) – define max step for ODEs solver algorithms. Defaults to 1e-3.
- **algo** (*str, optional*) – RK45, RK23 or DOP853 . Defaults to 'RK45'.

- **t_sim** (*tuple, optional*) – time for simualtion (start, stop). Defaults to (0,10).
- **x0** (*1xn array, optional*) – the initial state. Defaults to np.ones((n,)).
- **sample_time** (*float, optional*) – the sample time. Defaults to 1e-2.

integrate(*self*, *x0*, *u*, *t_span*)

class OpenControl.ADP_control.**NonLin**(*dot_x*, *dimension*)

This class represent non-linear system by ODEs.

dot_x

the dx/dt function, return 1D array output

Type func(t,x,u)

dimension

(n_state, n_input)

Type tuple

max_step

define max step for ODEs solver algorithms. Defaults to 1e-3.

Type float, optional

algo

RK45, RK23 or DOP853 . Defaults to 'RK45'.

Type str, optional

t_sim

time for simualtion (start, stop). Defaults to (0,10).

Type tuple, optional

x0

the initial state. Defaults to np.ones((n,)).

Type 1xn array, optional

sample_time

the sample time. Defaults to 1e-2.

Type float, optional

setSimulationParam(*self*, *max_step=0.001*, *algo='RK45'*, *t_sim=(0, 10)*, *x0=None*, *sample_time=0.01*)

Run this function before any simulations

Parameters

- **max_step** (*float, optional*) – define max step for ODEs solver algorithms. Defaults to 1e-3.
- **algo** (*str, optional*) – RK45, RK23 or DOP853 . Defaults to 'RK45'.
- **t_sim** (*tuple, optional*) – time for simualtion (start, stop). Defaults to (0,10).
- **x0** (*1xn array, optional*) – the initial state. Defaults to np.ones((n,)).
- **sample_time** (*float, optional*) – the sample time. Defaults to 1e-2.

integrate(*self*, *x0*, *u*, *t_span*, *t_eval=None*)

class OpenControl.ADP_control.**LTIController**(*system*, *log_dir='results'*)

This present continuous controller for LTI System

system
the object of LTI class

Type LTI class

log_dir
the folder include all log files. Defaults to 'results'.

Type string, optional

logX
the object of Logger class, use for logging state signals

Type Logger class

K0
The initial value of K matrix. Defaults to np.zeros((m,n)).

Type mxn array, optional

Q
The Q matrix. Defaults to 1.

Type nxn array, optional

R
The R matrix. Defaults to 1.

Type mxm array, optional

data_eval
data_eval x num_data = time interval for each policy updation. Defaults to 0.1.

Type float, optional

num_data
the number of data for each learning iteration. Defaults to 10.

Type int, optional

explore_noise
The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Type func(t), optional

logK
logger of the K matrix

Type Logger class

logP
logger of the P matrix

Type Logger class

t_plot, x_plot
use for logging, plotting simulation result

Type float, array

viz
True for visualize results on `Tensorboard`. Default to True

Type boolean

step(*self*, *x0*, *u*, *t_span*)
Step respond of the system.

Parameters

- **x0** (*1D array*) – initial state for simulation
- **u** (*1D array*) – the value of input within t_span
- **t_span** (*list*) – (t_start, t_stop)

Returns t_span, state at t_span (x_start, x_stop)

Return type list, 2D array

LQR(*self*, *Q=None*, *R=None*)

This function solve Riccati function with defined value function

Parameters

- **Q** (*nxn array optional*) – the Q matrix. Defaults to 1.
- **R** (*mxm arary, optional*) – the R matrix. Defaults to 1.

Returns the K, P matrix

Return type mxn array, nxn array

_isStable(*self*, *A*)

onPolicy(*self*, *stop_thres=0.001*, *viz=True*)

Using On-policy approach to find optimal adaptive feedback controller, requires only the dimension of the system

Parameters

- **stop_thres** (*float, optional*) – threshold value to stop iteration. Defaults to 1e-3.
- **viz** (*bool, optional*) – True for logging data. Defaults to True.

