Reward system design for incorporating control performance

K. Nagaki

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Reward system design for incorporating control performance

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

K. Nagaki

August 12, 2015

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (DCSC)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Reward system design for incorporating control performance

by

K. Nagaki

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: August 12, 2015

Supervisor(s):

dr.ir. G. A. D. Lopes

Reader(s):

dr.ir. M. Mazo Jr.

M. Bharatheesha PhD (BmechE)

S.P. Nageshrao PhD

Abstract

Reinforcement learning (RL) is a machine learning technique whereby the controller learns the control law by optimizing the received cumulative amount of reward. A reward is an instantaneous evaluation of the applied action at the current state, given by reward function. However in theory the reward function is assumed to be given, in practice it is an effort consuming work to design a good reward function. Reward is the only information about the learning task given to the controller and therefore optimizing the cumulative amount of reward corresponds to fulfilling a particular control performance. Designing the reward function to achieve the desired control specification is thus a crucial task to use RL as a reliable controller synthesizing algorithm. The goal of this thesis is to synthesize a method to design proper reward function to achieve the desired control performance.

This thesis focuses on two types of control performance. In the first part, reward function is designed to learn control law fulfilling classical control performance. Hereby an automaton is created to evaluate classical control criteria by mode-dependent reward functions. By modeling the process with an automaton, the control problem is divided into smaller subproblems such that the reward functions are kept simple. In the second part, temporal logic specification is converted into a reward system whereby a petri net is used to model the process suitable for rewarding. To the given temporal formula, reward function is assigned which is a function from the state, i.e. the marking, of the petri net. By putting information about the task into the petri net, reward function becomes simple and structured. Simulation experiments are done for several temporal logic specification.

Table of Contents

List of Figures

Acknowledgements

During the period I was working on this thesis, there were many moments where I lost my self confidence totally and felt helpless. I was deeply depressed and had really desired to escape from everything surrounding me. I am proud to myself that I currently writing this and I would like to thank everyone who had supported me during the thesis. I am really grateful to Gabriel, my supervisor, who had not forsake me even I took very long time to finish this thesis. He gave me a lot of useful suggestions so the time flew during our meetings. Also his enthusiasm had helped me to get motivated to work on. I would also thank Wytske, my counselor, for her assistance to build up my self confidence. Since I do not like to show my weakness to my family or familiars, talking with her had helped me to get out of depression. Finally, I would like to thank my parents for their understandings and supports during the time spent on this thesis.

Delft, University of Technology K. Nagaki
August 12, 2015

Chapter 1

Introduction

Reinforcement Learning (RL) is a semi-supervised learning method frequently used in robotic field. The control law is learned by optimizing the cumulative amount of reward, an instantaneous evaluation of the current state and applied action. The rewards are given to the controller as scalar values according to the specified reward function. This existence of reward function makes reinforcement learning a semi-supervised learning method, since the reward function gives not the exact information how to behave, but only the information either the current situation is good or fault. However, "good" for the practitioner does not have to mean also "good" for the controller.

In the theory of RL, it is usually assumed that the reward function is given. Although in practice, the RL practitioner has to design the reward function by hand since a reward function is problem specific. Reward is the only information about the learning task given to the controller and therefore optimizing the cumulative amount of reward corresponds to achieve a particular control specification. Thus how more complex the desired control performance is, the reward function design becomes also complex. However the reward function forms the core of RL and defines what the controller will learn, there are very less work done about the design of reward function. Even the designing is discussed, it is rather about how to make the controller able to learn and not what the controller will learn. To use RL as a reliable controller synthesizing method, it would be necessary to establish methods for designing reward functions, such that the controller learns the truly desired behavior. In this thesis, more focus is put on designing a reward function that let the controller learn the desired behavior.

In [3] was commented that the reward function should only define what the controller should learn and not how. However, human assumes a lot (or assumes even too much), whereby the definition of what to do becomes difficult. Take as an example walking robot from [4]. At the first instance, the robot was given positive reward when the center of mass of the robot had moved forward. When human walks, most are get used to walk forward and therefore it will become obvious to assume that walking is a way to move forward. However a robot does not know the assumption, rather, it does not

even know what walking is. The robot had learned to alternate stepping forwards and backwards, such that the center of mass moves forward while the robot itself stays at the same position. The other example is riding bike from [5]. The robot had to learn riding bicycle and reward was given for every ridden distance in forward direction as in the case of walking robot. The practitioners had expected that the bicycle would keep riding forward, but as result, the robot had learned to ride in small circles. Although in both cases the learned behavior was not the desired one, the learned policy was not "wrong" since the controller had optimized the way to achieve as much of rewards: the used reward function was not proper one.

Since RL is just a optimization method for collecting rewards, it will be natural to only give a scalar reward 1 if the desired control specification is achieved. Then spending enough time for learning, the controller might learn a totally unexpected policy which works better than a hard-coded controller. Thereby the learned controller would behave according to the given control performance because the reward was only given when the control performance was achieved. Though, this is not practical and usually designers wants that the controller learns the "perfect" manoeuvre which the designers had expected. To learn control laws in reasonable time, the reward function should be information rich enough. Also it is usually preferred to have a way to tune the controller such that the learned behavior gets close to the desired behavior. Thus a reward function should reward the controller in a natural way to avoid undesired behavior. Thereby it must contain enough information to enable the learning and it should have the availability to tune the resulting control law.

1-1 Goal of the thesis

The final goal of this thesis is to synthesize a methodology for designing a proper reward function, where we define a proper reward function as a reward function which:

- leads to a learning controller with desired behavior, i.e. fulfilling the given control performance;
- is structured naturally in the sense that only the truly desired things are rewarded such that learning undesired additional behaviors are avoided;
- supplies enough information to the controller such that the learning is done in reasonable time;
- has the availability to tune the learned control law;
- is able to design in convenient and structured way.

The first two conditions have some in common, since it could depend on the way of giving the control performance. Since there are many "languages" which represents control performance, it is impossible to define corresponding reward functions for all of the control performance languages. Therefore, we will focus on two types of control

performance in this thesis: classical control performance and temporal logic specification.

Obviously, when having a complex control performance, the reward function usually becomes also complex and second and fifth conditions will be hard to fulfill. We will tackle this problem by using discrete/hybrid models to model the process suitable for rewarding. Two ways of modeling are considered. First, by dividing the total problem into subproblems whereby each subproblem corresponds to a control criterion, and second, by translating the control performance into a model. In former case the reward function might be kept simple since it evaluates only one part of the problem. For the latter, we will put information about the problem into the model such that a relatively simple reward function remains as a function of states of the model. Thereby all the complexity of conditions and transitions are hidden in the model.

The fourth condition might not be necessary if the desired behavior is defined by control specification in detail. However, if the control specification only defines a rough behavior, it might be useful if there is a way to tune the control law.

1-2 Outline

The remaining chapters of this thesis is organized as follows. In chapter 2, preliminary knowledge is discussed whereby this chapter exists from three parts. The first part discusses about the algorithm, computational RL framework. In the second and third part, the constituents of the method proposed in later chapters is discussed briefly: models and languages representing the control performance. In chapter 3 a method is proposed to learn controller fulfilling criteria of classical control performance. In chapter 4 temporal logic specifications are converted into a reward system. The reward system exists from a petri net and a reward function as a function of the petri net states and the way of converting is defined. An algorithm for running this reward system is discussed. Finally we conclude the work with conclusion and recommendations (chapter 5).

Chapter 2

Preliminaries

2-1 Reinforcement learning

Reinforcement Learning (RL) is a semi-supervised learning technique whereby the learner learns how to act using the knowledge of its interaction with the environment. It is originally inspired by the animal learning behavior and as like the dog learns tricks by getting cookies by success, the *agent* (or the *controller*) learns the control policy by getting rewarded by its *environment* (or the *process*). The agent tries to maximize the total amount of rewards it receives, which means that the choice of the reward function becomes a crucial task to let learn the truly desired behavior.

2-1-1 Computational reinforcement learning framework

In RL, the *controller* will learn to solve the control problem by interacting with the *process*. A flow of interconnection is represented in figure 2-1. The controller interacts with a process by measuring the states and applying actions, and receives numerical *rewards* according to a given reward function. Everything outside the controller can be seen as the process including the robot's physical body, sensors and actuators as well as the low-level controllers. In RL the reward function is an important aspect which characterizes the learning and the control behavior. The fact that the reward function is placed outside the controller makes RL a semi-supervised learning technique: the controller does not know which actions are "correct".

Markov decision process

A reinforcement learning problem that satisfies the *Markov property* is called a Markov Decision Process (MDP). If the state signal retains all the relevant past information, it is said to satisfy the Markov property: the next state and reward can be determined

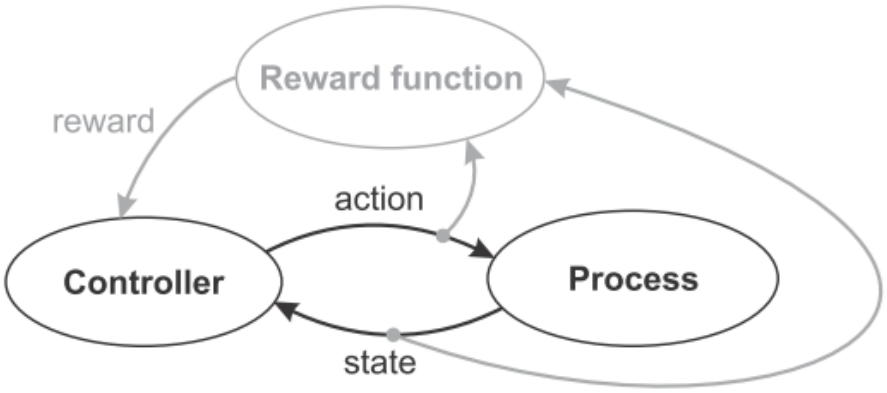


Figure 2-1: The flow of interaction of RL. The elements related to reward are depicted in grey. Figure adopted from [1]

from only the current state and action and do not need the information of the past states and selected actions. A particular deterministic MDP is defined by state and action spaces X and U and by one-step dynamics of the environment which includes the transition function f and the reward function ρ. Applying the action u_k in the state x_k at the discrete time step k, the next state x_{k+1} can be determined using the transition function $f : X \times U \to X$:

$$x_{k+1} = f(x_k, u_k).$$

Similarly the scalar reward signal can be computed according to the reward function $\rho : X \times U \times X \to \mathbb{R}$:

$$r_{k+1} = \rho(x_k, u_k, x_{k+1})$$

where it is assumed that the maximum of the reward function is finite. The *policy* can be interpreted as the control law in control theory: a rule for the controller to choose an action given a particular state. For a simple state feedback control law, where the action only depends on the state, the policy $\pi : X \to U$ is given as

$$u_k = \pi(x_k).$$

When the MDP has a terminal state which is the goal of the task, the task is called 'episodic'. The trajectory from an initial state to a terminal state is called 'trial' or 'episode'. On the other hand a MDP without terminal state is called 'continuous'. There is no specific goal which terminates the task.

Rewards and returns

In reinforcement learning the control specification is realized in terms of rewards. The rewards are scalar numbers and the controller behaves to maximize the received cumulative reward over the run. Reward functions is thus a cost functions and the RL algorithm has to optimize the total cost. The rewards have to be provided in such a way that maximizing rewards corresponds to fulfilling the control task. It is thus a

critical task to design the reward function to achieve the desired control specification. This typically requires experience from the control partitioner to avoid creating reward functions that can lead to unusable controllers even if theoretically they should return a desirable controller.

During the learning process, the controller tries to complete the task by maximizing the cost-function J or the expected *return*. The return R^π expresses the expected total amount of reward by following the policy π, R^π, and it is defined as some specific function of a sequence of the received rewards over the run. Return can be seen as the long-term performance while the reward is the performance in very short term (only one time step). Commonly used return is the *infinite-horizon discounted return* which is given by:

$$J(\pi) = R_k^\pi(x_0) = \sum_{k=0}^{\infty} \gamma^k r_{k+1} \tag{2-1}$$

where $\gamma \in [0, 1]$ is the discount factor. The discount factor is a measure for the importance of the future rewards. For $\gamma = 0$, the controller would act to maximize the immediate reward and ignores the effect in the future by choosing a specific action. This delayed rewards is one of the key feature of the RL problem: how to deal with the rewards that will be received in the future. From a mathematical point of view, discounting ensures that the return will always be finite when the reward is finite. Besides the discounted return there also exist an averaged return. In this work only the discounted return is used and readers interested in averaged return are referred to other works e.g. [2].

Value functions

The value functions are measures for how good the visited state (or state-action pair) is for the controller. Physically, the value functions can be interpreted as memories. The information of the states or state-action pairs are stored in the value functions and the controller makes decisions using this saved knowledge. There are two types of value functions: state-value functions (V-functions) and action-functions (Q-functions). The Q-function $Q^\pi : X \times U \to \mathbb{R}$ is defined as the expected return starting from a given state x, taking a given action a and thereafter following by policy π:

$$\begin{aligned} Q^\pi(x, u) &= \sum_{k=0}^{\infty} \gamma^k r_{k+1} \\ &= \rho(x, u, x') + \gamma Q^\pi(x', u'). \end{aligned} \tag{2-2}$$

Note that this Q-function holds when the discounted return setting 2-1 is used. When using the averaged return setting, the value functions is a bit different. Similarly, the V-function $V^\pi : X \to \mathbb{R}$ is defined as the expected return starting from state x and following policy π thereafter:

$$V^\pi(x) = \sum_{k=0}^{\infty} \gamma^k r_{k+1} = \rho(x, u, x') + \gamma V^\pi(x'). \tag{2-3}$$

The optimal policy π^*, which is a solution of the RL problem, is given as the policy that maximizes the value functions. The optimal value functions Q^* and V^* are characterized by the Bellman optimality equations, which is a necessary condition for promising the optimality and can be represented as:

$$Q^*(x,u) = \rho(x,u,x') + \gamma \max_{u'} Q^*(x',u') \tag{2-4}$$

$$V^*(x,u) = \max_{u} \left[\rho(x,u,x') + \gamma V^*(x')\right] \tag{2-5}$$

Thus the optimal policy computed from Q^* is the policy that satisfies

$$\pi^*(x) = \arg\max_{u} Q^*(x,u) \tag{2-6}$$

and similarly π^* computed from V^* satisfies

$$\pi^*(x) = \arg\max_{u} \left[\rho(x,u) + \gamma V^*(f(x,u))\right]. \tag{2-7}$$

It is more difficult to compute policies from V-functions than from Q-functions. In (2-6) the Q-function already includes the information about the quality of both the states and the transitions while in (2-7) the quality of the chosen action is taken in account explicitly because the V-functions only describes the quality of the visited states. Therefore, Q-functions are preferred to V-functions, although Q-functions are more costly to represent.

Exploration

When the complete and accurate model of the process is available, the controller can choose the actions such that the value functions will be maximized: it can take actions *greedy* with respect to the value function. Such policy that takes always greedy actions is called a *greedy policy*. However, in case by learning from scratch whereby the process is unknown, the initial value-functions are unreliable and a greedy policy does not guarantee optimality. To get reliable estimates of the Q-values, each state-action pairs have to be visited sufficient times. The controller has to *explore* its state and action spaces to obtain knowledge about the process, before it can make right decisions. This exploration can be done by sometimes taking an arbitrary action, which is called *undirected exploration*, or by selecting an action according to the obtained exploration-specific knowledge, called the *directed exploration* [6]. The spaces can be explored efficiently by using directed exploration and may lead to speed up of the learning, although the (near-)optimal policy is independent of the matter of exploration, if the whole region is sufficiently explored. Therefore in this thesis we assume the way of exploration is not influencing the performance specification of the controller.

In the later phase of the learning process, if the controller has sufficient knowledge, the amount of explorative action has to decrease and the controller must take greedy actions to obtain the optimality (*exploitation*). The question arises how long the controller has to explore and when it has to shift to exploitation. By taking long time for exploration,

the controller might be able to obtain more accurate results, but the time needed for learning will increase. If the controller exploits more its knowledge, the learning will speed up but the result can be poor. This trade-off is known as the *exploration-exploitation dilemma* [3],[6],[7]. To handle with the exploration-exploitation problem, time-varying policies can be used e.g. ε-greedy method (which selects an explorative action with probability ε and a greedy action with probability $1-\varepsilon$) and Boltzmann exploration [1].

2-1-2 Solution methods

The RL methods can be divided in three subclasses: *critic-only*, *actor-only* and *actor-critic* methods [2]. The Critic-only methods are methods that derive the optimal policy by first computing the optimal value functions. There is only an update-function for the critic and no explicit function for the policy. The model-based methods Dynamic Programming (DP) and the model-free methods Temporal Difference (TD) learning are the most popular methods. TD learning is discussed shortly. On the other hand, actor-only methods will search the optimal policy directly and they do not use value functions. As example the policy gradient methods is described. Thereafter, Actor-Critic (AC) methods is discussed which has combined the both methods to deal with continuous state and action spaces, which are more suitable for robotics applications.

Critic-only method - Temporal Difference learning

The model-based Dynamic Programming methods are very well developed mathematically, but have the disadvantage that they require a complete knowledge about the environment. Unlike DP, TD methods are model-free and they are implemented in an online, fully step-by-step incremental fashion. Though, TD methods are more complex to analyze. Both methods will search for obtaining the optimal value functions to derive the optimal policy.

The classical RL methods like DP and TD learning are developed assuming discrete state and action spaces. When you extend the methods to continuous state and action spaces, as the most robotics applications have, it becomes impossible to determine exact value functions for each state. Therefore, function approximators are used to deal with this problem [8].

As mentioned earlier, TD learning are implemented in an online, step-by-step incremental fashion: the policy is adapted during the episode according to the estimates of the value functions. The simplest TD method is known as the TD(0) [9] and its update function is defined by:

$$V(x_k) \leftarrow V(x_k) + \alpha_k \underbrace{[r_{k+1} + \gamma V(x_{k+1}) - V(x_k)]}_{\text{TD-error } \delta_k} \tag{2-8}$$

where $\alpha_k \in (0, 1]$ is the learning rate and the term between the brackets is called the TD-error δ_k. The TD-error is the difference between the predicted value $V(x_k)$ and the

updated value $r_{k+1} + \gamma V(x_{k+1})$ and updates the value function into the direction of the new, improved value. The influence of this TD-error on the value function estimate is determined by the learning rate α. Because the update-law is based on temporal estimates, it is said that TD method is a bootstrapping method.

Actor-only method - Policy gradient

Actor-only methods will search the optimal policy directly using optimization related techniques and they do not use value functions. Most of the actor-only methods parameterize the policy and they will optimize the parameters according to the cost function $J(\pi)$, which can be observed as the expectation of the return (2-1) as a function of the policy π:

$$J(\pi) = \sum_{k=0}^{\infty} \gamma^k r_{k+1}. \tag{2-9}$$

In basic policy gradient methods, the policy π is parameterized by a parameter vector ξ [2] and the optimal policy according to the cost-function is obtained by finding the optimal values in the parameter vector space. Because the parameterized policy π_ξ is a function of the ξ, the cost function (2-9) is also a function of the ϑ so the gradient of the cost function with respect to ξ can be written as:

$$\nabla_\xi J = \frac{\partial J}{\partial \pi_\xi} \frac{\partial \pi_\xi}{\partial \xi}. \tag{2-10}$$

By using the ordinary gradient ascent method, the optimality can be obtained by the update rule

$$\xi_{k+1} = \xi_k + \alpha_k \nabla_\xi J_k \tag{2-11}$$

where $\alpha_k > 0$ is the learning rate such that it is sufficient small to have $J(\pi(\xi_{k+1})) > J(\pi(\xi_k))$.

2-1-3 Actor-critic learning

Actor-Critic (AC) learning is developed to combine the advantages of both algorithms discussed in previously [10]. Actor-only methods have strong convergence property, but the variance of the estimated gradient can become quite large [11]. Furthermore the new estimated gradient is independent from the past estimates and the knowledge of the past are not used: there is no "learning". As opposed to it, critic-only methods can use the previously obtained knowledge which are stored in the value functions. The disadvantage is that they lack reliable guarantees on the near-optimality of the resulting policy [2]. By combining the variance in policy gradients can be reduced and the methods still have the convergence properties of actor-only methods.

In AC learning the controller is divided in two parts. Figure 2-2 shows the schematic overview of AC learning. The critic will estimate the value functions and the actor will search directly in the policy parameters space to improve the policy. The updates of the

policy parameters are directed according to the evaluation of the critic. AC methods are useful in robotic applications because they can handle continuous state and action spaces. As in critic-only methods, the critic estimates the value functions using the function approximators and like actor-only methods, the actor gives continuous actions as output.