Raises **ValueError** – raise when the user-defined number of data is not enough, make rank condition unsatisfied

Returns the optimal K, P matrix

Return type mxn array, nxn array

_afterGainKopt(*self*, *t_plot*, *x_plot*, *Kopt*, *section*)

_rowGainOnPloicy(*self*, *K*, *x_sample*, *t_sample*)

setPolicyParam(*self*, *K0=None*, *Q=None*, *R=None*, *data_eval=0.1*, *num_data=10*, *explore_noise=lambda t: ...*)

Setup policy parameters for both the On (Off) policy algorithms. Initalize logger for K, P matrix

Parameters

- **K0** (*mxn array, optional*) – The initial value of K matrix. Defaults to np.zeros((m,n)).
- **Q** (*nxn array, optional*) – The Q matrix. Defaults to 1.
- **R** (*mxm array, optional*) – The R matrix. Defaults to 1.
- **data_eval** (*float, optional*) – data_eval x num_data = time interval for each policy updation. Defaults to 0.1.
- **num_data** (*int, optional*) – the number of data for each learning iteration. Defaults to 10.
- **explore_noise** (*func(t), optional*) – The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Raises `ValueError` – raise when the initial value of the K matrix is not admissible

---

Note:

- The K0 matrix must be admissible
- data_eval must be larger than the sample_time
- num_data >= n(n+1) + 2mn

`offPolicy`(*self*, *stop_thres=0.001*, *max_iter=30*, *viz=True*)

Using Off-policy approach to find optimal adaptive feedback controller, requires only the dimension of the system

Parameters

- `stop_thres` (*float, optional*) – threshold value to stop iteration. Defaults to 1e-3.
- `viz` (*bool, optional*) – True for logging data. Defaults to True.
- `max_iter` (*int, optional*) – the maximum number of policy iterations. Defaults to 30.

Raises `ValueError` – raise when the user-defined number of data is not enough, make rank condition unsatisfied

Returns the optimal K, P matrix

Return type mxn array, nxn array

`_policyEval`(*self*, *dxx*, *Ixx*, *Ixu*)

`_getRowOffPolicyMatrix`(*self*, *t_sample*, *x_sample*)

`class` OpenControl.ADP_control.`NonLinController`(*system*, *log_dir='results'*)

This present continuous controller for Non-Linear System

`system`

the object of nonLin class

Type nonLin class

`log_dir`

the folder include all log files. Defaults to 'results'.

Type string, optional

`logX`

the object of Logger class, use for logging state signals

Type Logger class

`u0`

The initial feedback control policy. Defaults to 0.

Type func(x), optional

`q_func`

the function $q(x)$. Defaults to nonLinController.default_q_func.

Type func(x), optional

`R`

The R matrix. Defaults to 1.

Type mxm array, optional

phi_func
the sequences of basis function to approximate critic, $\phi_j(x)$. Defaults to nonLinController.default_phi_func

Type list of func(x), optional

psi_func
the sequences of basis function to approximate actor, $\psi_j(x)$. Defaults to nonLinController.default_psi_func

Type list of func(x), optional

data_eval
data_eval x num_data = time interval for each policy updation. Defaults to 0.1.

Type float, optional

num_data
the number of data for each learning iteration. Defaults to 10.

Type int, optional

explore_noise
The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Type func(t), optional

logWa
logging to value of the weight of the actor

Type Logger class

logWc
logging to value of the weight of the critic

Type Logger class

t_plot, x_plot
use for logging, plotting simulation result

Type float, array

viz
True for visualize results on `Tensorboard`. Default to True

Type boolean

setPolicyParam(*self*, *q_func=None*, *R=None*, *phi_func=None*, *psi_func=None*, *u0=lambda x: ...*, *data_eval=0.1*, *num_data=10*, *explore_noise=lambda t: ...*)
Setup policy parameters for both the On (Off) policy algorithms. Initalize logger for K, P matrix

Parameters

- **q_func** (*func(x), optional*) – the function $q(x)$. Defaults to nonLinController.default_q_func
- **R** (*mxm array, optional*) – The R matrix. Defaults to 1.
- **phi_func** (*list of func(x), optional*) – the sequences of basis function to approximate critic, $\phi_j(x)$. Defaults to nonLinController.default_phi_func
- **psi_func** (*list of func(x), optional*) – the sequences of basis function to approximate actor, $\psi_j(x)$. Defaults to nonLinController.default_psi_func
- **u0** (*func(x), optional*) – The initial feedback control policy. Defaults to 0.
- **data_eval** (*float, optional*) – data_eval x num_data = time interval for each policy updation. Defaults to 0.1.

- **num_data** (*int, optional*) – the number of data for each learning iteration. Defaults to 10.
- **explore_noise** (*func(t), optional*) – The exploration noise within the learning stage. Defaults to lambda t:2*np.sin(100*t).

Note:

- u0 must be admissible controller
- the squences of basis functions $\phi_j(x), \psi_j(x)$ should be in the form of *linearly independent smooth*
- data_eval must be larger than the sample_time
- num_data >= n(n+1) + 2mn

---

step(*self*, *dot_x*, *x0*, *t_span*)
Step respond of the no-input system

Parameters

- **dot_x** (*func(x)*) – no-input ODEs function
- **x0** (*1D array*) – the initial state
- **t_span** (*tuple*) – (t_start, t_stop)

Returns t_span, state at t_span (x_start, x_stop)

Return type list, 2D array

feedback(*self*, *viz=True*)
Check stability of the initial control policy u0

Parameters **viz** (*boolean*) – True for visualize results on Tensorboard. Default to True

Returns t_plot and x_plot

Return type list, 2D array

offPolicy(*self*, *stop_thres=0.001*, *max_iter=30*, *viz=True*)
Using Off-policy approach to find optimal adaptive feedback controller, requires only the dimension of the system

Parameters

- **stop_thres** (*float, optional*) – threshold value to stop iteration. Defaults to 1e-3.
- **viz** (*boolean*) – True for visualize results on Tensorboard. Default to True
- **unlearned_compare** (*boolean*) – True to log unlearned states data, for comparision purpose.
- **max_iter** (*int, optional*) – the maximum number of policy iterations. Defaults to 30.

Returns the final updated weight of critic, actor neural nets.

Return type array, array

_unlearn_controller(*self*, *t_plot*, *x_plot*, *section*)

_afterGainWopt(*self*, *t_plot*, *x_plot*, *Waopt*, *section*)

_policyEval(*self*, *dphi*, *Iq*, *Iupsi*, *Ipsipsi*)

_getRowOffPolicyMatrix(*self*, *t_sample*, *x_sample*)

static default_psi_func(*x*)
The default sequences of basis functions to approximate actor

Parameters **x** (*1xn array*) – the state vector

Returns the polynomial basis function. If $x = [x_1, x_2]^T$ then $\psi(x) = [x_1, x_2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3]^T$

Return type list func(x)

static default_phi_func(*x*)
The default sequences of basis functions to approximate critic

Parameters **x** (*1xn array*) – the state vector

Returns the polynomial basis function. If $x = [x_1, x_2]^T$ then $\phi(x) = [x_1^2, x_1x_2, x_2^2, x_1^4, x_1^2x_2^2, x_2^4]^T$

Return type list func(x)

static default_q_func(*x*)
The default function of the q(x) function

Parameters **x** (*1D array*) – the state vector

Returns x^Tx

Return type float

class OpenControl.ADP_control.**Logger**(*log_dir='results'*, *filename_suffix=''*)
Bases: object

Real-time visualize simulation results, using Tensorboard

writer
the same as SummaryWriter

Type class

log(*self*, *section*, *signals*, *step*)
Log the signals with the name of section in the step time

Parameters

- **section** (*str*) – name of signals
- **signals** (*array*) – signals
- **step** (*int*) – only *int* type is acceptable. Please convert *float* timestep to *int*

end_log(*self*)
Call this function to end logging

OpenControl.classiccontrol

Submodules

OpenControl.classiccontrol.bibo

Module Contents

Classes

BIBO	Abtract class for implement the algorithms. Which dertermine
Gerschgorin	Abtract class for implement the algorithms. Which dertermine
Lyapunov	Abtract class for implement the algorithms. Which dertermine
Hurwitz	Abtract class for implement the algorithms. Which dertermine

class OpenControl.classiccontrol.bibo.**BIBO**
Bases: abc.ABC

Abtract class for implement the algorithms. Which dertermine the stability of system

abstract conclusion(*self*) → int

conclude about the stability of system. This function implement the algorithms.

Returns

if 1, system is stable if 0, this algorithms unable to conclude about stability os system if -1, system is unstable

Return type [int]

class OpenControl.classiccontrol.bibo.**Gerschgorin**(*A: numpy.ndarray*)
Bases: *BIBO*

Abtract class for implement the algorithms. Which dertermine the stability of system

conclusion(*self*) → int

conclude about the stability of system. This function implement the algorithms.

Returns

if 1, system is stable if 0, this algorithms unable to conclude about stability os system if -1, system is unstable

Return type [int]

class OpenControl.classiccontrol.bibo.**Lyapunov**(*A*, *P=None*, *Q=None*)
Bases: *BIBO*

Abtract class for implement the algorithms. Which dertermine the stability of system

static is_definite_positive_matrix(*M*)
Determine a maxtrix if

Parameters **M** – 2D-ndarray

Returns True if matrix is definite positive

Return type [Boolean]

conclusion(*self*)

conclude about the stability of system. This function implement the algorithms.