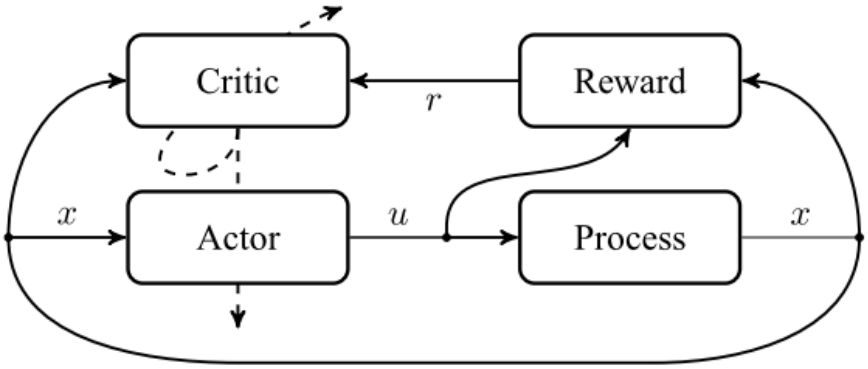


Figure 2-2: The schematic overview of AC learning. The dashed line indicates the updates performed by the critic. Figure adopted from [2].

Standard actor-critic

By having the estimated policy $\hat{\pi}_\xi$ and value function $\hat{V}_\theta$, both parameterized by parameter vectors $\xi \in \mathbb{R}^p$ and $\theta \in \mathbb{R}^q$ respectively, the linear basis function approximation can be given by

$$\hat{\pi}_\xi(x) = \xi^T \phi_a(x) \tag{2-12}$$

$$\hat{V}_\theta(x) = \theta^T \phi_c(x) \tag{2-13}$$

where $\phi_a(x) \in \mathbb{R}^p$ and $\phi_c(x) \in \mathbb{R}^q$ are basis functions of actor and critic respectively. The parameterizing is not necessary linear, but here it is chosen for the sake of simplicity. Because the actor and critic are updated separately, the critic has only to evaluate the current state so there is no need any more to consider the action-value functions and in place the state-value functions can be used, which are cheaper to represent.

The update of critic is done by exploiting the TD-error δ (2-8) in a gradient ascent rule [2]:

$$\theta_{k+1} = \theta_k + \alpha_c \delta_{k+1} \nabla_\theta \hat{V}_{\theta_k}(x_k) \tag{2-14}$$

with $\alpha_c \in (0, 1]$ as the learning rate for the critic parameters. To speed up the learning, eligibility traces can be adopted. With eligibility traces, the value functions of the states that are visited in the past are all updated at once [3], [10]. Using the replacing traces, the eligibility trace $e_k(x) \in \mathbb{R}^q$ is updated according to:

$$e_k(x) = \begin{cases} \lambda\gamma e_{k-1}(x) & \text{if } x \neq x_k \\ 1 & \text{if } x = x_k \end{cases} \tag{2-15}$$

where γ is the discount factor and λ is the trace decay rate. Implementing this in (2-14), the critic update rule becomes

$$e_{k+1} = \lambda\gamma e_k(x) + \nabla_\theta \hat{V}_{\theta_k}(x_k) \tag{2-16}$$
$$\theta_{k+1} = \theta_k + \alpha_c \delta_{k+1} e_{k+1}. \tag{2-17}$$

The actor is updated as like in actor-only method by (2-11), where $\nabla_\xi J_k$ is

$$\nabla_\xi J_k = \delta_{k+1} \Delta u_k \nabla \hat{\pi}_\xi(x_k). \tag{2-18}$$

Δu_k is a zero-mean exploration term perturbing the policy $\hat{\pi}_\xi(x_k)$, which causes difference in actual taken action and the defined action by the policy. When the critic concludes that the Δu_k have good influence to the performance ($\delta > 0$), the policy is updated in the direction of the perturbation Δu_k and otherwise ($\delta < 0$) away from Δu_k.

2-2 Control performance languages

When controllers are designed, the quality of the controller is determined by its control performance. The desired control performance can vary widely and even in the same technologic field, each controller might demanded to satisfy totally different performances. Take as an example the controllers in the robotics field. Manipulators for assembling small components as chips are required to react fast and very accurate, while for manipulators working on the same area as human workers is the safety property most important: fast and robust movements have to be avoided. While in control theory the control performance is an important aspect, in RL-field the performance of the policies are often neglected and only the learning performance are studied. Further in this work, two types of control performance is used to check the quality of the learned controller: classical control performance and simple temporal logic functions.

2-2-1 Reward function

Reward function could be seen as a language specifying the control performance in RL problems. Although in theory of RL a reward function is assumed to be given, in practice the design of a good reward function is a crucial issue. There do not exist structured strategies to design a reward function that leads to the desired behavior and RL practitioners have to design reward functions often by hand. For relatively simple tasks this hand-designing will not be problematic, although it will become more complex and effort consuming when you scale up to more complex tasks. Most natural form of a reward function is as follows:

$$\rho(x, u, x') = \begin{cases} c & \text{if the desired control specification is achieved} \\ 0 & \text{otherwise} \end{cases} \tag{2-19}$$

with $c > 0$ as a constant. This type of reward function is often used in RL problems with discrete state space, whereby the control performance is given in form as "visit a particular state". Since maximizing the cumulative rewards over time should correspond to fulfilling the control performance, this reward function will result always in a controller fulfilling the control specification theoretically. However for poorly defined control specification, the control law could be undesired. This reward function will work worse for continuous state space problems. The probability of achieving the control specification is too low which causes for a bad update rate of the value functions and thus learning: the information contained by the reward function is too little. Therefore for continuous problems, the absolute or quadratic error are usually used to penalize the non-desired states

$$\rho(x, u, x') = -C|x - x_{\text{des}}| \tag{2-20}$$

$$\rho(x, u, x') = -(x - x_{\text{des}})^T Q(x - x_{\text{des}}) \tag{2-21}$$

where $C > 0$ and $Q > 0$ are cost matrices.

Reward Shaping

To solve complex problems the reward functions should be information rich enough to be able to update the values of the states with appropriate speed. Editing a sparse reward function into a dense one is called *reward shaping*. The idea of shaping is to solve an edited (easier) problem, whereby it is supposed that the solution of the changed problem is also optimal in the original problem. Thus by using reward shaping, an MDP is solved using dense reward function ρ' instead of the original reward function ρ, which contains more information and hopefully leads to the same optimal policy [12], [5], [13]. Ng et al. had found that special way of editing reward function will keep the optimality of the original reward function [14]. However, heuristically shaped reward function has usually no guarantee that it remains optimality of the original reward function.

Inverse Reinforcement Learning

As the name already implies, the problem definition of Inverse Reinforcement Learning (IRL) is the opposite as that from the ordinary RL: while in ordinary RL the goal is to obtain a policy that is optimal in given environment with a given reward function, in IRL the goal is to find a reward function that makes the given policy (or a sample trajectory) optimal. When having state trajectories available with desired characteristics, It might be able to run an IRL algorithm to get a reward function that will results in a policy with desired behavior [15], [16].

2-2-2 Classical control performance

The classical control performance are used to measure the control performance by analyzing the transient step response of the system, especially when using the Proportional-

Integral-Derivative (PID) controllers. Usually four aspects of the transient response are evaluated:

- rise time t_r is the required time to go from 10% to 90% of the reference value;
- overshoot M_p is the amount that the system overshoots its reference value, expressed in ratio;
- settling time t_s is the required time that the state settles within a certain range of the reference value;
- steady-state error e_{ss} is the residual amount of deviation of the state from the reference value.

Figure 2-3 shows the four aspects graphically. Mention that this measure of performance does not take the effect of noise in account. Therefore it might be difficult for the RL controller to obtain a good performance since in RL an action noise is compulsory for learning.

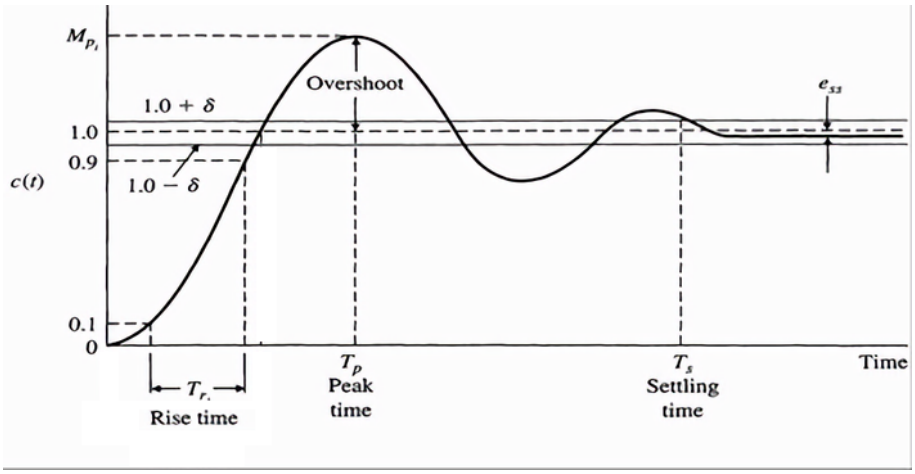


Figure 2-3: Classical control performances in the time domain step response.

2-2-3 Metric Interval Temporal Logic

Temporal Logic (TL) is a convenient formalism for specifying and verifying properties of reactive systems that are often used in computer scientific field. The main purpose is to represent the performance as a formula using special syntaxes. In model checking, there are techniques to check whether the given model fulfills the given TL formula. Metric Interval Temporal Logic (MITL) is a subset of TL, whereby quantitative timing constraints are added on the operators. Further in this thesis, MITL formulas are used with an aim to represent the control performance and not for model checking.

The temporal formula is built up over a finite set of atomic propositions AP using the following grammar:

$$\varphi ::= \top \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid a \mid \Diamond_I \varphi \mid \Box_I \varphi \mid \varphi_1 \mathcal{U}_I \varphi_2 \tag{2-22}$$

where the syntaxes are listed in table 2-1 and subscript $I \subseteq \mathbb{R}$ is a time interval. This interval makes MITL different from ordinary TL and defines the quantitative timing when the continuing temporal formula has to be achieved. Let have the the temporal formula $\Diamond_{\{5,10\}} a$ as an example. The temporal formula is satisfied when a becomes true within time from 5 to 10 seconds later. When the subscript I is not given, it means that the timing is not for importance and can be interpret as $I = \{0, \inf\}$.

Table 2-1: Syntaxes

Textual	Symbolic	Explanation
	a	Atomic proposition
	$\neg\varphi$	Negation
	$\varphi_1 \wedge \varphi_2$	Conjunction
F φ	$\Diamond\varphi$	In the **F**uture. Eventually φ becomes true.
G φ	$\Box\varphi$	**G**lobally. φ has to hold always.
φ_1 **U** φ_2	$\varphi_1 \mathcal{U} \varphi_2$	**U**ntil. φ_1 has to hold until φ_2 becomes true at a future position.

A traditional stabilization problem could be represented as temporal formula

$$\varphi_1 = \Diamond\Box\phi \tag{2-23}$$

where $\phi \in AP$ is true if $x \in x_{\text{des}}$ with $x_{\text{des}} \subset X$ holds. The control performance reads as "Eventually Globally ϕ" and the controller fulfills the performance if this temporal formula is true. Thus in the future, always $x \in x_{\text{des}}$ must hold and therefore this temporal formula represents a stabilization problem. Also the common obstacle avoiding control performance could be expressed conveniently as

$$\varphi_2 = \Box\neg\psi \tag{2-24}$$

where $\psi \in AP$ is true if $x \in x_{\text{obs}}$ with $x_{\text{obs}} \subset X$ holds. φ_1 and φ_2 forms together the goal reaching obstacle avoidance problem

$$\varphi = \varphi_1 \wedge \varphi_2 = \Diamond\Box\phi \wedge \Box\neg\psi. \tag{2-25}$$

2-3 Discrete/Hybrid models

To represent varying conditions of a process, discrete modeling is an effective way. Even when the state space is continuous, the set of states could be labeled as discrete states or conditions such that it can be modeled. When a process contains both discrete and continuous states, it is called a hybrid system and hybrid models are used to solve control tasks on that kind of systems.

In this section (hybrid) automata and petri net are introduced briefly.

2-3-1 Automata

A finite automaton or a finite state machine is a mathematical model of computation broadly used in several fields as computer science, communication systems and also in control theory. It is an abstract machine that computes the states transitions in response to the given inputs and can be used to model event-driven systems. On/off switch is a very simple example of a finite state machine. It has two states 'on' and 'off' and when there is an input (e.g. a button is pushed) the state will change. A finite state machine is defined by the triple $\Sigma(Q, U, F)$ with a finite set of states Q, a finite set of input U and a state transition function $F : I \times Q \to Q$. The dynamics of an automaton could conveniently be represented by a directed graph.

Hybrid automaton

Automatons can model digital systems with discrete states, however many digital systems interacts with physical processes which live in continuous state space. To deal with such kind of hybrid systems, the ordinary automaton is augmented to the hybrid automaton. The discrete part of the system is modeled by the traditional automaton: there are still discrete states (also called the modes) and driven by an event the state will change from one to another. While in traditional automata theory this event is an input from the input set, in hybrid automaton the event depends on the continuous dynamics. Each mode is associated to an invariant, which describes the conditions that the continuous states has to satisfy at that mode. The evolution of the continuous states are typically described by differential equations and whenever the states violate the invariant conditions, a mode transition takes place. To which mode it will change will depend on the guard which specifies the subset of continuous state space where a certain transition is enabled. When transition takes place the continuous states will go through the reset map which specifies the relation between the new and previous continuous state.

A hybrid automaton H is an 8-tuple $H = (D, X, f, \text{Init}, \text{Inv}, E, G, R)$ with

- $D = \{q_1, \ldots, q_N\}$ is a finite set of discrete states or modes;
- $X = \mathbb{R}^n$ is a set of continuous states;
- $f : Q \times X \to X$ is a vector field or the function describing the evolution of the states;
- $\text{Init} \subseteq Q \times X$ is a set of initial states;
- $\text{Inv} : Q \to \mathcal{P}(X)$ describes the invariants;
- $E \subseteq Q \times Q$ is a set of edges;
- $G : E \to \mathcal{P}(X)$ is a guard condition;
- $R : E \to \mathcal{P}(X \times X)$ is a reset map.

Here $P(X)$ is the power set of X.

A simple example is a thermostat-heater system. The continuous state is the temperature which evolves according to the thermodynamic law. Depending on the current temperature, the mode of the heater changes between 'on' and 'off': the mode will become 'on' when the temperature is too low and 'off' when too high.

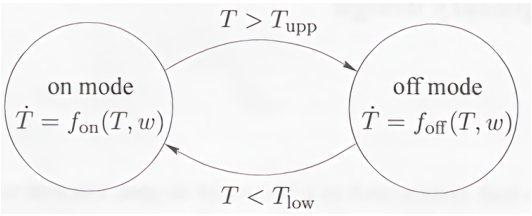


Figure 2-4: Hybrid automaton for a thermostat-heater system.

2-3-2 Petri net

An alternative to model the discrete events of a process could be a Petri net (PN). A petri net graph consists of places, transitions and arcs. The transitions are expressed by bars and it defines whether events occur. The transition bars are connected by arcs with the places, expressed by arrows and circles respectively, and each place implies a condition. The places may contain discrete number of marks called tokens which visualize the active conditions. The distribution of tokens over the whole places is called marking and represent the state of the system. Formally a petri net graph is a bipartite graph (P, T, A, M_0) with $P = \{p_1, p_2, \ldots, p_n\}$ and $T = \{t_1, t_2, \ldots, t_m\}$ are set of places and transitions respectively, $A \subset (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions and vice versa and M_0 is the initial marking. Figure 2-5 shows an example of a petri net graph.

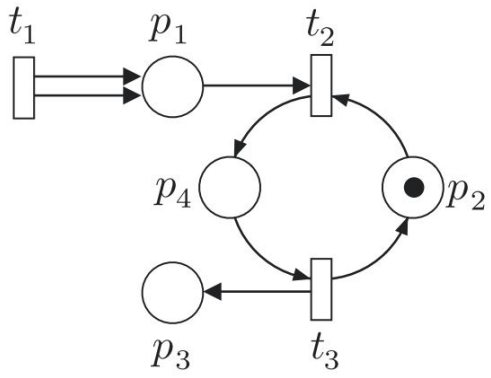


Figure 2-5: An example of a petri net.

The state evolutions are expressed by the movement of the tokens from places to places. The connection of the places and transitions shows relations from conditions and occurring events. If an arc runs from a place to a transition, that place is called the input place of the transition and must hold i.e. the input place must contain a token to enable

the transition. The transition is enabled when all the input places contain (at least one) token(s) and then the transition may fire. The firing represents an occurrence of the event e.g. an action taken. After firing, from each input place a token is consumed and a token is generated at all output places, places where arcs run from the transition. Those output places might again be the input places from other transitions.

The state of the petri net system is expressed by the marking matrix $M \in \mathbb{N}_0^{1\times n}$ where n is the number of places. The evolution of the marking matrix can be determined by:

$$M' = TD + M \tag{2-26}$$

where $T \in \{0,1\}^{1\times m}$ is the transition matrix with m as number of transitions and $D \in \mathbb{N}_0^{m\times n}$ is the composite change matrix. The transition matrix represents which transition fires. The composite change matrix defines the relations of the places and the transitions and is calculated by

$$D = D^+ - D^- \tag{2-27}$$

where D^+ and D^- are $m \times n$ matrixes filled in with one if place p_j is the output/input place from transition t_i and zeros elsewhere.

2-4 Summary

In this chapter, the basics of RL, some modeling techniques and control performances are discussed briefly. The advantage of reinforcement learning is that the designer does not need to give the robot a data which strictly defines the task. This is also the reason that RL is called a semi-supervised learning method. However, to obtain a truly "optimal" controller, it is necessary to design a reward function carefully. A possible way how to attack this complex designing is discussed further in this work.

Chapter 3

Reinforcement Learning for classical control performances

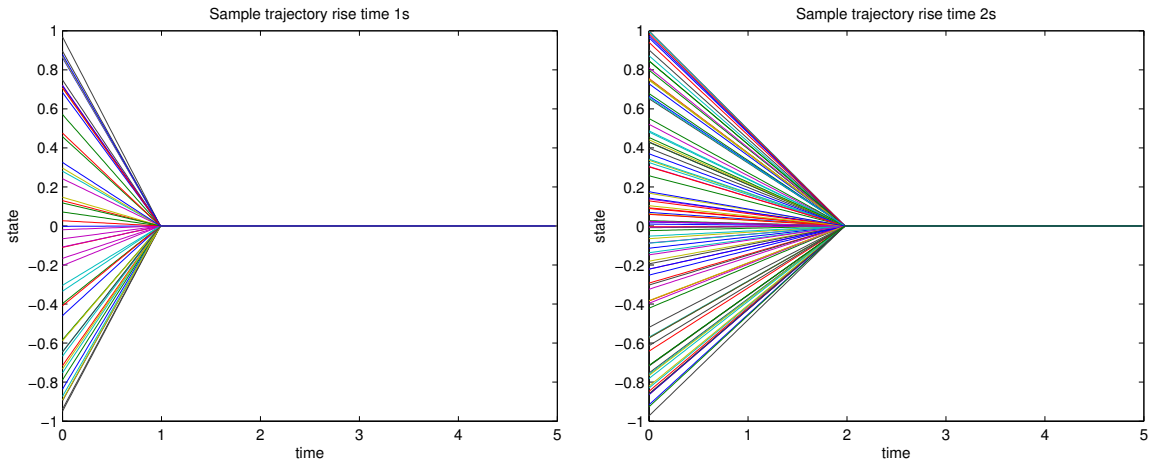
Proportional-Integral-Derivative (PID) controller is by far the most used controller in the world. This is not only due to its simple structure but also due to its possibility to define the control performance conveniently. However, PID is a linear controller and tuning of its parameters maybe challenging if the underlying system is highly nonlinear. RL on the other hand, it is very suitable to designing nonlinear controllers for highly nonlinear systems, due to its trial and error approach. Though to make RL a well-behaved learning controller synthesizing method, a structured reward design method for incorporating desired control performance is essential. Thereby a question could be "Is it possible to apply the classical PID controller specification to a RL controller?". In this chapter RL incorporating classical control performance, especially the overshoot, is considered and discussed.

3-1 Difficulties by reward function design for classical control performance

A requirement for a RL problem is that the system should be described by a MDP. Consider for example rise time performance. Will it be possible to incorporate time related performance in the reward function $\rho(x, u, x')$ where $t \notin x$? To consider this, first a mass-damper problem is discussed using inverse reinforcement learning.

Apprenticeship learning [16] is used where a reward function is obtained in iterative way by comparing the given and the found *feature expectations*. Assuming the reward function linearly parameterized in the features, the value function can be rewritten as a product of parameter vector and the feature expectations. Using a linear optimization technique this feature expectations is determined iteratively such that at least one of the policy learned with found feature expectations will result close to the given trajectories.