Returns

if 1, system is stable if 0, this algorithms unable to conclude about stability os system if -1, system is unstable

Return type [int]

class OpenControl.classiccontrol.bibo.**Hurwitz**(*A*)
Bases: *BIBO*

Abtract class for implement the algorithms. Which dertermine the stability of system

conclusion(*self*)

conclude about the stability of system. This function implement the algorithms.

Returns

if 1, system is stable if 0, this algorithms unable to conclude about stability os system if -1, system is unstable

Return type [int]

OpenControl.classiccontrol.controller

Module Contents

Classes

StateFeedBackController	Abstract class for controller based on state feedback .asdkjasd
PoleStatement	Design controller, move the eigvalues of system to desire.
LQR	this class implement Optimal control ,LQR

class OpenControl.classiccontrol.controller.**StateFeedBackController**
Bases: abc.ABC

Abstract class for controller based on state feedback .asdkjasd

abstract compute(*self*)
Return the matrix of controller

class OpenControl.classiccontrol.controller.**PoleStatement**(*pole*, *system*, *algo='Arckerman'*)
Bases: *StateFeedBackController*

Design controller, move the eigvalues of system to desire. this class implement 3 method. Roppernecker, Arckerman, Modal

_Roppernecker(*self*)
This function Implement Controller, Designed based on Ropernecker algorithms

Raises **ValueError** – if system has no matrix B

Returns 2D array, matrix of controller. got the shape (input_shape,state_shape)

Return type [np.ndarray]

_Arckerman(*self*)

Implement the arckerman method.

Raises **ValueError** – if B is None or system declared without input matrix (B)

Returns the controller matrix. Result of algorithms. see _Roppernecker funtion for mor detail.

Return type [nd.array]

_Modal(*self*)

compute(*self*)

the system instance call this method to get the controller matrix, designed based on selected algorithms.

Returns [description]

Return type [np.ndarray]

class OpenControl.classiccontrol.controller.**LQR**(*system*, *E*, *F*)

Bases: *StateFeedBackController*

this class implement Optimal control ,LQR

compute(*self*)

Return the matrix of controller

OpenControl.classiccontrol.linearsystem

Module Contents

Classes

LTI	main object

class OpenControl.classiccontrol.linearsystem.**LTI**(***kwargs*)

main object #dimension: ndim of state,input and output vector

Raises

- **assert** – [description]
- **ValueError** – [description]
- **ValueError** – [description]

Returns [description]

Return type [type]

bibo_result

property states_shape(*self*) → int

property inputs_shape(*self*) → int

property outputs_shape(*self*) → int

property max_step(*self*)

property A(*self*)

property B(*self*)

property C(*self*)

property D(*self*)

property dimension(*self*) → list
An attributes of system

Returns got the length 3, dimention of

Return type list

eigvals(*self*)
Compute the eigen values of system matrix (matrix A)

Returns [1D array of eigvalues]

Return type [np.ndarray]

is_stable(*self*, *algorimth='hurwitz'*, ***kwagrs*) → int
[Compute the stability of system]

Parameters **algorimth** (*str, optional*) – [select the algorithms to determine stability of system]. Defaults to 'hurwitz'.

Returns

1 - if system is stable

0 - if selected algorithms can't conclude about stability of system

-1 - if system is unstable

Return type int

is_controlable(*self*, *algorimth='kalman'*, ***kwagrs*) → bool
Determine the controllability of system.

Parameters **algorimth** (*str, optional*) – select the algorithms to determine controllability of system. Defaults to 'kalman'.

Raises **ValueError** – if the input matrix (matrix B) not found

Returns True if system is controlalbe

Return type bool

is_observable(*self*, *algorimth='kalman'*) → bool
Determine the observability of system.

Parameters **algorimth** (*str, optional*) – select the algorithms to determine observability of system. Defaults to 'kalman'.

Raises **ValueError** – if the output matrix (matrix C) not found

Returns True is system is observable

Return type bool

`setup_simulink`(*self*, *max_step=0.001*, *algo='RK45'*, *t_sim=(0, 10)*, *x0=None*, *sample_time=0.01*, *z0=None*)

Run this function before any simulations. This method set the necessary params for running simulation.

Parameters

- **max_step** (*float, optional*) – define max step for ODEs solver algorithms. Defaults to 1e-3.
- **algo** (*str, optional*) – RK45, RK23 or DOP853 . Defaults to 'RK45'.
- **t_sim** (*tuple, optional*) – time for simualtion (start, stop). Defaults to (0,10).
- **x0** (*1xn array, optional*) – the initial state. Defaults to np.ones((n,)).
- **sample_time** (*float, optional*) – the sample time. Defaults to 1e-2.

`step_response`(*self*, *input_function=None*, *logs_file=None*)

simulate behavior of Opne-Loop system Run self.setup_simulink() before running this this method

Parameters

- **input_function** (*[function], optional*) – address of input funtions, this funtion take a prams - t, and return the values of input in time t. Defaults to None.
- **logs_file** (*[string], optional*) – [path to logs folder,]. Defaults to None.

Returns

2D-np.ndarray time series to simulate

x_out: 2D-np.ndarray state of system in time series ,between self.t_sim

y_out: 2D-np.ndarray output of system in time series,

Return type time_array

`apply_state_feedback`(*self*, *R*, *input_function=None*, *logs_file=None*)

simulate the behavior of close-loop system (feedback_system) Run self.setup_simulink() before running this this method

Parameters

- **R** – (2D-np.ndarray) state_feedback_matrix
- **input_function** – function

Returns

2D-np.ndarray time series to simulate

x_out: 2D-np.ndarray state of system in time series ,between self.t_sim

y_out: 2D-np.ndarray output of system in time series,

Return type time_array

`apply_observer`(*self*, *L*, *input_function=None*, *logs_file=None*)

`apply_output_feedback`(*self*, *L*, *R*, *input_function=None*, *logs_file=None*)

OpenControl.classiccontrol.observer

Module Contents

Classes

Luenberger	

class OpenControl.classiccontrol.observer.**Luenberger**(*pole*, *system*, *algo='Arckerman'*)

algorithms = ['Ropernecker', 'Arckerman']

_Roppernecker(*self*)

_Arckerman(*self*)

compute(*self*)

OpenControl.classiccontrol.utils

Module Contents

Functions

is_definite_positive_matrix(M)	Determine a maxtrix if
get_coefficient(s)	Finding Coefficient of a term in Polynomial with predefine solution

OpenControl.classiccontrol.utils.**is_definite_positive_matrix**(*M*)

Determine a maxtrix if

Parameters **M** – 2D-ndarray

Returns True if matrix is definite positive

Return type [Boolean]

OpenControl.classiccontrol.utils.**get_coefficient**(*s*)

Finding Coefficient of a term in Polynomial with predefine solution :param s: solution of Polynomial :type s: list

Returns [description]

Return type np.ndarray

OpenControl.visualize

Submodules

OpenControl.visualize.visualize

Module Contents

Classes

Logger	Real-time visualize simulation results, using Tensorboard

Attributes

logX

class OpenControl.visualize.visualize.**Logger**(*log_dir='results'*, *filename_suffix=''*)

Bases: `object`

Real-time visualize simulation results, using Tensorboard

writer

the same as SummaryWriter

Type class

log(*self*, *section*, *signals*, *step*)

Log the signals with the name of section in the step time

Parameters

- **section** (*str*) – name of signals
- **signals** (*array*) – signals
- **step** (*int*) – only *int* type is acceptable. Please convert *float* timestep to *int*

end_log(*self*)

Call this function to end logging

OpenControl.visualize.visualize.**logX**

Package Contents

Classes

Logger	Real-time visualize simulation results, using Tensorboard

class OpenControl.visualize.**Logger**(*log_dir='results'*, *filename_suffix=''*)

Bases: object

Real-time visualize simulation results, using Tensorboard

writer

the same as SummaryWriter

Type class

log(*self*, *section*, *signals*, *step*)

Log the signals with the name of section in the step time

Parameters

- **section** (*str*) – name of signals
- **signals** (*array*) – signals
- **step** (*int*) – only *int* type is acceptable. Please convert *float* timestep to *int*

end_log(*self*)

Call this function to end logging

8.1.2 Submodules

OpenControl._version

Module Contents

OpenControl._version.**__version__ = 0.3**

OpenControl._version.**__author__ = VNOpenAI**

8.1.3 Package Contents

OpenControl.**__version__ = 0.3**

OpenControl.**__author__ = VNOpenAI**

- genindex

Development

You can check out the latest version of the source code with the command:

```
git clone https://github.com/VNOpenAI/OpenControl
```

Installation

Before install OpenControl, ensure that all the dependent packages in the requirements is instelled. Then

    pip install OpenControl

Your contributions are welcome! Simply fork the GitHub repository and send a pull request.

Please see the Developer's Wiki for detailed instructions.

Links

- Issue tracker: https://github.com/VNOpenAI/OpenControl/issues

CHAPTER

NINE

QUICK-START

For quick tutorials and application, please review the Colab Notebook

CHAPTER

TEN

INDICES AND TABLES

- modindex
- search

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X