Example: apprenticeship learning. *Having a simple mass-damper system with states* $x = \begin{bmatrix} q & \dot{q} \end{bmatrix}^T$ *with* $q \in [-1, 1]$ *and* $\dot{q} \in [-5, 5]$ *are position and velocity relatively, the problem is to stabilize the mass position at* $q = 0$ *from its initial position, randomly chosen from the range* $[-1, 1]$*. Given two sample trajectory sets A and B with different time used to reaching the goal (see Figure 3-1), the obtained reward functions using apprenticeship learning are discussed. Both trajectory sets exist from 50 samples. The model of the system can be found in the appendix (B.*



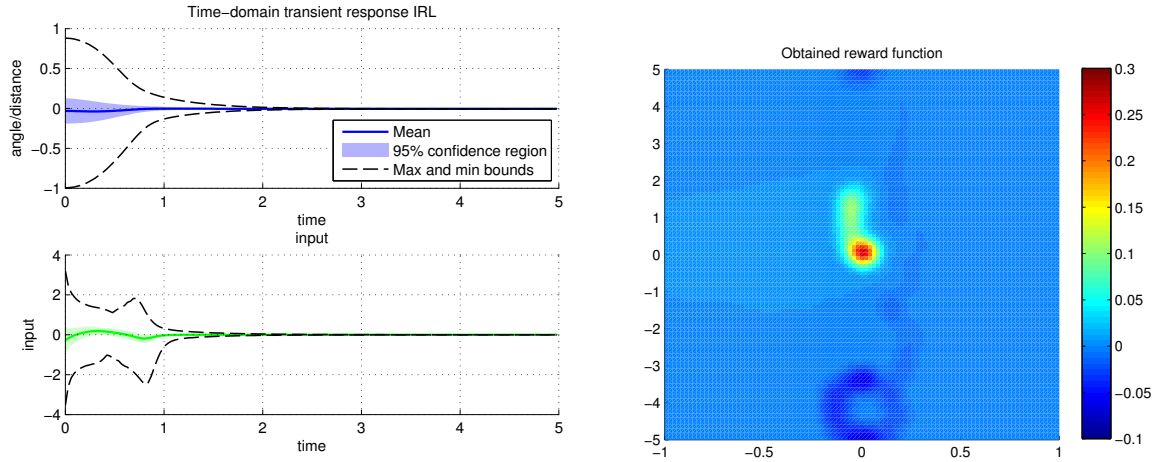
(a) Case A: the reference value is reached in 1 second. (b) Case B: the reference value is reached in 2 seconds.

Figure 3-1: Two set of sample trajectories.

Figure 3-2 and 3-3 show the obtained reward functions with the transient simulations which were closest to the original ones. For both cases the agent succeed to stabilize near 0. However, the time taken to go to reference value is for both cases about 1 second, while the sample trajectories of case B took 2 seconds to go.

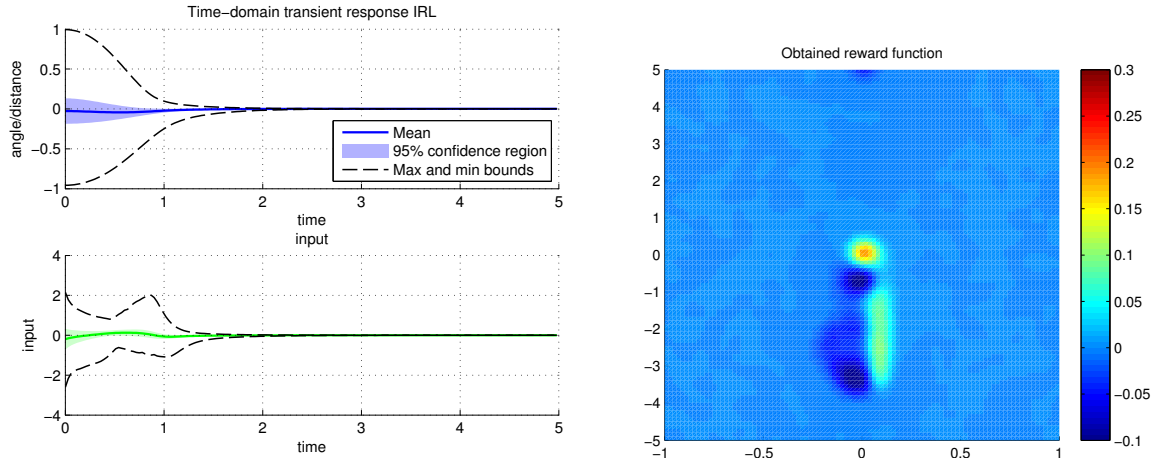
When the reward functions are compared, they look very similar: both reward functions have a peak at $(0, 0)$ and reward is near 0 elsewhere. Differences are the existing non-zero reward areas and the height of the peak. Seemingly the non-zero rewards around the peak have no or only very less influence to the policy, since for both cases, the policy behaves very similar and there are no obvious difference between them that might be caused by non-zero rewards (plot (a) from Figures 3-2 and 3-3). Also the height of the peak has less influence. Since the structure is the same, the amount of the reward (in association with the choice of learning parameters) has only influence on the learning behavior: the value function of the state $(0, 0)$ will updated with larger steps in case A than in B, however it will have no influence on the time properties of the policy. From this, we will conclude that time should also show up in the reward structure to let learn time related performances. Thus the system has to be augmented by time when incorporating rise time.

In a maze problem, time property is incorporated in reward functions by giving negative rewards each time step, to reach the goal as fast as possible. This is done under two assumptions. First, the initial value of all the states are set to zero, and second, the



(a) (b)

Figure 3-2: (a) Transient plot of simulation run without noise, repeated for 50 times. The policy is learned using the reward function obtained with IRL, is used as sample trajectories. (b) Obtained reward function plot from top-view.



(a) (b)

Figure 3-3: (a) Transient plot of simulation run without noise, repeated for 50 times. The policy is learned using the reward function obtained with IRL, is used as as sample trajectories. (b) Obtained reward function plot from top-view.

problem terminates when the goal is reached. The agent observes the negative reward as a "penalty" only when it had expected higher reward at that state. Therefore, a negative reward is given at every time step, such that the agent receives the largest cumulative amount of rewards when it reaches the goal with as less as possible steps. However for a problem which continues even after the goal has reached, as like a stabilization problem, the agent usually remains receiving rewards as long as it stays at the goal. Here, the best policy is to reach the goal as fast as possible and stay there until the end time. It does not make sense anymore to give negative rewards each time step since the

best policy already contains the information about the time optimality. When negative rewards are given for stabilization problems, it is either due to exploration or due to reward shaping to make learning easier for the agent.

The rise time is defined by the required time to go from 10% to 90% of the final value, so assuming the final value is always the same, the information of the initial value has also to be known when the rise time is evaluated. Since the initial state is chosen randomly during the learning, the initial state should also be contained in the augmented state. The same holds when evaluating overshoots. The maximum allowed peak value is derived by the equation

$$q_{\text{peak}} = q_{\text{desired}} + M_p(q_{\text{desired}} - q_{\text{initial}}) \tag{3-1}$$

where M_p is a given ratio of maximum allowed overshoot. Thus keeping the Markovian property by having the augmented system state $x_{\text{aug}} = \begin{bmatrix} q & \dot{q} & q_0 & t \end{bmatrix}^T$, it should be able to tackle stabilization problems with required rise time and overshoot. However, to begin with a simple problem, we will first consider stabilization problem with a given maximum allowed overshoot. The time does not need to be a state anymore and the augmented state becomes $x_{\text{aug}} = \begin{bmatrix} q & \dot{q} & q_0 \end{bmatrix}^T$.

The timing to evaluate the performance will also be important. When using PID controller, the performance is evaluated after the run. In RL, giving reward only at the end of the run is not a proper way of rewarding. Because of the definition of TD-error (2-8), only the last state will be evaluated and the performance during the run will be neglected. It is somehow comparable to teaching manner to the dogs. When a dog does something wrong, it has to be taught immediately such that it understands it was behaving bad. Even if you scold the dog an hour later, it cannot understand why it was scolded and it will not learn to avoid bad behaves. Thus in RL the performance has to qualified real time, but then the question is how to design the reward function. The most obvious way is to design a piecewise reward function depending on the living subspace. However, designing piecewise reward function becomes complex soon and it will be very problem specific. It might even be that a time dependent piecewise reward function is needed.

Instead of making reward function complex, the system model can be changed such that the reward function itself remains relatively simple. We suggest to use automaton to make timing of performance evaluation easier. There are several advantages to use automaton. First, we can define by ourself the "checkpoints". Since we can define the transition conditions by ourself, we are able to define when to evaluate a specific control criteria. Also the order of evalution can be chosen. Second, by having a separate reward function for each mode, we can construct a complex reward system at the end, using simple reward functions as building blocks. Third, we can fix the rewarding structure. We can reuse the automaton for other stabilization problems only by changing its transition conditions and mode-wise reward function. In the next section we will discuss our proposed method formally.

3-2 Hybrid Reinforcement Learning

We introduce a new framework for Hybrid Reinforcement Learning (HyRL), whereby the term "hybrid" refers to the combination of continuous and discrete states in a system. We will have a hybrid automaton $H = (D, X, f, \text{Init}, \text{Inv}, E, T, \varrho)$ where D is finite set of discrete states (modes), X is a set of continuous states, f is a transition function of continuous states, Init is a set of initial states, Inv describes conditions that continuous state has to satisfy in given mode, E is a set of edges, T is a set of transition relation and ϱ is a set of reward functions (see Figure 3-4). The guard and reset map conditions are combined into transition relations. We will go through the explanation of HyRL by considering the following example:

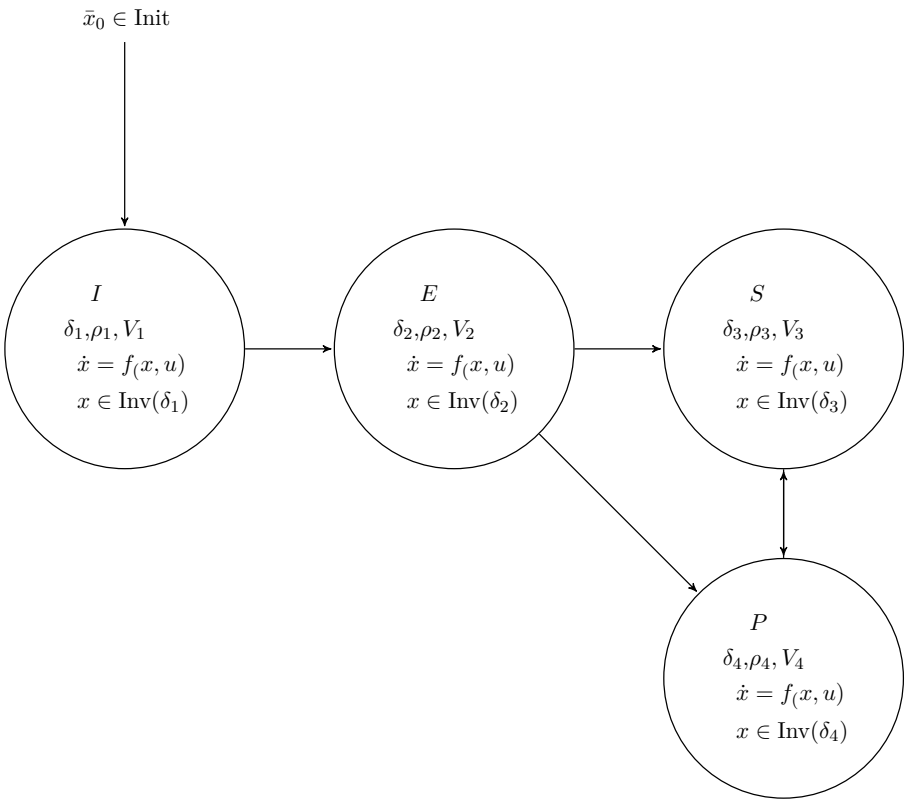


Figure 3-4: Hybrid automaton describing the condition of the system.

Example 2: Mass damper stabilization without overshoot. *The problem is to stabilize the mass-damper system of example 1 (for model see A-1) at $q = 0$ without any overshoot ($M_p = 0$) by using HyRL.*

To tackle the stabilization problem, an automaton with the next modes is created:

- 'idle' mode (I). The initial mode and the agent begins to go towards the desired value;
- 'evaluation' mode (E). According to the given transition condition, the automaton

jumps from 'idle' to 'evaluation' mode. Particular control criteria is evaluated real time;

- 'penalizing' mode (P). When the control criteria is violated, the automaton jumps to this 'penalizing' mode. Only negative reward is given and the controller should learn to avoid this mode;
- 'stabilizing' mode (S). If the agent fulfills the desired control performance and reaches the desired value, the automaton escapes the 'evaluation' mode and ends here. In this mode the controller should stay around the desired value. When this condition is violated, the automaton jumps to the 'penalizing' mode.

Depending on the number of desired control performance, the number of 'evaluation' mode will change. For each classical control criterion, an 'evaluation' mode is created such that the automaton jumps to the relevant evaluating mode at proper timing.

Example 2 (continuation). Since we only care the overshoot performance, an hybrid automaton with four nodes is associated (Figure 3-5). The continuous states will evolve according to the equation of motion and the invariant sets are noted in the figure. The automaton jumps to 'evaluation' mode when it reaches 95% of the desired value and 'evaluation'-'penalizing' transition is done if the mass position exceeds the desired value. From the evaluation mode, the automaton jumps to 'stabilizing' if the velocity is zero or downwards. The mode remains 'stabilizing' as long as the position is between 90% and 100% of the desired value. Since the initial state is random over the range of [-1,1], we have to care about over- and undershoot. The given invariant is for position and only valid for overshoots (not undershoots where $q_0 > 0$, for complete transition condition see appendix A-1.

As mentioned in the previous section, there is need to augment the system states. We introduce a memory state z which stores the values needed for computation of the performance thresholds. Thereby, also the mode of the automaton is added such that the augmented state $\bar{x}$ becomes finally:

$$\bar{x}_{k+1} = \begin{bmatrix} x_{k+1} \\ z_{k+1} \\ \delta_{k+1} \end{bmatrix} = \begin{bmatrix} f(x_k, u_k) \\ m(x_k, z_k) \\ \Gamma(x_k, z_k, \delta_k) \end{bmatrix} \tag{3-2}$$

where x is the original system states, δ is the mode of the automaton. Using the feedback control law $u_k = h(\bar{x}_k)$, the closed loop system becomes

$$\bar{x}_{k+1} = \bar{f}(\bar{x}_k) = \begin{bmatrix} f(x_k, \pi(\bar{x}_k)) \\ m(x_k, z_k) \\ \Gamma(x_k, z_k, \delta_k) \end{bmatrix} \in \bar{X} \times D. \tag{3-3}$$

Since the memory state z is added to the system, the whole system remains Markovian. Therefore the reward function in each mode is a function of the augmented state $\bar{x}_k$, input u_k and next state $\bar{x}_{k+1}$:

$$r_{k+1} = \rho(\bar{x}_k, \pi(\bar{x}_k), \bar{x}_{k+1}) \in \varrho \tag{3-4}$$

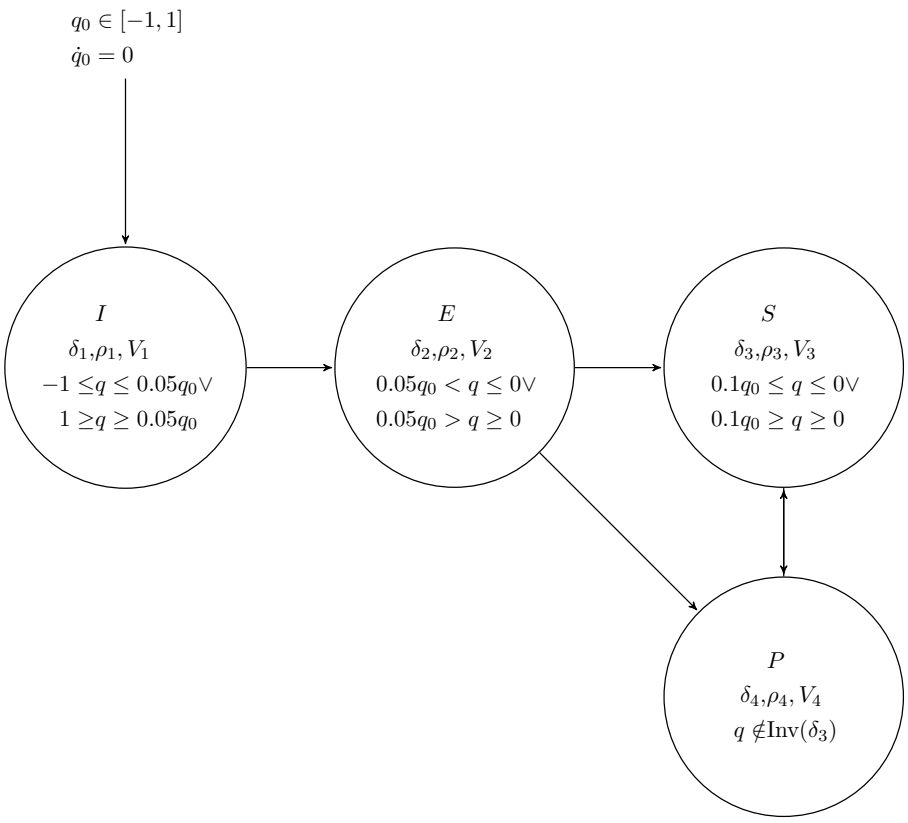


Figure 3-5: Hybrid automaton for over-/undershoot performance $M_p = 0$.

where the reward function $\rho(\cdot)$ might be different for each mode d. Finally, we have a hybrid automaton $H = (D, \bar{X}, \bar{f}, \mathrm{Init}, \mathrm{Inv}, E, G, R, \varrho)$. The control law will be learned by running RL on this hybrid automaton with some adjustments in computation of the approximators, which is discussed in the next subsection.

Example 2 (continuation). To compute the overshoot threshold, we need to know the initial value. The augmented system state becomes

$$\bar{x}_k = \begin{bmatrix} q_k \\ \dot{q}_k \\ q_{0,k} \\ \delta_k \end{bmatrix} \tag{3-5}$$

where $D = \{\delta_1, \delta_2, \delta_3, \delta_4\} = \{I,E,S,P\}$.

3-2-1 Approximators

As like in the ordinary RL problem, the main goal is to find an optimal policy $\pi^*(\bar{x})$ and therefore the computation of the value functions is necessary. Since there are multiple reward functions (one for each mode), each mode also needs an separate value function for the specific reward function. There are multiple ways to estimate the value

functions, e.g. approximators linear in the basis function, neural networks etc. Here, a basis function approximation linear in the basis function ϕ is defined. The value function for each discrete state can be shown as:

$$\hat{V}_i(\bar{x}) = \theta_i^T \phi_{c,i}(\bar{x}) \tag{3-6}$$

where i represents the discrete state and θ is the parameter that has to be learned. Besides the mode-dependent value function, we will introduce a general value function $\hat{V}(\bar{x})_{\mathrm{g}}$ which contains information that holds for each discrete state. The general value function is introduced to make the value function smooth and to prevent large gaps by transitions between two modes. This general value function forms the base of the total value function. Each mode-dependent value function characterizes the total value function according to the associated reward function. The general value function will be updated independent of the mode, while the mode-dependent value functions are only updated when the mode is active.

The total value function is the sum of the general part and the mode dependent part

$$\hat{V}(\bar{x}) = \hat{V}(\bar{x})_{\mathrm{g}} + \hat{V}(\bar{x})_i = \theta_{\mathrm{g}}^T \phi_{c,\mathrm{g}}(\bar{x}) + \theta_{c,i}^T \phi_i(\bar{x}) \tag{3-7}$$

and by saving all the parameters and basis functions in large vectors, approximated value function becomes

$$\hat{V}(\bar{x}) = \begin{bmatrix} \eta_i(\delta)\theta_i^T & \cdots & \eta_j(\delta)\theta_j^T & \theta_{\mathrm{g}}^T \end{bmatrix} \begin{bmatrix} \phi_{c,i}(\bar{x}) \\ \vdots \\ \phi_{c,j}(\bar{x}) \\ \phi_{c,\mathrm{g}}(\bar{x}) \end{bmatrix} \tag{3-8}$$

where η is the switching function

$$\eta_i(\delta) = \begin{cases} 1 & \text{if } \delta = i \\ 0 & \text{elsewhere.} \end{cases} \tag{3-9}$$

In the same way, we will define approximated policy as

$$\hat{\pi}(\bar{x}) = \begin{bmatrix} \eta_i(\delta)\xi_i^T & \cdots & \eta_j(\delta)\xi_j^T \end{bmatrix} \begin{bmatrix} \phi_{a,i}(\bar{x}) \\ \vdots \\ \phi_{a,j}(\bar{x}) \end{bmatrix}. \tag{3-10}$$

Mention here that the approximated policy does not have global term: for each mode the controller could learn a different policy.

3-2-2 Mode-wise reward function

The reward function in each mode could be seen as building blocks to design a complex reward function for the total problem. However, there remains still the question how to choose this mode-wise reward functions. By choosing complex reward functions for

each mode, the final reward structure may become too difficult even for the designer to understand which part of the task is rewarded and which not. Therefore in this work we will use two types of simple and intuitive reward functions as building blocks. The first one is the traditional discrete reward function, which is also mentioned in previous chapter (2-19)

$$r = \rho(x, u, x') = \begin{cases} c & x \in S \\ 0 & \text{otherwise} \end{cases} \tag{3-11}$$

with $S \subseteq X$. However this reward function works worse for a very rare condition (e.g. visit a single state while having a continuous state space), by choosing the condition and the value of c properly, it could give rough information to the agent what is desired. For example with $c < 0$ and $S = X$, the agent receives always a penalty and it tries to avoid this situation somehow. However this example works not for an ordinary RL problem, since we have multiple modes, giving only a penalty in one mode will let avoid the controller the transition to this mode.

The second reward function is

$$r = \rho(x, u, x') = -C|x - x_{\text{des}}| \tag{3-12}$$

where $C > 0$. This reward function will force the agent to stay close at the desired state. It is also used frequently in continuous state space RL. In this work, only the absolute error in position is penalized.

Designing a proper reward function is a complex work. By associating the problem with hybrid automaton, the complex part of reward function, as like defining the timing of rewarding, is done in the hybrid automaton design. The remained simple part of what to reward can be tuned mode-wisely with simple reward functions.

Example 2 (continuation). Since we have four modes I, E, S and P, we also have four reward functions, see table 3-1. In I, the agent is penalized when it is getting away from the desired value. In E and S, the absolute error between the current and desired value is penalized. Furthermore in S, it is rewarded when it stays in stabilizing mode: this can be seen as reward corresponding to the velocity. In P the agent is always penalized with a big value, so the best policy is avoiding P while staying in S.

Table 3-1: Mode-wise reward functions for mass-damper stabilization with overshoot performance $M_p = 0$.

Mode	Reward function
I	$\rho_1 = \begin{cases} 0 & \text{if } \dot{q} > 0 \mid q_0 < q_{\text{des}} \text{ (overshoot)} \\ 0 & \text{if } \dot{q} < 0 \mid q_0 > q_{\text{des}} \text{ (undershoot)} \\ -3 & \text{else} \end{cases}$
E	$\rho_2 = -3\lvert q_{\text{des}} - q \rvert$
S	$\rho_3 = -3\lvert q_{\text{des}} - q \rvert + \begin{cases} 3 & \text{if } \delta = \delta' = \text{S} \\ 0 & \text{else} \end{cases}$
P	$\rho_4 = -9$

3-3 Simulation experiments

After the modeling of the augmented system is complete, RL can be run. Continuously, we will work on further with the mass-damper system of the example and here the simulation results are discussed. Further, mass-damper system with given rise time performance is experimented.

3-3-1 Simulation: Mass-damper without overshoot

The used simulation parameters in the simulation experiments are listed in table 3-2. The initial state was randomly chosen from $[-1, 1]$.

Table 3-2: Simulation parameters for simulation experiments HyRL.

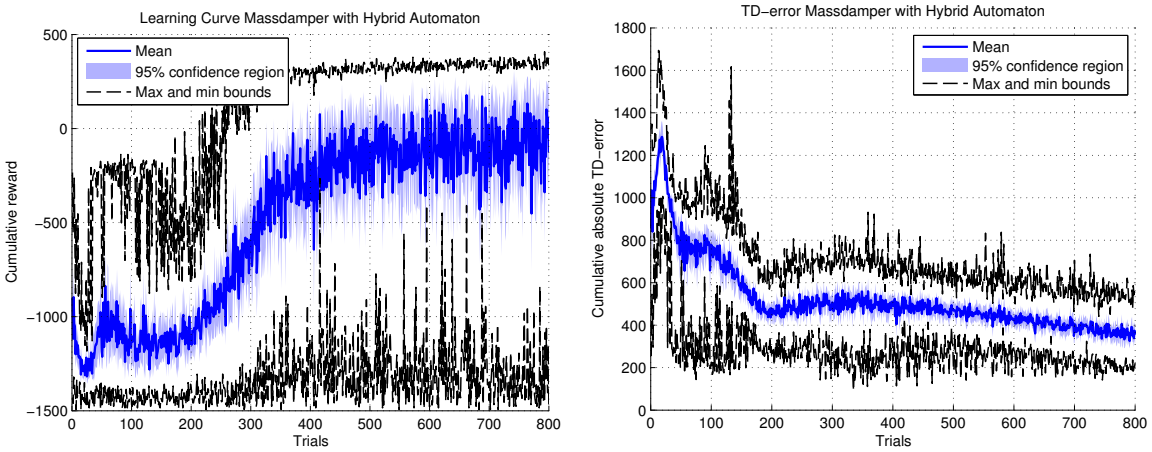
Simulation parameters	Symbol	Value	Units
Number of trials	-	800	-
Trial duration	T_t	5	s
Sample time	T_s	0.03	s
Discount factor	γ	0.98	-
Eligibility trace decay	λ	0.65	-
Learning rate critic	α_c	0.3	-
Learning rate actor	α_a	0.08	-
Number of radial basis functions	-	$10 \times 10 \times 5$	-

Figure 3-6 shows the learning curve plots of the simulation. The learning experiment is repeated for twenty times. Both the learning curve and the TD-error are converged, although the learning curve is very noisy. This noisy learning behavior is caused by the combination of random initial position and the defined invariant set in stabilizing mode, which is explained during discussing the time domain response. For a fixed initial value of -1, the learning curve converged very nicely.

Time domain response

However the convergence was not perfect, the controller had learned to avoid overshoot, i.e. the controller had learned to avoid visiting the penalizing mode. Figure 3-7 shows the transient response of position and input, simulated using the learned policy. The background color represents the current mode of the hybrid automaton. The penalizing mode is given in red color, however it is not presence since it was not visited in this run. In the begin of idle state, the mass is given a hard push by the controller such that it moves towards the goal. After it is in the neighbor of the desired position, the automaton jumps to the evaluating and starts to give an input against the moving mass, such that it will not exceed zero.

Figures of 3-8 represents the responses for all the initial states. It can be seen that both over- and undershoot performance of $M_p = 0$ are achieved. The small peaks of



(a) The averaged learning curve of the experiment repeated for 20 times.

(b) The averaged cumulative TD-error plot.

Figure 3-6: Learning behavior by learning overshoot performance for mass-damper system.

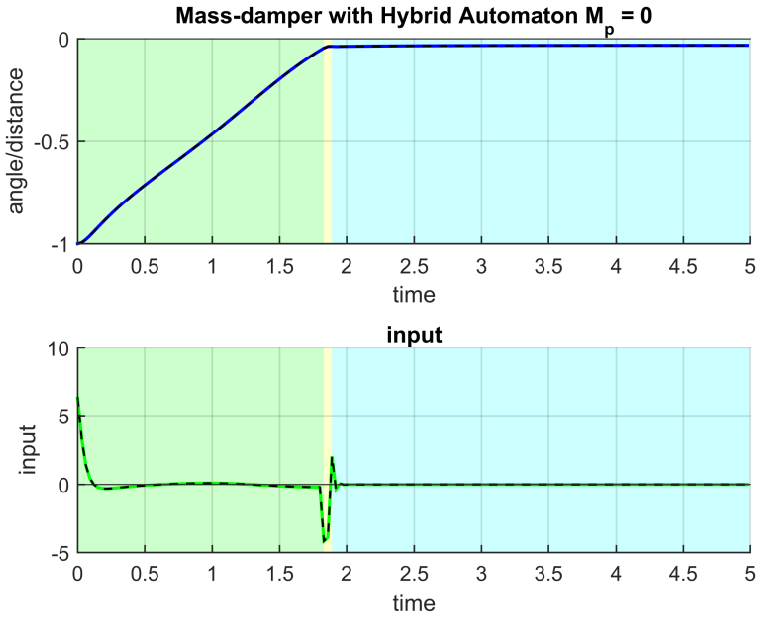
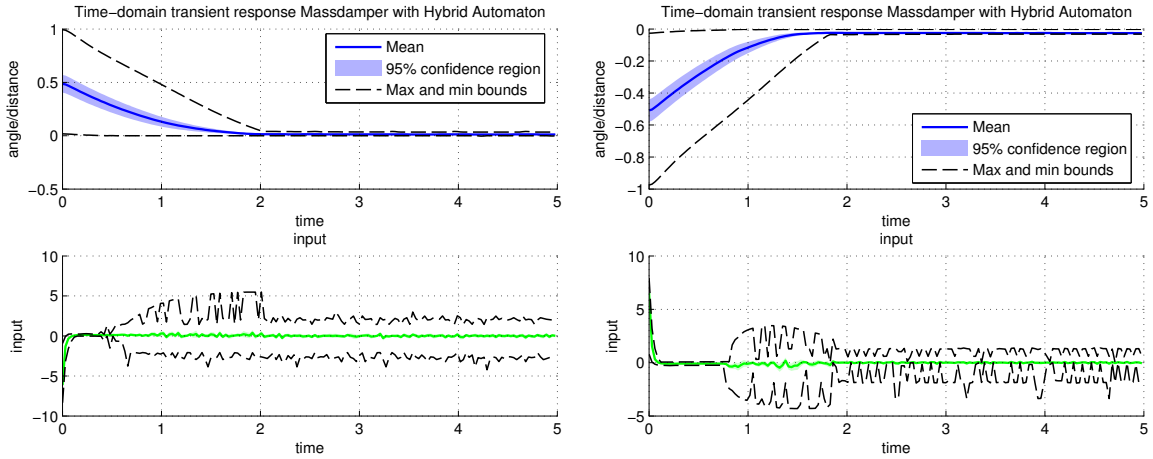


Figure 3-7: A transient response of the mass-damper system and the modes: idle (green), evaluating (yellow), stabilizing (blue) and penalizing (red) which is not visited in this simulation run.

the input plots are the inputs to stop the mass. Since the timing to jump from idle to evaluating mode is not always the same, we see multiple peaks in the input plot. After about two seconds the mass stays near by the center, however there were runs whereby the input was still not zero. This phenomena was visible when q_0 was very close to zero, which is also the cause of noisy learning behavior. Figure 3-9 shows some examples of such undesired results. This is due to our defined invariant set of

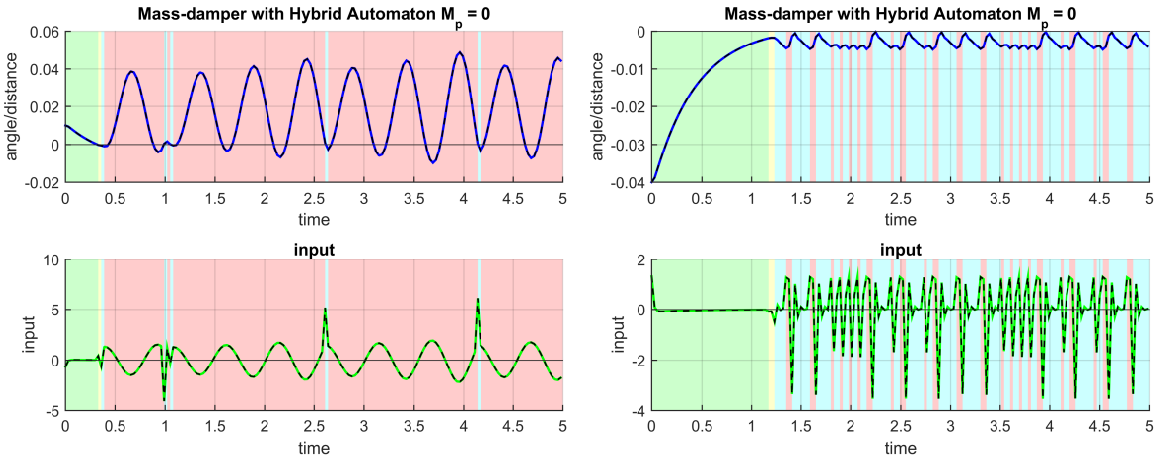


(a) Simulated mean transient response with $q_0 \in [0, 1]$. The undershoot performance is considered.

(b) Simulated mean transient response with $q_0 \in [-1, 0]$. The overshoot performance is considered.

Figure 3-8: Simulation results for the mass position and input force. The simulation is repeated for 50 times.

stabilizing mode, $0.1q_0 < q < 0$ (in case of overshoot). The defined region was too small when $q_0 \in [-0.1, 0.1]$, the controller was not able to stabilize the mass in that region. Therefore the agent visits the penalizing state often as in the figures, and the cumulative reward becomes less. Precise control is still a very complex task for learning controllers, since it makes use of approximators. Also the exploration noise makes it difficult for the agent to control precisely.



(a) The invariant in mode S is $0 < q < 0.001$

(b) The invariant in mode S is $-0.004 < q < 0$

Figure 3-9: Examples of bad results for cases $q_0 \in [-0.1, 0.1]$.

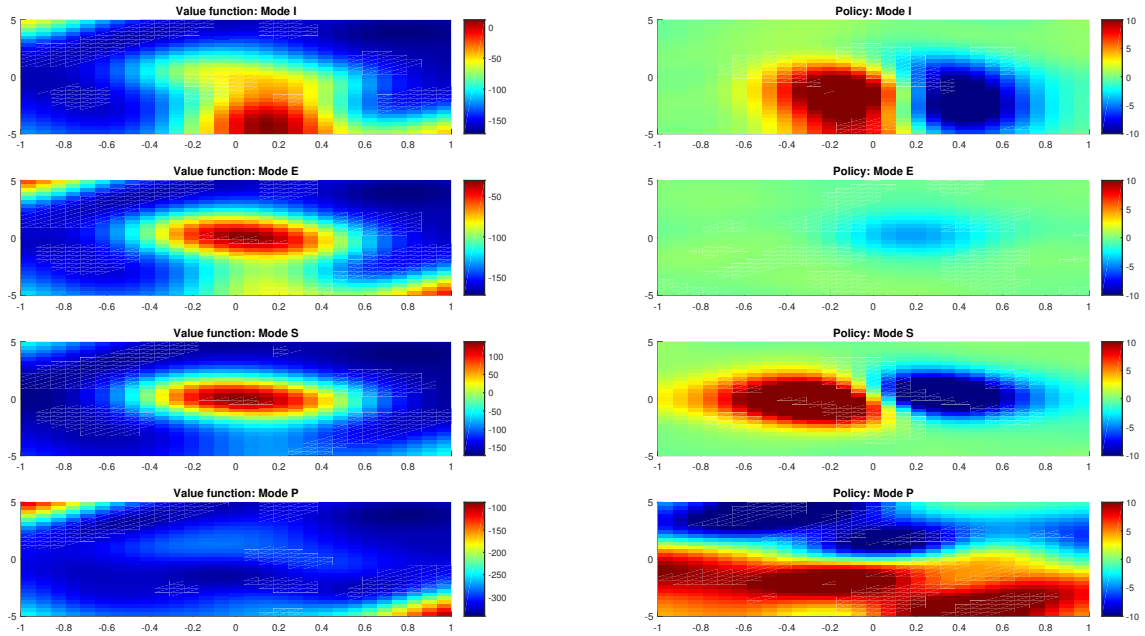
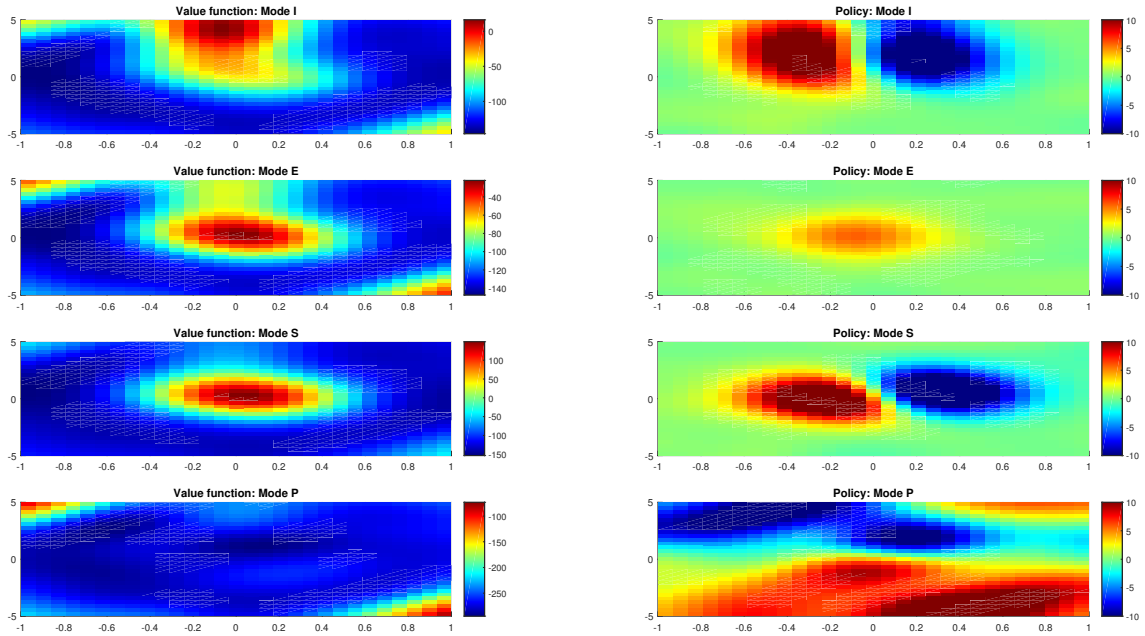


Figure 3-10: Value functions and policy for $q_0 = -1$.

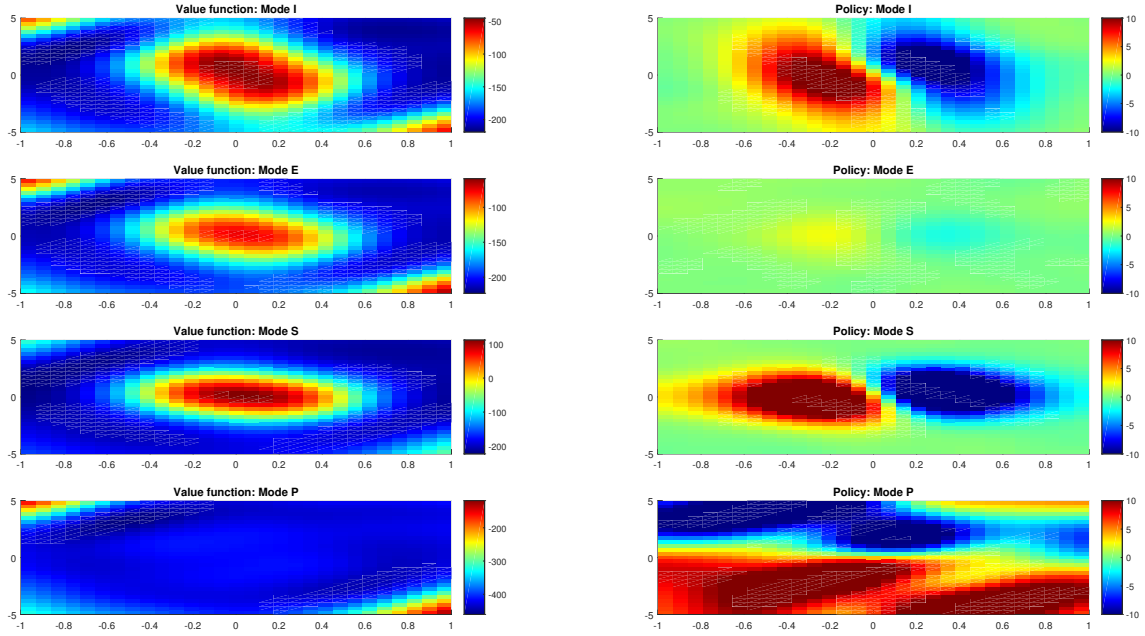
Value functions and policy

Figure 3-10 represents the value functions and policies when $q_0 = -1$. In mode I, the agent begins at $(-1, 0)$ and the value increases when the state becomes close to the center position. Around $q = -0.1$, the automaton jumps over to evaluating mode. The value functions of mode E and S are quite similar since the mode-wise reward functions are similar. Interesting is that the value in mode E is more negative compared with the mode S. This is probably the effect of general part of value function approximation, to keep the gap in value functions small. When the jump from idle to evaluate occurs, the value in both plots are about -50 (yellow in value function of mode I and red in mode E).Since the automaton does not stay long time in mode E, the update rate is low and makes the value over the whole region low. In the policy plot is clearly visible that the controller is giving input against the moving mass, since the value around (0,0) is negative. The policy of mode S is traditional for stabilization problems and it forces the agent to stay around the center. The policy in mode P is quite intuitive: when the velocity is above 0, the input is negative such that the mass is pushed back to the center. Same holds for when velocity is smaller than 0, only the input is then positive.

The value functions and policies from $q_0 = 1$ (Figure **??**(a)) are very similar to Figure 3-10, only reversed in vertical direction such that it prevents undershoot. For intermediate initial positions as $q_0 = 0$, the value functions and policies of $q_0 = -1$ and 1 are merged and they contain characteristics from both (Figure 3-11). It is typical that the policy of mode S is similar for the three initial conditions: it is visited in the later stage of the run and the initial value does not have influence anymore.



(a) $q_0 = -1$



(b) $q_0 = 0$

Figure 3-11: Value functions and policy for (a) $q_0 = 1$ and (b) $q_0 = 0$. The plots of $q_0 = 1$ are very similar to those from $q_0 = -1$, although mirrored vertically. The plots of $q_0 = 0$ is a mix of those from -1 and 1.

3-3-2 Simulation: Mass-damper with rise time

Besides the overshoot, also the rise time performance is considered using the same structure of hybrid automaton. The goal is designing a controller which will go from its initial state to 90% of the desired value within $t_{r,min} = 0.6$ to $t_{r,max} = 0.8$ second. We will use the augmented state $\bar{x} = \begin{bmatrix} q & \dot{q} & q_0 & t & \delta \end{bmatrix}^T$ with the following automaton (Figure 3-12) and mode-wise reward functions (Table 3-3). The most simulation parameters are the same as the overshoot example, as listed in Table 3-2. The trial number is increased to 2000 and the number of basis function is also increased to $10 \times 10 \times 5 \times 5$. For the transition conditions, please see the appendix (B). Since the initial states in $[-0.1, 0.1]$ had difficulties by stabilizing, the set of initial states is changed into $[-1, -0.5]$ to make the problem easier for the controller.

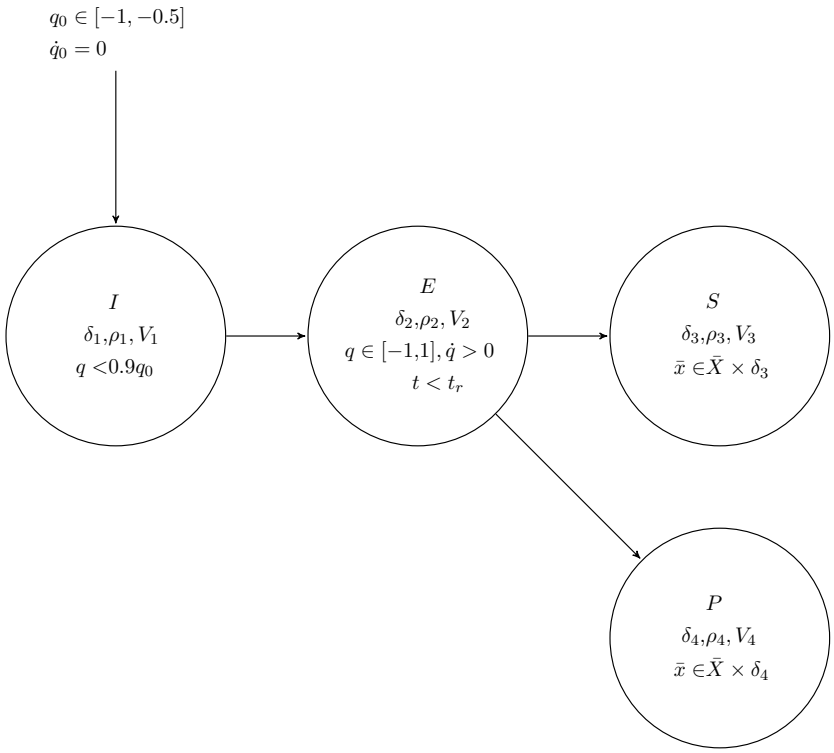


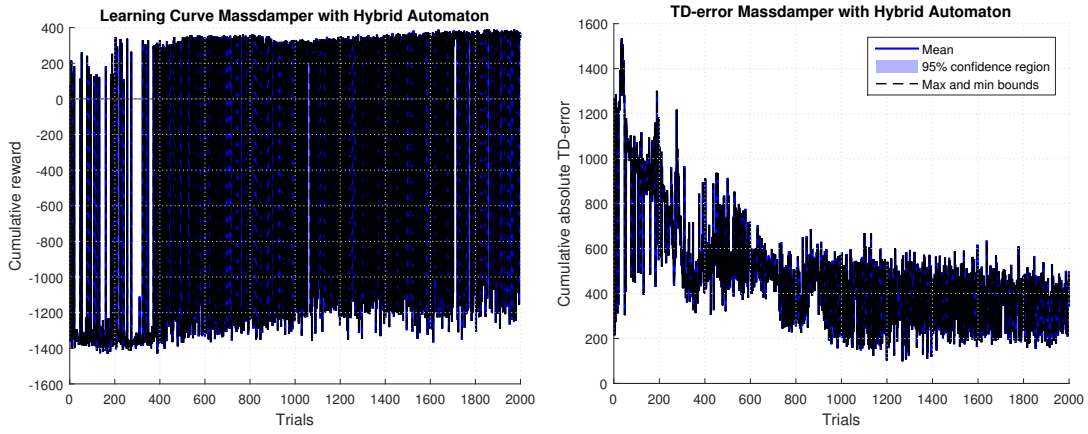
Figure 3-12: Hybrid automaton for rise time performance $t_r = [0.6, 0.8]$. The automaton jumps to evaluating mode after 10% of the desired value has reached. From E, it jumps to either S or P, where the automaton will live for the entire episode time. To avoid that the agent will "wait" at 90% of the desired value until the given time comes, we forbid $\dot{q} < 0$ in E so that it is "rising" to the desired value. If this is violated, the automaton jumps to P and otherwise to S.

The reward function of mode E reinforces the agent to go to 90% of the desired value. Usually by using this reward function, the controller tries to reach the desired value as fast as possible, however we have forbidden this behavior by defining the transition relation from E to P. The automaton jumps to penalizing when it reaches 90% too fast or too slow or when the controller tries to "wait" until the time elapse, i.e. when the velocity is negative.

Table 3-3: Mode-wise reward functions for mass-damper stabilization with rise time performance $t_r = 0.8$.

Mode	Reward function
I	$\rho_1 = \begin{cases} 0 & \text{if } \dot{q} > 0 \mid q_0 < q_{\text{des}} \\ 0 & \text{if } \dot{q} < 0 \mid q_0 > q_{\text{des}} \\ -3 & \text{else} \end{cases}$
E	$\rho_2 = -3(0.1q_{\text{des}} - q)$
S	$\rho_3 = -3\lvert q_{\text{des}} - q\rvert + \begin{cases} 3 & \text{if } \delta = \delta' = \text{S} \\ 0 & \text{else} \end{cases}$
P	$\rho_4 = -9$

Mention here that once the automaton visits the penalizing mode, it will stay there for the rest of the episode time. This makes the learning difficult and especially noisy, as can be seen in Figure 3-13. The edges between penalizing mode and stabilizing mode is erased because the agent could "neglect" some reward losses. Due to the nature of a stabilization problem, how longer the the agent stays at the stabilizing point, how more reward it receives. Then even a penalty is given during the way to reach the goal, it might be seen as an undesirable but acceptable loss since it can compensate it by earning more reward by stabilizing. Obviously, this means that the given policy is not optimal, however there are situations whereby this occurs. It is not convincing that high reward can be earned while the performance was violated and therefore when the performance was violated, it will stay in the penalty mode.



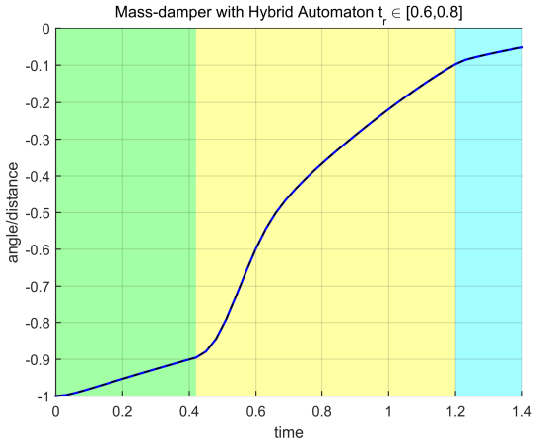
(a) Cumulative rewards.

(b) Cumulative absolute TD-error.

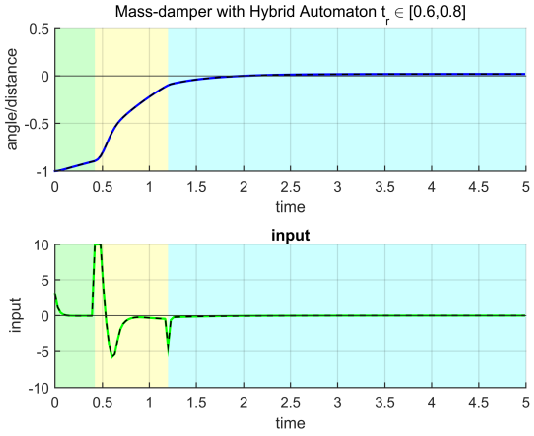
Figure 3-13: Learning behavior for mass-damper with rise time performance.

Transient response

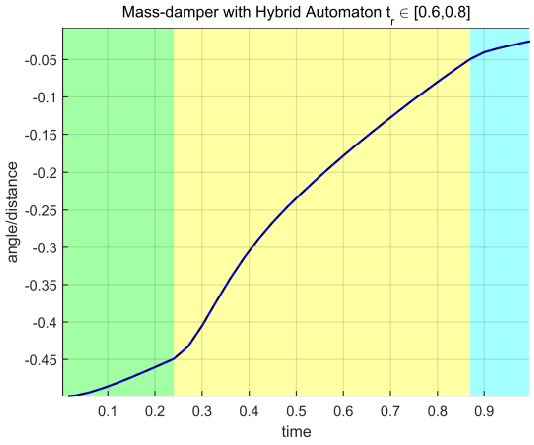
Figure 3-14 shows the simulation results of obtained policy. Figure 3-14(d) shows that for each initial state in $[-1, -0.5]$ the stabilization was succeed. In figures (a) and (c) we



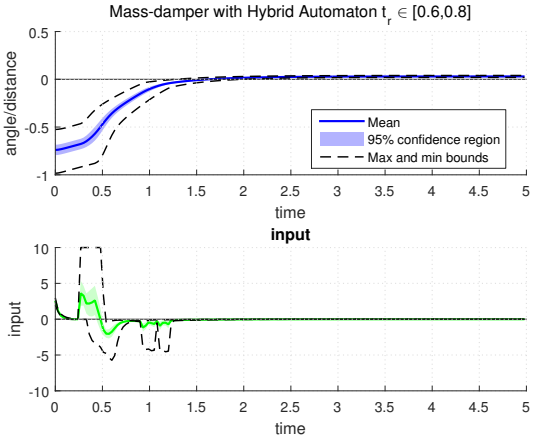
(a) $q_0 = -1$, zoomed in to the significant part.



(b) $q_0 = -1$



(c) $q_0 = -0.5$, zoomed in to the significant part.



(d) Averaged plot for $q_0 \in [-1, -0.5]$ over 30 runs.

Figure 3-14: Transient simulation for (a), (b) $q_0 = -1$ and (c) $q_0 = -0.5$. (d) shows the averaged plot over 30 runs. The color representing again the mode of the automaton: green is mode I, yellow is E and blue is S.

can see that the rise time is about 0.78 and 0.63 seconds respectively, which are within the given range. Since all the simulations are between the max and min bounds created by trajectories of $q_0 = -0.5$ and $q_0 = -1$ (see (d)), we can expect that for all the initial states the rise time performance is fulfilled. We can see that the the controller gives a hard push to the mass after jumped to mode E, and it is adjusting time for reaching $q = 0$ by giving negative input to decrease the velocity. The given control specification is thus achieved. The strange thing is that in mode I the trajectory rises very slowly. Since we had defined in the hybrid automaton to start counting time in mode E, the controller is not trying to gaining time to fulfill the rise time criteria. It might be that however the control criteria was achieved, the policy was still not optimal.

3-3-3 Mass-damper with rise time and overshoot performance

After the policies for two different control criteria are learned, we can create an automaton with both evaluating mode existing. We will obtain an hybrid automaton with five modes, where I, S, P and two E modes are present: one for rise time and the other for overshoot performance. Running the hybrid automaton in Figure 3-16, the simulation result of Figure 3-15 is obtained. Since the separate policy for rise time and overshoot had both achieved the control performance, also the combination of policies fulfills both control specification. As in the plots in the previous subsections, the rise time is about 0.78 and there is no overshoot.

The advantage of this hybrid automaton setting is that by combining multiple modes with its learned policies, a controller for complex task can be obtained. By preparing multiple modes with policies learned to achieve different control performances, the only work to do is create an automaton with desired behavior. By running the automaton, all the performance should be satisfied if the learning of each policy had gone well.

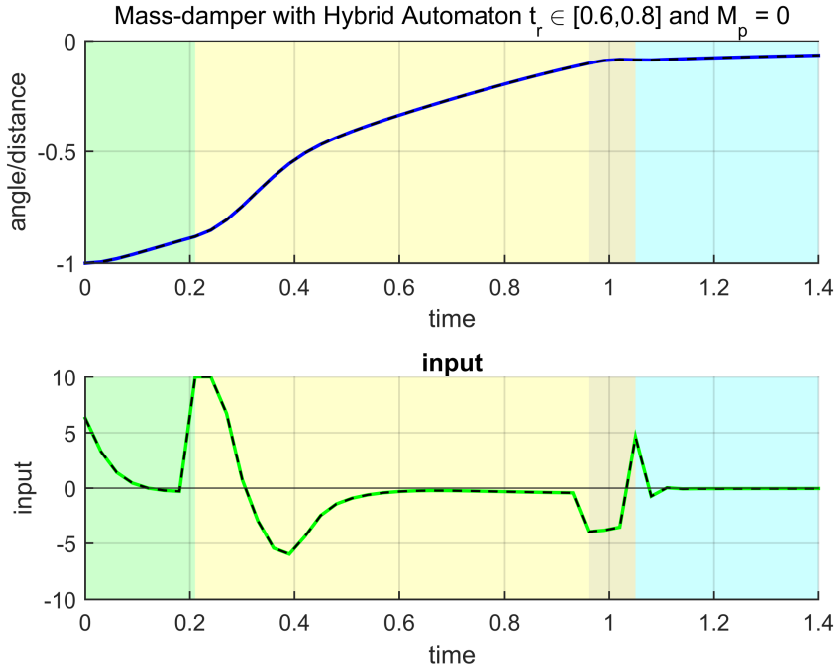


Figure 3-15: Simulation response running the combined automaton with learned policies previously for $q_0 = -1$.

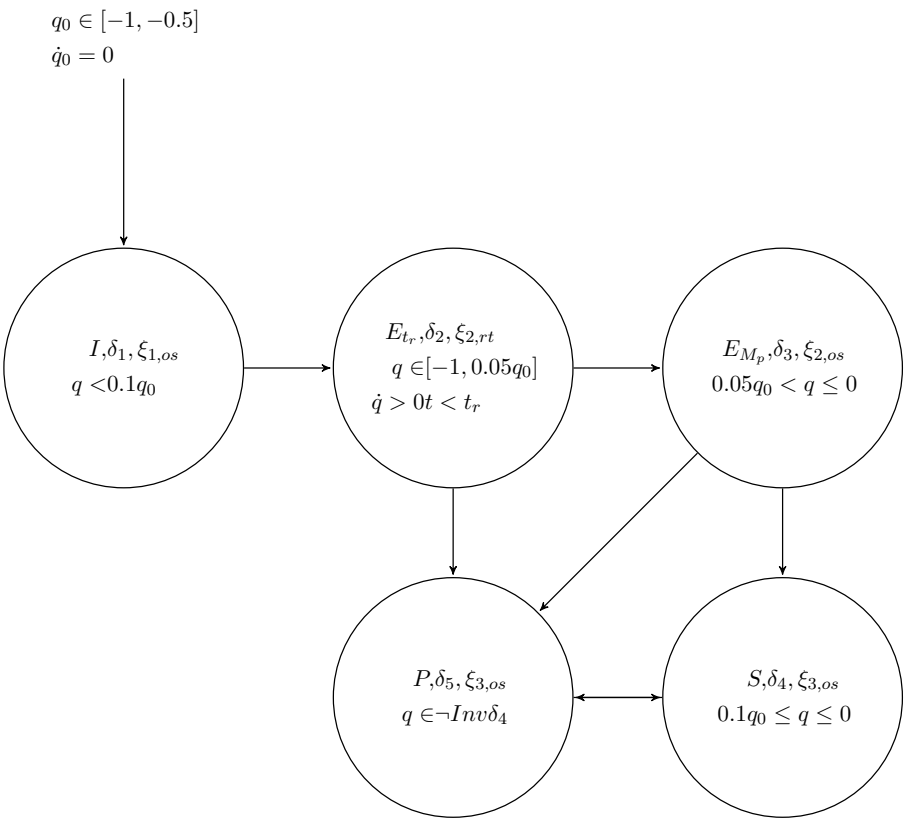


Figure 3-16: Hybrid automaton for rise time performance $t_r \in [0.6, 0.8]$ and $M_p = 0$. The automaton jumps to evaluating mode for rise time after 10% of the desired value has reached. After the given rise time is elapsed or when the 95% is reached, evaluating mode for overshoot becomes active. In the latter case, the performance of rise time is not fulfilled. From the second evaluating mode, it jumps to either S or P.

3-4 Discussions

In this section one approach of reinforcement learning for hybrid automaton is proposed. There is very less work done about RL for a system which containing both discrete and continuous states and to our knowledge, this is the first plain RL for hybrid system. As works that deals with hybrid systems, there are few Hierarchical Reinforcement Learning (HRL) techniques. One example is the options framework ([17], [18]). In this framework, lower level RL learns extended actions, called options, which is saved as a new action. The plain and extended actions could be chosen by the higher level agent to learn its task. In [19] the robot had learned to stand up by combining Q-learning and actor-critic learning. The higher level uses Q-learning to set subtasks for the lower level controller, which uses actor-critic to learn achieving the subtask.

As one of the similarities of the proposed technique and HRL is that in both approaches multiple reward functions are available which only holds for just a part of MDP. Designing reward function for a complex problem is an effort and time consuming job. Especially when you try to incorporate sub reward functions, corresponding

to subgoals, into one global reward function, this may end up in a huge and complex function. However, in both HRL and proposed one, the problem was decomposed into multiple subproblems with its own reward function, rather than combining these multiple reward functions into one. This will end up in relatively simple reward functions which are easier to design.

In the proposed method, each mode with its policy had the function of a building block for the total controller. There is the advantage that the modes can be reused and the reward function design is less complex since only small part of the total problem is covered.

The question remains whether the proposed method designs a proper reward function as defined in the introduction (Chapter 1). It definitely has the availability for tuning and the reward function also contains enough information. The reason of many learning trials was more due to the augmentation of the states and multiple modes. Also the control law fulfilled the required control criteria. However, the design of reward function itself was still on heuristics of the designer.

Chapter 4

Reinforcement learning for Temporal Logic specification

Temporal logic is commonly used for specifying and verifying properties of systems as finite state machines. In this work, temporal logic is used to describe the control performance and it is translated into reward system for RL, consisting from Petri net and reward function. Since a Petri net represents active conditions with tokens, rewards can be represented conveniently by using the marking of the Petri net. Running RL over the system using the reward system, a policy is obtained which should fulfill the given temporal formula, i.e. the control specification.

The proposed method consists of 5 steps.

1. There is a control task for a dynamical system $f(x, u)$ which has to achieve the control performance φ written in Metric Interval Temporal Logic (MITL).
2. Translate φ into a Petri net Γ and a reward function, which will form the reward system of the learning problem.
3. Augment the system state by the marking matrix M such that $\bar{x} = \begin{bmatrix} x \\ M \end{bmatrix}$. The total system becomes $\Sigma(\bar{x}, u) = \begin{cases} f(x, u) \\ \Gamma(x, M) \end{cases}$.
4. Apply RL on the whole system Σ to learn the optimal actor parameter ξ^*.
5. Run $\Sigma(\bar{x}, \pi(\xi^*, \bar{x}))$.

The difficulties of this method are in step 2 and 4, which are discussed further on.

4-1 Reward system design

Using temporal logic, the control specification can be represented conveniently. While it is frequently used as a tool for verification of discrete systems, in this work we use it as a language to represent the desired control performance. The temporal logic expression is translated into a set of Petri net and reward function, whereby the reward function is a function of the Petri net markings. Each syntax of temporal logic discussed in preliminary chapter is assigned to a reward function.

By translating the MITL expressions into a reward system, we will use the traditional stabilization problem as an example:

Example 1: Translate stabilization control performance φ written in temporal logic expressions

$$\varphi = \Diamond\Box\kappa \tag{4-1}$$

with the atomic proposition $\kappa \in AP$ into a reward Petri net Γ. For κ holds

$$\kappa = \top \text{ if } x \in \varkappa \tag{4-2}$$

where $\varkappa \subset X$ is the desired set of states and X is the continuous state space.

Since we will work on continuous state space, the desired set of states should be given instead of a single state. The use of approximators and the existence of exploring noise make the stabilization at a single state impossible. For this kind of problem, the state space could be divided into two subsets: $\varkappa$ and $\neg\varkappa$ where $\varkappa \cap \neg\varkappa = X$. Dividing the whole state space into multiple subsets could also be seen as a rough discretization of the state space. Starting from $x_0 \in \neg\varkappa$, the RL agent has to find a policy which makes 4-1 true.

4-1-1 Modeling Petri net from temporal logic expressions

Since we will design a reward function only dependent on the marking of the Petri net, conversion of temporal formula to Petri net is a necessary and important work. If we define places for each MITL syntax, the Petri net becomes huge for even relatively simple control performance. Also superfluous places will made which models the same part of the system. Therefore, we only have defined Petri nets for particular components of the MITL expressions. By combining all Petri nets, the reward Petri net is formed suitable for evaluating the performance. Thereby we distinct between two types of Petri nets: state related sub Petri net and timing related sub Petri net. These sub Petri nets in parallel, forms the total reward Petri net.

State related sub Petri net models the atomic propositions related to the states. Going back to example 1, the state related Petri net of a stabilization problem will exist from two places. One place for κ and another for $\neg\kappa$. There are also two transitions which connects the two places, to represent the transitions from κ to $\neg\kappa$ and vice versa. On the other hand, timing related sub Petri net models the temporal syntaxes as *Eventually*, $\Diamond$, and *Until*, $\mathcal{U}$. For every temporal syntax a Petri net with at least one

place is defined. The purpose of this Petri net is to define the timing of achieving the state related control performance. Again getting back to example 1, we can design a timing related Petri net consisting from one place. This is due to the temporal syntax $\Diamond$, which implies that the continuing expression $\Box\kappa$ should become true in the future as long there is a token in the place corresponding to $\Diamond$.

Example 1 (continuation): Petri net Γ *modeling 4-1 (Figure 4-1). While living a token in* p_τ, $\Box\kappa$ *should become true in the future to achieve the control specification. However, since stabilizing at* κ *is the only control task and the timing does not make sense, this sub Petri net becomes trivial. Therefore this sub Petri net is neglected further on and the marking will always accounted as* $p_\tau = 1$. *The initial marking* $M_0 = \begin{bmatrix} p_{\neg\kappa,0} & p_{\kappa,0} \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix}$, *the composite change matrix* $D = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}$ *and transition matrix* $T = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}^T = \begin{cases} t_1 = 1 \text{ if } (x_{k+1} \in \varkappa \mid p_{\neg\kappa} = 1) \\ t_2 = 1 \text{ if } (x_{k+1} \in \neg\varkappa \mid p_\kappa = 1) \end{cases}$ *where* t_1 *and* t_2 *are zero for else. The evolution of Petri net becomes*

$$\Gamma(x, M) : M_{k+1} = T_{k+1}(x, M)D + M_k \tag{4-3}$$

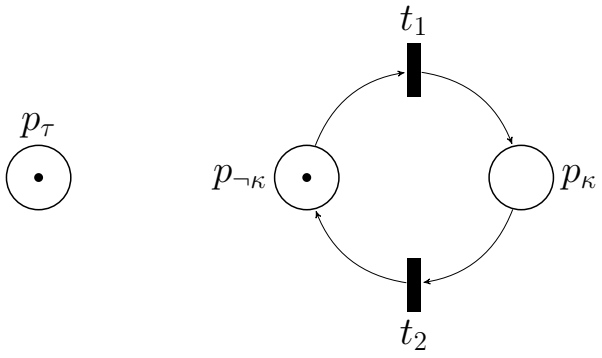
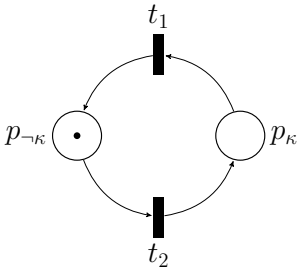
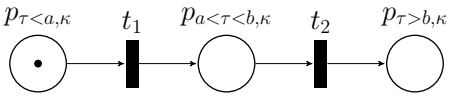
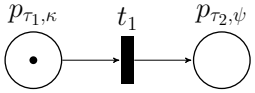


Figure 4-1: *Petri net modeling control performance 4-1. Right is the state related sub Petri net with its initial marking. Left the single place with token is timing related sub Petri net.*

Table 4-1 shows the sub Petri nets with its corresponding temporal syntax and/or atomic proposition. Mention here that for modeling *Globally*, $\Box$, the time related Petri net is not defined, since the continuing expression should always hold and defining timing becomes trivial. If the control performance contains multiple atomic propositions or temporal syntaxes, then the total reward Petri net becomes the corresponding Petri nets in parallel. Although there are also situation whereby multiple Petri nets in parallel could be simplified as a single Petri net. Especially in situation whereby the whole state space is divided into multiple subsets, this simplification is effective to decrease the total number of places and tokens. In the simulation section there are multiple examples where such simplification is applied (see 4-3-2 and **??**). Thereby the way of placing the transitions would be totally problem dependent and there are no fixed rules

how to connect the places. Also the conditions when the transitions will fire is problem specific. For the temporal MITL operators with time interval, the transitions will fire according to time.

Table 4-1: Table for translating temporal syntaxes into sub Petri nets for reward system. The sub Petri nets could be distinct between timing and state related Petri nets. The atomic propositions κ and ψ are true if $x \in \varkappa$ and $x \in \Psi$ respectively where $\varkappa, \Psi \subset X$.

State related sub Petri nets	
Atomic proposition	Sub Petri nets
κ	
Timing related sub Petri nets	
Temporal operator	Sub Petri nets
Eventually, $\Diamond\kappa$	
Eventually (MITL), $\Diamond_{\{a,b\}}\kappa$	
Until, $\kappa \,\mathcal{U}\, \psi$	

4-1-2 Metric Interval Temporal Logic reward functions

For the given MITL control performance, we will assign a reward function to each operator and atomic proposition. Reward function assigned to unary operators is a function of the expression behind it. Binary operators as $\mathcal{U}$ (**U**ntil) and $\wedge$ (AND, conjunction), is a function of expressions before and behind it. Thus for 4-1, the reward function becomes $r_\Diamond(r_\Box(r_\kappa(M_k, M_{k+1})))$. Since it is usual in reward function design to consider the states after the action is applied, in this thesis $p_{(\cdot)}$ represents the number of tokens in place $(\cdot)$ at time step $k+1$. For number of tokens at time step k, the tilde operator is used such that $\tilde{p}_{(\cdot)}$ represents the number of tokens in place $(\cdot)$ at time step k.

Table 4-2 shows the assigned reward functions for the operators and atomic proposition. Each reward function could be multiplied by some cost to weigh each temporal formula differently, however for the convenience it is not written in the table. Usually for control task whereby a certain condition has to hold the entire run, e.g. reaching the goal while avoiding the obstacle $\Diamond\kappa \wedge \Box\neg\psi$, then that condition $\Box$ should be weighed heavier (see

4-3-2). Also when a performance occurs for a short time, the weighting factor should become large to put emphasis on that specific performance.

Table 4-2: Table representing reward functions assigned to temporal formulae. $p_{(\cdot)}$ is the number of tokens in place $(\cdot)$.

Temporal formula	Reward function	Explanation
κ	$r_\kappa = p_\kappa - p_{\neg\kappa}$	Reward is given if κ is true and penalize if κ is false.
$\neg\kappa$	$r_{\neg\kappa} = -r_\kappa$	Multiply the reward above by -1, such that reward is given if $\neg\kappa$ is true.
$\Box\kappa$	$r_{\Box\kappa} = (1+\tilde{p}_\kappa)r_\kappa$	Give extra reward when κ is true for two time steps in sequence.
$\Diamond\kappa$	$r_{\Diamond\kappa} = \tilde{p}_{\tau,\kappa} r_\kappa$	Give reward if κ should become true in the future.
$\Diamond_{\{a,b\}}\kappa$	$r_{\Diamond\kappa} = \tilde{p}_{a<\tau<b,\kappa} r_\kappa$	Give only reward within time a and b such that κ becomes true within the given time interval.
$\kappa \wedge \psi$	$r_\wedge = \min(r_\kappa, r_\psi)$	Penalize if one of the condition is not true.
$\kappa \,\mathcal{U}\, \psi$	$r_\mathcal{U} = p_{\tau_1} r_\kappa + p_{\tau_2} r_\psi$	Reward if κ is true until ψ becomes true. Thereafter, give only reward if ψ is true.

Example 1 (continuation): The reward function for 4-1 becomes

$$\begin{aligned} r_{\Diamond\Box\kappa}(M_k, M_{k+1}) = r_\Diamond(r_\Box(r_\kappa(M_k, M_k+1))) &= \tilde{p}_{\tau,\kappa}(1+\tilde{p}_\kappa)(p_\kappa - p_{\neg\kappa}) \\ &= (p_\kappa - p_{\neg\kappa}) + \tilde{p}_\kappa p_\kappa - \tilde{p}_\kappa p_{\neg\kappa}. \end{aligned} \tag{4-4}$$

Notice $p_{\tau,\kappa}$ is always 1. The terms between the brackets are rewards corresponding to the position states, the reward is positive if κ is true and negative else. The third term will give extra reward if a token lives in p_κ for two time steps in sequence, thus it reduces transitions and forces the agent to let not fire the transition. If nevertheless the transition fires and the token p_κ is consumed, the last term will extra penalize the agent.

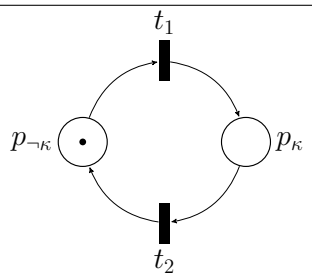
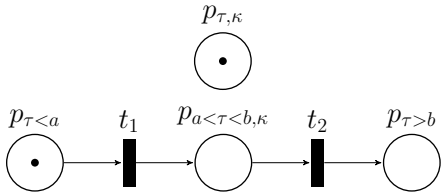
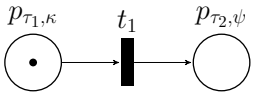
4-1-3 Table of reward system

Table 4-3 summarizes the translation of temporal formulae into Petri net rewarding system.

4-2 Reinforcement learning with Petri nets

After the control performance φ is translated into Petri net Γ with reward function r_φ, the system is augmented. We will obtain an augmented state $\bar{x} = \begin{bmatrix} x \\ M \end{bmatrix}$ and total system $\Sigma(\bar{x}, u) = \begin{cases} f(x,u) \\ \Gamma(\bar{x}) \end{cases}$. Since the most existing RL methods only deals with either discrete or continuous states, we need to rearrange the consisting method. We will use the standard actor-critic method as a base for our method, Standard Actor-Critic with Petri Net (SACPN).

Table 4-3: Table summarizing translation of temporal formulae into Petri net reward system. For each temporal formula a cost could be multiplied to tune the importance of each control task. $p_{(\cdot)}$ is the number of tokens at time step $k+1$ in place $p_{(\cdot)}$, while $\tilde{p}_{(\cdot)}$ is the number of tokens at previous time step k in place $p_{(\cdot)}$

Temporal formula	Reward function	Sub Petri net
κ	$r_\kappa = p_\kappa - p_{\neg\kappa}$	
$\neg\kappa$	$r_{\neg\kappa} = -r_\kappa$	
$\Box\kappa$	$r_{\Box\kappa} = (1+\tilde{p}_\kappa) r_\kappa$	
$\Diamond\kappa$	$r_{\Diamond\kappa} = \tilde{p}_{\tau,\kappa} r_\kappa$	
$\Diamond_{\{a,b\}}\kappa$	$r_{\Diamond\kappa} = \tilde{p}_{a<\tau<b,\kappa} r_\kappa$	
$\kappa \wedge \psi$	$r_\wedge = \min(r_\kappa, r_\psi)$	
$\kappa \,\mathcal{U}\, \psi$	$r_\mathcal{U} = p_{\tau_1} r_\kappa + p_{\tau_2} r_\psi$	

4-2-1 Approximators

Since Petri net is a discrete way of modeling the system, using approximators to approximate the marking $\hat{M}$ will not be a proper way to compute the state of the Petri net. For large Petri net with many places becomes the dimension of the basis functions also large, which requires a lot of computation power. Thereby parallel Petri nets with multiple tokens in the whole system makes the computation even more complex. Therefore we suggest to assign parameter vectors of the approximators to each place, such that each place has its own approximated policy $\hat{\pi}$ and value function $\hat{V}$. The total policy and value function are the sum of assigned policies and value functions of the places where the tokens live in.

Assume we have a total system $\Sigma(\bar{x}, u) = \begin{cases} f(x,u) \\ \Gamma(x,M) \end{cases}$, where $x \in X^{n\times 1}$ and $M \in P^{1\times m}$.

Then assign for each place i estimated policy $\hat{\pi}_i$ and value function $\hat{V}_i$:

$$\hat{\pi}_i(\xi, \bar{x}) = \frac{p_i}{\sum M} \xi_i^T \phi_a(x) \tag{4-5}$$

$$\hat{V}_i(\theta, \bar{x}) = \frac{p_i}{\sum M} \theta_i^T \phi_c(x) \tag{4-6}$$

where $\phi_a(x) \in \mathbb{R}^s$ and $\phi_c(x) \in \mathbb{R}^t$ are basis functions of actor and critic respectively, p_i is the number of tokens in place i and $\sum M > 0$ is the total number of tokens, assuming there is at least 1 token in the system every time. We will use linear parameterization due to its simplicity and convenient representation. Mention hereby that the number of tokens is a measure for dominance of the place corresponding policy. The policy is divided by the total number of tokens to avoid continuous increasing of the TD-error in case when transition exists which fires tokens without consuming any. Although in this work, there are no situations whereby two or more tokens live in a place, thus every policy weighs all even as long there is a token at the place.

The total policy and value function is the sum over all places

$$\begin{aligned} \hat{\pi}(\Xi, \bar{x}) = \sum_m \hat{\pi}_i(\xi, \bar{x}) &= \sum_m \frac{p_i}{\sum M} \xi_i^T \phi_a(x) \\ &= \frac{1}{\sum M} M \Xi^T \phi_a(x) \end{aligned} \tag{4-7}$$

$$\begin{aligned} \hat{V}(\Theta, \bar{x}) = \sum_m \hat{V}_i(\theta, \bar{x}) &= \sum_m \frac{p_i}{\sum M} \theta_i^T \phi_c(x) \\ &= \frac{1}{\sum M} M \Theta^T \phi_c(x) \end{aligned} \tag{4-8}$$

where

$$\Xi = \begin{bmatrix} \xi_1 & \xi_2 & \dots & \xi_m \end{bmatrix} \in \mathbb{R}^{m \times s} \text{ and } \Theta = \begin{bmatrix} \theta_1 & \theta_2 & \dots & \theta_m \end{bmatrix} \in \mathbb{R}^{m \times t}. \tag{4-9}$$

4-2-2 Standard Actor Critic with Petri Net

Because the parameters are now matrices instead of vectors, also the update function of the parameters is changed. The update rules of actor (2-11) and critic (2-17) becomes

$$\Xi_{k+1} = \Xi_k + \alpha_a \delta_{k+1} \Delta u_k \phi_a M_{k+1} \tag{4-10}$$

$$\Theta_{k+1} = \Theta_k + \alpha_c \delta_{k+1} e_{k+1} M_{k+1}. \tag{4-11}$$

Rest of the computation remains the same as in the original SAC. For clarity the pseudo code of SAC with Petri nets is given in Algorithm 1.

Algorithm 1 Standard Actor-Critic with Petri Net (SACPN)

Input: γ, λ, learning rates α and reward function r_φ

$e_0 = 0$
Initialize x_0, M_0 and function approximators
Apply random input u_0
$k \leftarrow 0$
loop
  Determine x_{k+1}, M_{k+1} and compute $r_{\varphi_{k+1}}(M_k, M_{k+1})$
  $\delta_{k+1} \leftarrow r_{k+1} + \gamma V_{\Theta_k}(\bar{x}_{k+1}) - V_{\Theta_k}(\bar{x}_k)$
  // **Update actor**
  $\Xi_{k+1} \leftarrow \Xi_k + \alpha_a \delta_{k+1} \Delta u_k \nabla_\Xi \hat{\pi}_{\Xi,k} M_{k+1}$
  // **Update critic**
  $e_{k+1} \leftarrow \lambda\gamma e_k + \nabla_\Theta \hat{V}_{\Theta_k}(\bar{x}_k)$
  $\Theta_{k+1} \leftarrow \Theta_k + \alpha_c \delta_{k+1} e_{k+1} M_{k+1}$
  // **Choose action**
  $u_{k+1} \leftarrow \hat{\pi}_{\Xi_{k+1}}(\bar{x}_{k+1})$
  Choose exploration $\Delta u_{k+1} \sim \mathcal{N}(0, \sigma^2)$
  Apply $u_{k+1} + \Delta u_{k+1}$
  $k \leftarrow k + 1$
end loop

4-3 Simulation experiments

The proposed method is validated on simulation using a pendulum model for 5 different control tasks. Table 4-5 shows the desired control performance and its translation to ordinary human language. For all control performance, the task is to swing the pendulum from its initial position to the desired positions. The fully measurable state $x = \begin{bmatrix} q & \dot{q} \end{bmatrix}^T$ consists of the angle q and the angular velocity $\dot{q}$ of the pendulum. Only for the first control task the input signal is limited to $u \in [-3, 3]$ such that the pendulum has to build up momentum to be able to swing up. For other tasks hold $u \in [-10, 10]$ which is enough input to swing the pendulum freely. For the model of the pendulum, the readers are referred to the appendix (see A-2). Also the models of Petri nets are given in the appendix (see C). The simulation parameters are given in table 4-4.

Table 4-4: Simulation parameters used for simulation experiments of SACPN. The number of trials are different for each problem.

Simulation parameters	Symbol	Value	Units
Trial duration	T_t	3	s
Sample time	T_s	0.03	s
Discount factor	γ	0.97	-
Eligibility trace decay	λ	0.67	-
Learning rate critic	α_c	0.3	-
Learning rate actor	α_a	0.1	-
Number of radial basis functions	-	10×10	-

Table 4-5: The experimented 5 control performance. The atomic propositions κ, ψ and υ are true if $x \in \varkappa, \Psi, \Upsilon$ respectively where $\varkappa, \Psi, \Upsilon \subset X$ holds respectively.

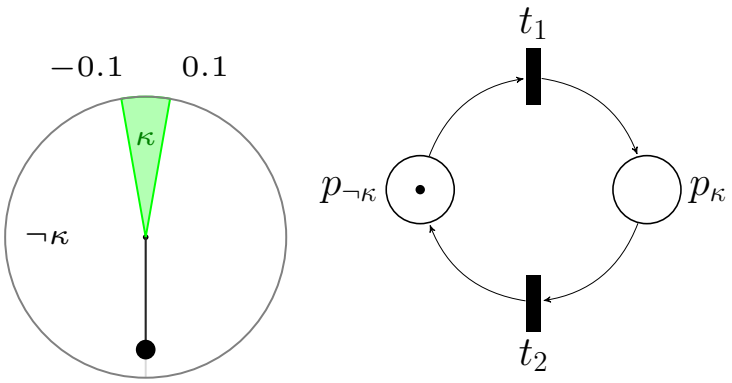
	Temporal formula	Human language
1	$\Diamond\Box\kappa$	Stabilization problem. Visit the desired subset $\varkappa$ and stay there for the entire time.
2	$\Diamond\Box\kappa \wedge \Box\neg\psi$	Stabilization with obstacle avoidance. Stabilize at $\varkappa$ but do not visit Ψ during the run.
3	$\Diamond\Box\kappa \wedge \Box\neg\psi \wedge \Box\upsilon$	Stabilization with obstacle avoidance and velocity limit. Stabilize at $\varkappa$ while never visiting Ψ and the velocity should always within the values defined by Υ.
4	$\Box\Diamond\kappa \wedge \Box\Diamond\upsilon \wedge \Box\neg\psi$	Visit $\varkappa$ and Υ alternately while avoiding Ψ.
5	$\neg\kappa \, \mathcal{U}_{\{1,2\}} \left(\kappa \, \mathcal{U}_{\{2,\infty\}} \, \Box\neg\kappa\right)$	Visit $\varkappa$ within a to b seconds, however do not visit before and after a and b seconds.

4-3-1 Simulation 1: $\Diamond\Box\kappa$

The first control performance is a traditional stabilization problem for an inverted pendulum with input signal limitation. The goal is to keep the pendulum upwards while starting form the downward position. Since the maximum input signal cannot deliver enough power to swing up the pendulum at once, the controller has to build up momentum to be able to swing the pendulum up. Figure 4-2 shows the region of κ and the reward Petri net used for the learning. The reward function for this problem is the same as in the example 4-4

$$r_1(M_k, M_{k+1}) = \tilde{p}_{\tau,\kappa}(1 + \tilde{p}_\kappa)(p_\kappa - p_{\neg\kappa}) \tag{4-12}$$

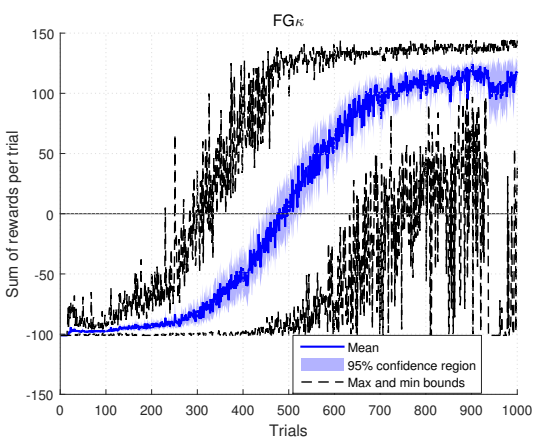
where $\tilde{p}_{(\cdot)}$ is the number of tokens at previous time step k at place $p_{(\cdot)}$. Mention that $p_{\tau,\kappa}$ is always 1 and by modeling the Petri net it is neglected since single place without transitions is trivial.



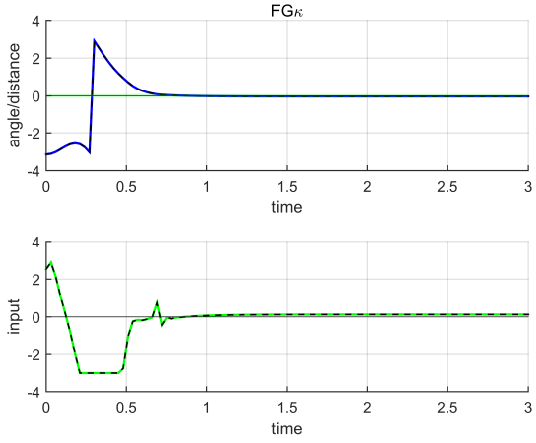
(a) Picture showing when the atomic proposition κ is true.

(b) The reward Petri net for $\Diamond\Box\kappa$.

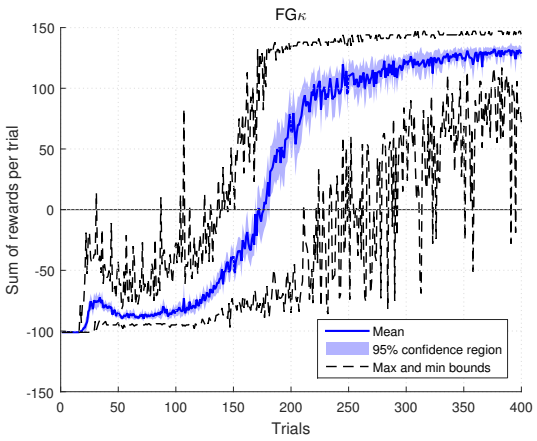
Figure 4-2: Picture describing the problem 1 and the corresponding reward Petri net.



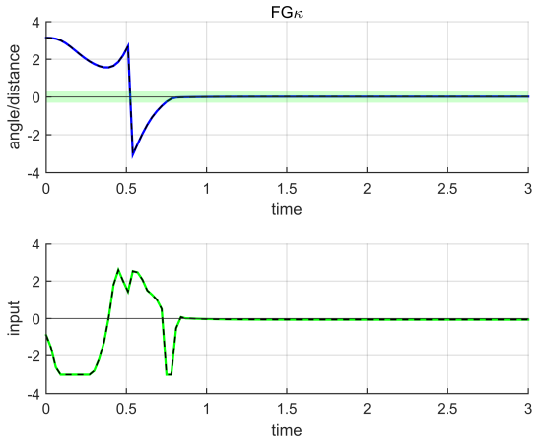
(a) Averaged Learning curve over 30 runs, where $\kappa = \top$ if $-0.1 < q < 0.1$. Notice there are trials whereby the learning was not done well.



(b) Transient simulation of swinging up pendulum using parameters from a succeed learning trial. The green region $-0.1 < q < 0.1$ is the region where κ is true.



(c) Averaged Learning curve over 30 runs, using different condition of κ. The controller had learned a good policy in about 400 trials, which is much faster than the previous result.



(d) Transient simulation, the green region shows the subset $-0.3 < q < 0.3$ where $\kappa = \top$.

Figure 4-3: Results of learning problem 1

Result

Figure 4-3(a) and (b) shows the averaged learning curve over 30 runs and the transient simulation without noise. From the learning curve we can notice that the learning was not gone acceptable. By learning longer time we can expect that the curve will converge nicely. The reason why the learning take time is because of the very small region of κ. Another experiment is done with broadened desired subset. Instead of $\kappa = \top$ if $-0.1 < q < 0.1$, we have used $\kappa = \top$ if $-0.3 < q < 0.3$. The result is visible on Figure 4-3(c) and (d). The agent was able to learn swinging up much faster while transient response is similar. Defining the conditions of atomic proposition should be

done carefully such that the main task does not change. However by implementing it in a smart way, we can make the learning easier for the controller.

Also interesting is that even you assign a subset as a goal to reach, the controller stabilizes at $q = 0$. Because of the exploration noise, it can happen that pendulum crosses over the border of the desired region. To avoid this the controller learns to stay close to a state far away from the border, which is the center of the region.

4-3-2 Simulation 2: $\Diamond\Box\kappa \wedge \Box\neg\psi$

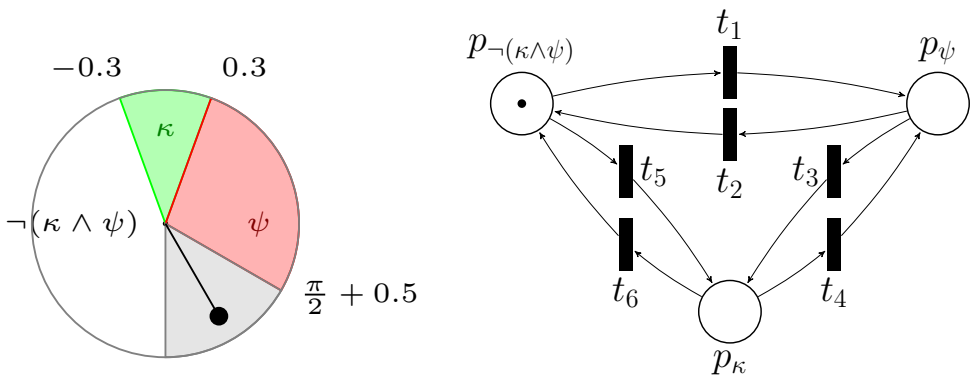
The second task is to stabilize the pendulum again at upwards position, however during the swinging up it should avoid region ψ (see Figure 4-4). The gray part in the figure is the set of initial states, thus the fastest way to reach κ is to swing up counter clock-wise however the task is to let it learn swinging up clock-wise. Notice that the reward Petri net is simplified combining $p_{\neg\kappa}$ and $p_{\neg\psi}$ as $p_{\neg(\kappa\wedge\psi)}$. Mention that this simplification and way of connecting the transitions are completely problem dependent. The reward function is

$$r_2(M_k, M_{k+1}) = \min\left(\tilde{p}_{\tau,\kappa}(1+\tilde{p}_\kappa)(p_\kappa - p_{\neg\kappa}), 10(1+\tilde{p}_{\neg\psi})(p_{\neg\psi} - p_\psi)\right) \tag{4-13}$$

where

$$p_{\neg\kappa} = p_\psi + p_{\neg(\kappa\wedge\psi)} \text{ and } p_{\neg\psi} = p_\kappa + p_{\neg(\kappa\wedge\psi)} \tag{4-14}$$

representing the total number of tokens from places where $\neg\kappa$ and $\neg\psi$ holds respectively. The reward function of $\Box\neg\psi$ is weighed heavier to be sure to avoid the artificial obstacle. Since we take the minimum of two rewards, as long as one of the condition is violated a negative reward is given. The controller could only receive the largest reward when the both control specifications are fulfilled.



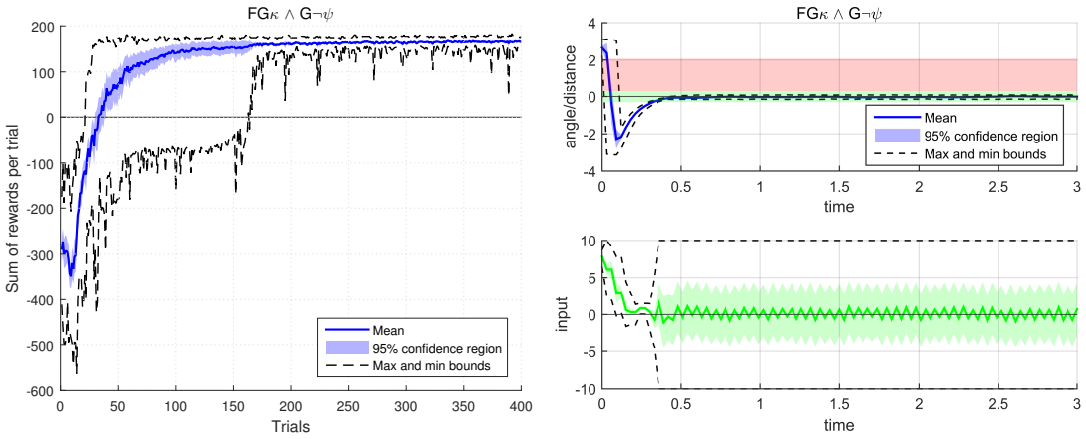
(a) Picture showing the corresponding regions of the atomic propositions κ and ψ. The gray region is the set of initial states.

(b) The reward Petri net for $\Diamond\Box\kappa \wedge \Box\neg\psi$.

Figure 4-4: Picture describing the problem 2 and the corresponding reward Petri net.

Result

The averaged learning curve in Figure 4-5 shows a very nice convergence behavior of learning. In the transient plot it seems as the pendulum has visited the non-desired region, but this is because of the jump from π to $-\pi$. Due to the circular motion, $q = \pi$ represents the same position as $q = -\pi$.



(a) Averaged Learning curve over 30 runs.

(b) Transient simulation.

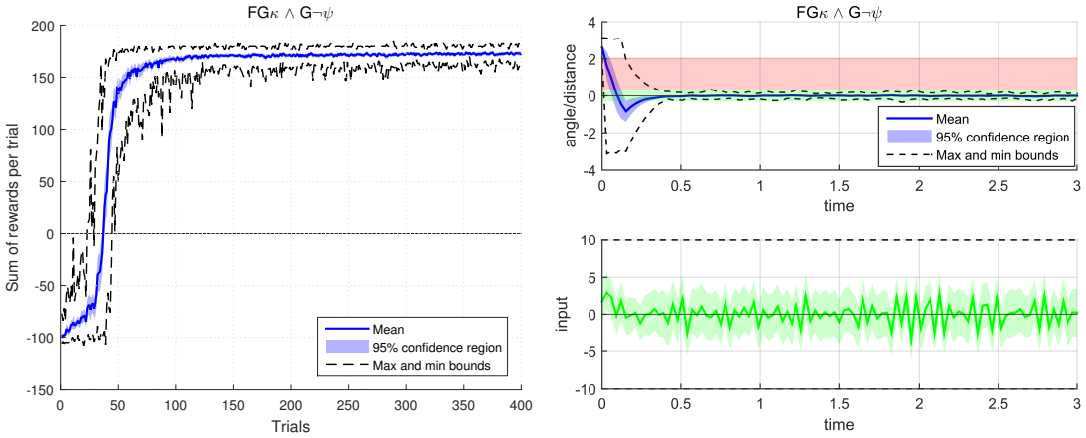
Figure 4-5: Results of learning problem 2.

Figure 4-6 shows the result of simulation run using different weighting factors for reward function. The used reward function was

$$r_{2'}(M_k, M_{k+1}) = \min\left(\tilde{p}_{\tau,\kappa}(1+\tilde{p}_\kappa)(p_\kappa - p_{\neg\kappa}), (1+\tilde{p}_{\neg\psi})(p_{\neg\psi} - p_\psi)\right) \tag{4-15}$$

whereby both terms of the reward function had the same weight. In Figure 4-6(b) it is clearly visible that the condition of $\Box\neg\psi$ was not always fulfilled, while the learning curve had shown a very nice convergence in Figure 4-6(a).

Since the assigned reward function for a particular specification does not take other specifications in consider, the given reward in $\neg(\kappa \wedge \psi)$ and ψ are the same: the reward from $\Box\neg\psi$ gives a positive reward but it has no influence on the total reward because of the minimum operator. Only when the transition $\neg(\kappa \wedge \psi) \rightarrow \psi$ occurs, an extra penalty is given. Therefore, in unweighed situation the controller can choose for the trajectory counter clock-wise, because it can "compensate" the extra penalty of transition by reaching the desired subset quickly and collecting there the rewards. By weighing the specification heavily, the behavior to get over the undesired region can be reduced. However, this way of rewarding is actually not a natural way of rewarding, since the weighing factor will depend on the simulation time. Usually for longer simulation time, the controller can "compensate" more penalties received during the trajectory. The reward function is natural in narrow time perspective since the controller receives always penalties as long as the specification is not fulfilled. Although, in wide time perspective the reward function is still not natural because the weighting factors decides whether the control performance will be achieved or not. Mention in the figure that



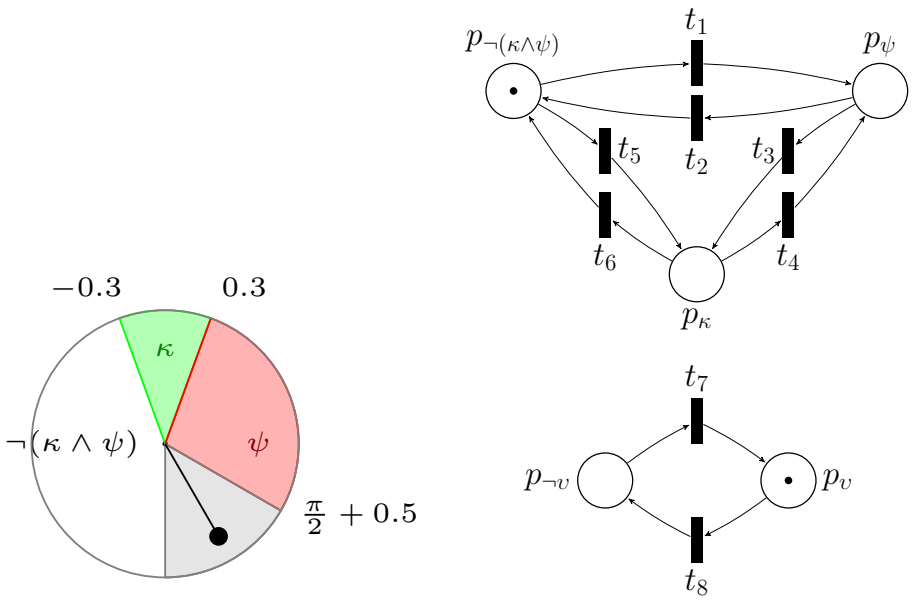
(a) Averaged learning curve over 30 runs.

(b) Averaged transient response simulated using the learned policies over 30 runs.

Figure 4-6: Results for learning using reward function with different weighting factors.

the value which the learning curves converge are the same as in the weighted reward function because of the minimum operator.

4-3-3 Simulation 3: $\Diamond\Box\kappa \wedge \Box\neg\psi \wedge \Box\upsilon$



(a) The way the regions divided is the same as in problem 2.

(b) The reward Petri net for $\Diamond\Box\kappa \wedge \Box\neg\psi \wedge \Box\upsilon$.

Figure 4-7: Picture describing the problem 3 and the corresponding reward Petri net.

Third control performance is an arranged version of the second one. As in the second problem, the agent should stabilize upwards while avoiding a certain region but thereby a velocity constraint is added. The angular velocity $\dot{q}$ should always below 10 rad/s, thus $\upsilon = \top$ if $\mid \dot{q} \mid < 10$ holds. The reward function is

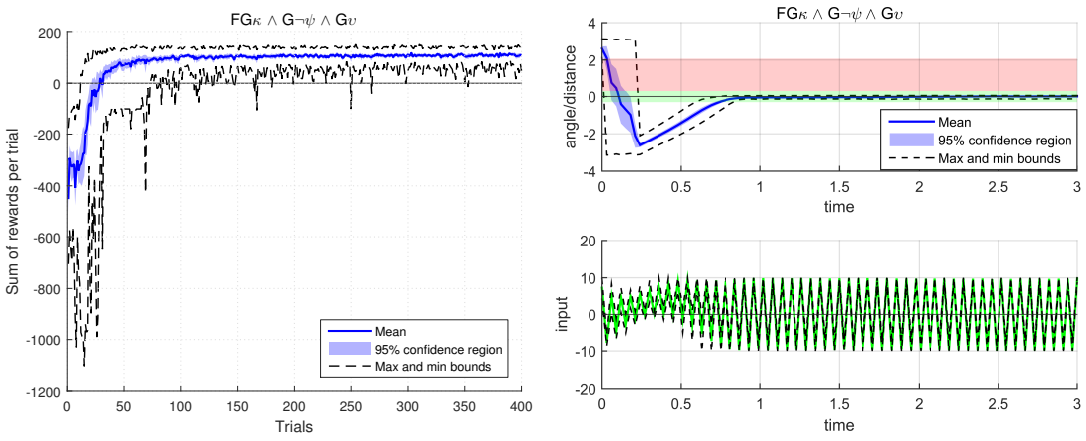
$$r_3(M_k, M_{k+1}) = \min\left(\tilde{p}_{\tau,\kappa}(1+\tilde{p}_{\kappa})(p_{\kappa} - p_{\neg\kappa}), 10(1+\tilde{p}_{\neg\psi})(p_{\neg\psi} - p_{\psi}), 10(1+\tilde{p}_{\upsilon})(p_{\upsilon} - p_{\neg\upsilon})\right). \tag{4-16}$$

The weighting costs are found heuristically.

Result

Figure 4-8 shows the result when also the angular velocity is constrained. Obviously the pendulum reaches the upward position much slower than in the previous problem. Figure 4-9 shows the plot of angular velocity of both problems, and it is clearly visible that the high velocity peak in problem 2 is vanished in problem 3. The controller reduces the velocity by alternating the input very fast, such that the acceleration will not become too large.

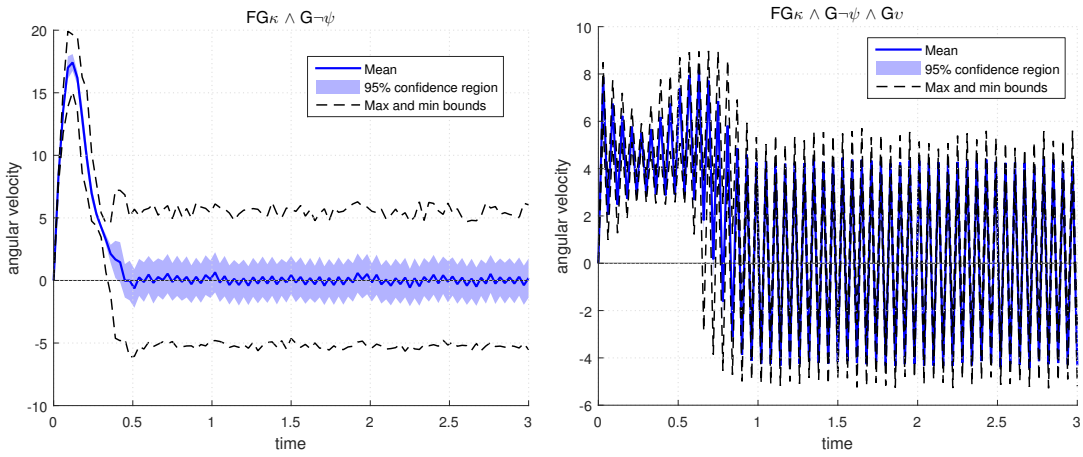
In this problem, there were cases whereby the pendulum learned to swung to the upward position directly, violating the latter two control specifications. Thereby the controller tried again to compensate the large penalties by rewards for staying at the upward position. From the fact that most of the time the desired controller was learned, a number of bad learning might be because of the choice of learning parameters. Maybe the chosen actor learning rate was too large whereby the convergence to the optimal policy became difficult.



(a) Averaged Learning curve over 30 runs.

(b) Transient simulation.

Figure 4-8: Results of learning problem 3.



(a) Problem 2.

(b) Problem 3 with velocity constraint.

Figure 4-9: Angular velocities simulated using policies with different control performance.

4-3-4 Simulation 4: $\square\diamond\kappa \wedge \square\diamond\upsilon \wedge \square\neg\psi$

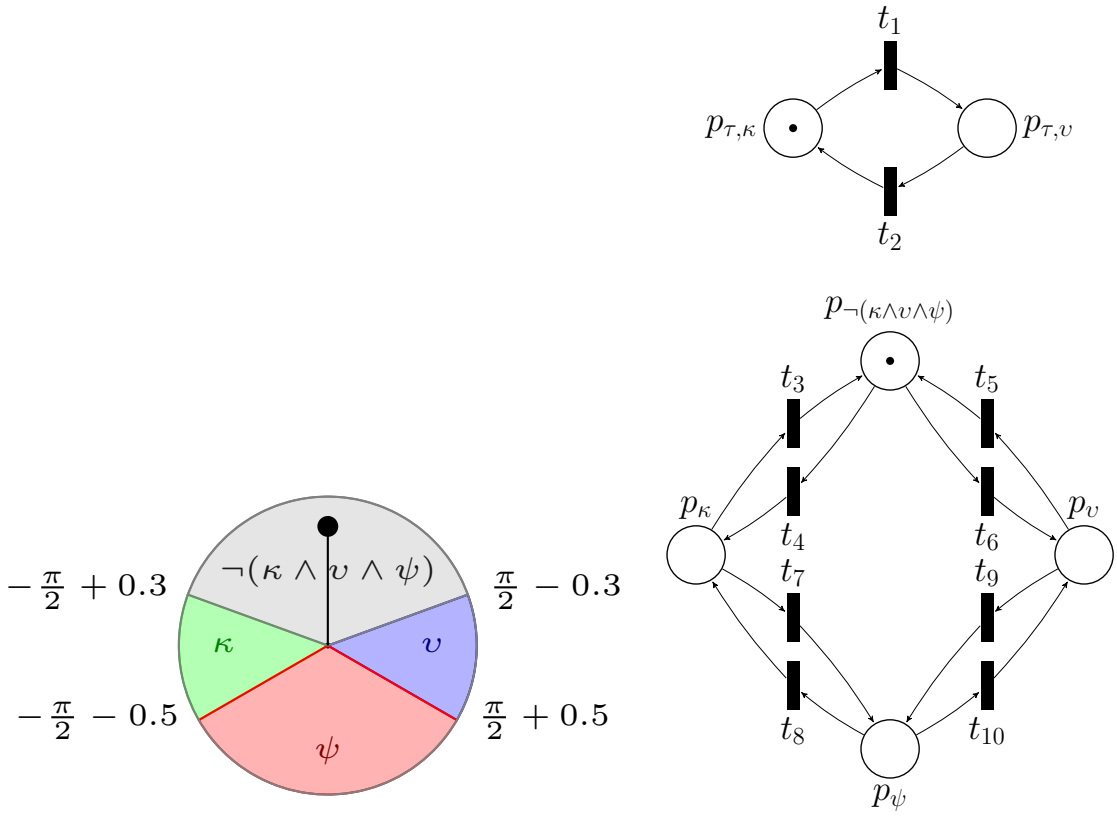
In fourth problem the controller has to learn to repeat visiting two subsets alternately, while avoiding a particular subset. The pendulum should swing to the left and right side alternately, while not falling to the downward position (Figure 4-10). The reward function is

$$r_4(M_k, M_{k+1}) = \min\left(5(1+\tilde{p}_\kappa)\tilde{p}_{\tau,\kappa}(p_\kappa - p_{\neg\kappa}), 5(1+\tilde{p}_\upsilon)\tilde{p}_{\tau,\upsilon}(p_\upsilon - p_{\neg\upsilon}), 25(1+\tilde{p}_{\neg\psi})(p_{\neg\psi} - p_\psi)\right). \tag{4-17}$$

If there is a token in $p_{\tau,\kappa}$, the pendulum has to swing to the left side of the circle, thus κ should become true. When κ becomes true, t_2 will fire such that the reward corresponding to reaching the right half side will become active. After the pendulum swings to the right side and υ becomes true, t_1 will fire and again the controller has to swing the pendulum to left side.

Result

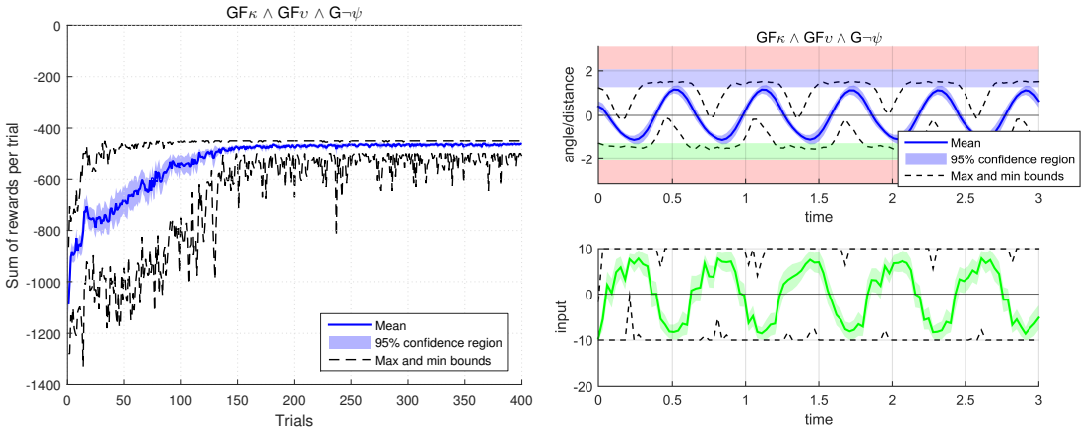
As Figure 4-11 shows, the learning curve converges at negative value in contrast with previous problems. This is due to the fact that the pendulum should swing to two points alternately. Positive reward is only given at the time step when the region is visited corresponding to the active timing related place, i.e. if κ becomes true when $p_{\tau,\kappa} = 1$ and if υ becomes true when $p_{\tau,\upsilon} = 1$. From the next time step it only receives negative rewards since the pendulum should already swing to the opposite side. The controller is thus always forced to reach different subset, which means that it is almost always penalized. Therefore the cumulative reward becomes negative.



(a) The control task is to swing the pendulum to κ and υ alternately, while avoiding ψ. Again gray region is the set of initial states.

(b) The reward Petri net for $\square\diamond\kappa\wedge\square\diamond\upsilon\wedge\square\neg\psi$.

Figure 4-10: Picture describing the problem 4 and the corresponding reward Petri net.



(a) Averaged Learning curve over 30 runs.

(b) Transient simulation.

Figure 4-11: Results of learning problem 4.

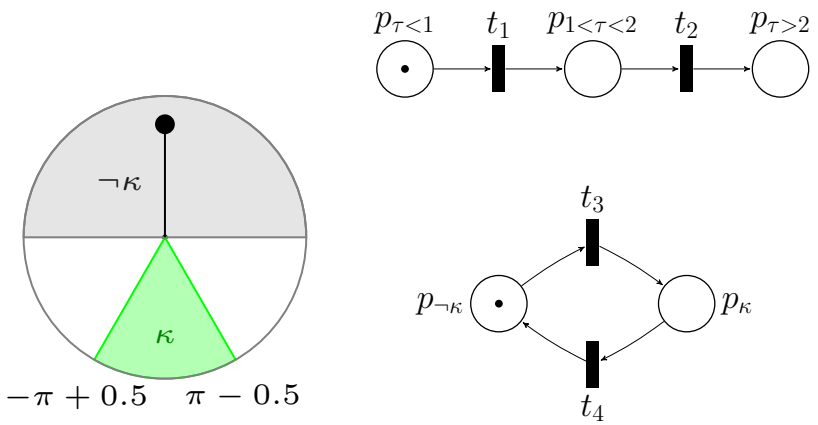
4-3-5 Simulation 5: $\neg\kappa \; \mathcal{U}_{\{1,2\}} \; \left(\kappa \; \mathcal{U}_{\{2,\infty\}} \; \Box\neg\kappa\right)$

The final control task uses temporal operators the time interval, whereby the timing of visiting a certain set of state becomes important. Here, the set of initial states is the upper half plane of the circle (Figure 4-12). Before 1 second elapses, the controller is forbidden to go to downward position. After 1 second is elapsed, the pendulum should stay at the downward position and again after 2 seconds the pendulum should leave the downward position. Since without control the pendulum falls to downward position immediately, the controller should learn to keep the pendulum outside the asked range first second. The transitions t_1 and t_2 will fire according to the elapsed time τ. The purpose of this experiment is to check whether the proposed method can deal with Petri net whereby the transition is only in one way, since there is no way to visit the previous state again.

The reward function is

$$r_5(M_k, M_{k+1}) = 2p_{\tau<1}(p_{\neg\kappa} - p_\kappa) + 3p_{1<\tau<2}(p_\kappa - p_{\neg\kappa}) + p_{\tau>2}(1 + \tilde{p}_{\neg\kappa})(p_{\neg\kappa} - p_\kappa) \quad (4\text{-}18)$$

and again the costs are found heuristically. This reward function could be interpreted as a time varying reward function, since every time only one of the three terms is active.



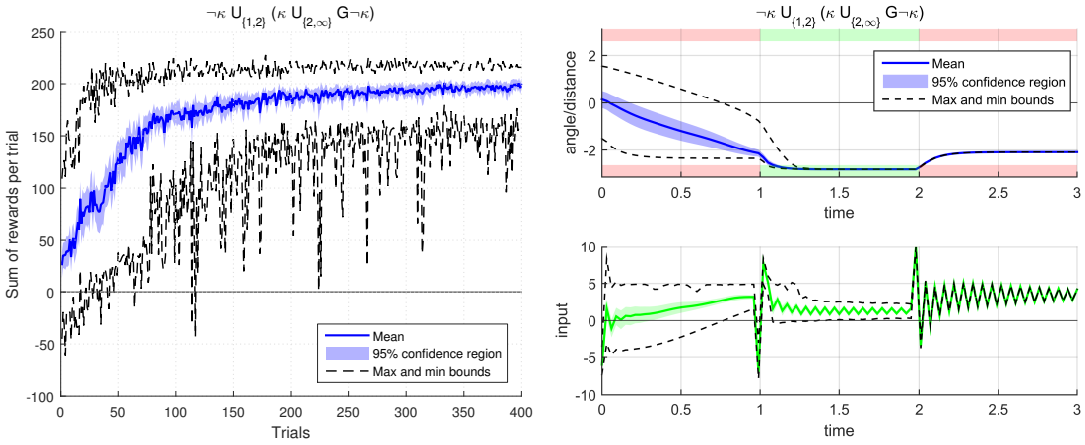
(a) Depending on the elapsed time, the pendulum should enter or leave κ. Again gray region is the set of initial states.

(b) The reward Petri net for $\neg\kappa \; \mathcal{U}_{\{1,2\}} \left(\kappa \; \mathcal{U}_{\{2,\infty\}} \; \Box\neg\kappa\right)$. The transitions t_1 and t_2 fires according to the elapsed time.

Figure 4-12: Picture describing the problem 5 and the corresponding reward Petri net.

Result

Figure 4-13 shows the averaged learning curve and the transient simulation using the learned policy. As given in the control performance, the pendulum swings downwards after 1 second and stay there until one more second elapses. In time $[0, 1]$ and $[2, 3]$ we can see that the pendulum keeps still some distance from the border of defined region. This is due to the exploration noise: the controller takes enough margin such that the desired performance is still achieved even when there is noise.



(a) Averaged Learning curve over 30 runs.

(b) Transient simulation.

Figure 4-13: Results of learning problem 5.

4-4 Discussion

In this chapter we have discussed how to design a learning controller which fulfills the control performance given in temporal logic. Thereby the key factor is how to convert the temporal formulae to a reward system. We have proposed to convert the temporal formulae into a Petri net, such that the reward function becomes a function of state of the Petri net. Converting the temporal formulae into a discrete model is often used in computer science, and there exist also works synthesizing controllers for a MDP with temporal logic constraints [20], [21]. [20] is closer to our work: they convert the linear temporal logic specification into a Rabin automaton and gives positive and negative rewards according to the given acceptance condition. To obtain the policy, TD-learning is used to the new augmented MDP. Contradictory to their work, our proposed method deals with continuous states and Petri net is used instead of a Rabin automaton. There are few reasons why Petri nets are preferable than automata. First, the states of Petri net can be given in a convenient way i.e. by marking vector which represents the number of tokens in each place. Presenting multiple active conditions can also be done conveniently. The reward function could be easily given as a function of number of tokens in each place (condition). Second, the expansion of Petri net could be done easily by setting up Petri nets in parallel, which means that we can easily add new control performance to the existing one. Problem 2 to 3 in simulation section is an example of adding a new control specification. And finally, each place can contain multiple tokens. However we did not have Petri nets where multiple tokens live in a place, this can be useful for rewarding. As an example, we can create a place which counts how many times the specification is violated.

The reward functions given in 4-3 are all relatively simple reward functions, which rewards by fulfilling and penalizes by violating given conditions. Thus when we have complex problems with large state spaces, it might be that reward function is informing too less whereby the learning becomes slow or even impossible. Though in such kind

of cases, we might be able to pump in information, while keeping the same structure of reward function, only by rearranging the true false condition of the atomic proposition. As an example, imagine a learning task to play reversi. The control performance is $\Diamond\Box\varphi$ whereby φ is defined as "true if the agent wins the game". Obviously, it will take long time until the controller learns winning the game. However by redefining φ as "true if there is more agent's color than the opponent's", the learning might speed up. The main task remains the same since winning the game means there is more agent's color on the board than the opponent's. In the latter case the agent receives far more information which reinforces the agent's learning. So we can hid useful information in Petri nets such that learning becomes easier, while the reward function remains the same.

In the proposed method, designing reward system consists of two crucial steps: converting the control performance written in temporal logic to Petri nets and converting the control performance to a reward function. The reward function defines as usual what the controller should learn: "good" things are rewarded and "bad" things are penalized. To let the controller know what "good" and "bad" is for us, we design the Petri net. We are using the Petri net as a tool to inform the controller about our assumptions.

The designed reward functions satisfies most conditions of a proper reward function defined in introduction (Chapter 1). The learned controller fulfills the control performance and it is also structured naturally in narrow time perspective, since the rewards are only given when the specification is fulfilled. However, achieving the control performance still depends on the weighting factors and it cannot be said that the reward function is structured truly natural. Once the specification is violated, there is need to give permanent penalty, such that the maximum cumulative reward will not be achieved. We had shown one way of how the temporal formula might be converted to a Petri net and reward function, which could be said as structured way of designing reward function. There remains the question whether the reward function enables learning in reasonable time, however this might be improved by editing the atomic sentence as discussed above.

Chapter 5

Conclusions and recommendations

In this thesis, two methods are proposed to learn controllers fulfilling two types of control performance using Reinforcement Learning (RL). This chapter summarizes the findings of this thesis and lists the recommendations for future work.

5-1 Conclusions

The goal of this thesis was to synthesize a design methodology of a proper reward function which can be used in RL, such that the learned control law will fulfill the given control performance. Stated differently, the goal was to translate the control performance into a proper reward function which satisfies several conditions, introduced in chapter 1. Thereby the focus was put on classical control performance and temporal logic specification, discussed in chapters 3 and 4 respectively. In both cases a discrete model is used which makes the process more suitable for rewarding. Thereby the arranged version of Standard Actor-Critic (SAC) is used to run the whole system including the discrete model. The most important conclusions from simulation experiments is given for the proposed methods.

5-1-1 Reinforcement Learning for classical control performance

- Using Hybrid Reinforcement Learning (HyRL), a controller was learned which fulfills classical control criteria of overshoot and rise time, validated using a mass-damper system. By modeling the process using an automaton, the total problem is divided into multiple simpler problems and therefore the reward function assigned for each mode could be kept simple. HyRL enabled to learn different policy for each mode defined by the hybrid automaton.
- By reusing the modes with their corresponding policies, new automaton could be created with a new control law. Therefore it is possible to learn the modes for

fulfilling rise time and overshoot performances separately and later combine them to obtain a control law fulfilling both.

5-1-2 Reinforcement Learning for temporal logic specification

- The Temporal Logic (TL) specification was converted into a set of petri net and reward function, whereby the reward function was a function of marking from the petri net. Applying Standard Actor-Critic with Petri Net (SACPN), which is an arranged version of Standard Actor-Critic (SAC), a control law was learned satisfying the given TL specification. Thereby the petri net modeled several conditions of the problem and the desired conditions were then rewarded by the reward function.
- The conversion of TL into a reward system was defined, which is validated on five different control performances. For all the simulation experiments, the obtained results were satisfactory.
- Although the reward system had shown good results, the obtained control law still depends on the chosen weighting factors, which has to be chosen heuristically. The reward system should contain information how often the given specification was violated, such that the resulting control law fulfills the specification independent from the weighting factors.
- By designing the reward system as a set of process model and the reward function, it became possible to keep the reward function simple. Complex state and time relations were all hidden by the petri net.
- There is availability to provide more information to the controller by changing the atomic sentence, while the structure of the reward function remains the same.

5-2 Recommendations

- In this work HyRL is used with as purpose to learn controller satisfying classical control performance. The classical control performance is limited since it can only be used for evaluating stabilization and a part of reference tracking problems. It is recommended in future research to apply the method to a more complex control performance which has more power of represent the performance. It might be also interesting to apply the method on a hybrid automaton as a learning algorithm for hybrid system.
- Since the system was augmented by initial state and time, computing radial basis functions costs time. Using other approximators as tile coding may decrease the computation time.
- In this thesis, the SACPN was only applied to the pendulum setup. Since the pendulum is in one dimension, the desired movements were somehow fixed. It is

recommended for future research to apply SACPN on a more complex system, such that the controller find the truly optimal policy from several policies fulfilling the same control performance. Imagine thereby an goal reaching obstacle avoidance problem ($\Diamond\Box \text{goal} \wedge \Box\neg \text{obstacle}$) in two dimension. There will be multiple trajectories that would fulfill the control specification, however by tuning the weighting factors for each term, it might be able to learn controllers with different behavior. For example by weighting the latter term heavier, a prudence controller might be learned which takes a large margin to avoid the obstacle for sure. Conversely by weighting the first term heavier, a controller might be synthesized which looks for the shortest path even it might collide to the obstacle in an irregular situation.

- In this thesis, the reward functions are designed with the expectation that by giving reward according to the given specification at each time step, the obtained reward function will lead to achieve the given control specification. However, only by inspecting whether the specification is fulfilled instantaneously, the learning could lead to undesired result as like in simulation example 2 in chapter 4 where the controller had "compensated" the penalty in earlier stage by rewards in later stage of the run. The reward function should contain information about violated specifications such that the controller cannot maximize the cumulative amount of reward by violation.

- However SACPN had worked in simulation experiments in this thesis, there is no guaranteed convergence. Finding a theoretical convergence guarantee of the algorithm will be a necessary work.

- There is also no guarantee that proposed way of assigning reward function, always leads to a desired controller. To our knowledge, there is still no work done which gives theoretical guarantees of a particular control performance by use of corresponding reward function. It is even unknown whether such guarantee can be given theoretically. Finding this guarantee will be an innovative step to fully understand computational RL.

5-3 Final words

As the title suggests, in this thesis the reward systems are designed for two types of control performance. The reward function kept simple to avoid unusable control laws by combining the reward function with discrete models expressing the process in a convenient way for rewarding, which forms together the reward system. Specifically the conversion of Metric Interval Temporal Logic to a reward system had shown satisfactory results in simulations where the controller indeed learned control law which fulfilled the given control performance. However, there is no theoretical guarantee that this way of designing reward system always leads to a control law fulfilling the MITL performance. The obtained control law is still dependent on the weighting parameters, while it is preferable that the reward function leads to achieving control performance independent of them. Also the reward system design for more complex systems are left open for

future research. To conclude, in this thesis a step has been taken to design a proper reward function for fulfilling desired control performance, with the hope of some day being able to let the controllers learn truly desired behaviors.

Appendix A

Simulation models

This appendix describes the two models used for simulation experiments. The first simulation setup is the mass-damper model used in chapter 3. The second setup is the pendulum model used in chapter 4.

A-1 Mass-damper system

The mass-damper system consists of a single mass and it will move in one dimensional direction by applying force on it. The friction of the mass ensures that the system is always stable. Figure A-1 is the picture of the system.

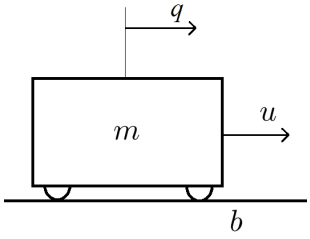


Figure A-1: A mass-damper system.

The equation of the motion of this system is

$$M\ddot{q} = -B\dot{q} + u \tag{A-1}$$

where M and B are mass and damper constant relatively. Table A-1 shows the used system parameters.

Table A-1

Model parameter	Symbol	Value	Unit
Mass	M	0.5	kg
Damping	B	0.05	N s/m

A-2 Pendulum

The pendulum setup consists of a weightless link with a mass attached on the end of it. The other side of the link is connected to the electro motor, which actuates the link with mass. A picture of this system is shown in Figure A-2.

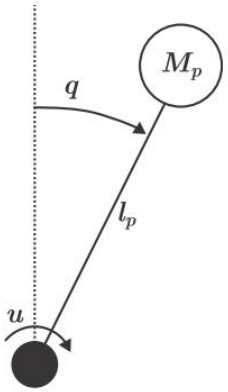


Figure A-2: The pendulum setup.

The equation of motion of pendulum is

$$J\ddot{q} = M_p g l_p \sin(q) - \left(b + \frac{K^2}{R}\right)\dot{q} + \frac{K}{R}u \tag{A-2}$$

where q is the angle of the pendulum measured from the upright position. The model parameters are given in Table A-2.

Table A-2

Model parameter	Symbol	Value	Unit
Pendulum inertia	J	$1.91 \cdot 10^{-4}$	kg m^2
Pendulum mass	M_p	$5.50 \cdot 10^{-2}$	kg
Gravity	g	9.81	m/s^2
Pendulum length	l_p	$4.20 \cdot 10^{-2}$	m
Damping	b	$3 \cdot 10^{-6}$	N m s/rad
Torque constant	K	$5.36 \cdot 10^{-2}$	N m/A
Rotor resistance	R	9.50	Ω

Appendix B

Hybrid automaton models

In this appendix the transition conditions of the hybrid automatons used in chapter 3 is given. The figure of the petri net and the table of mode-wise reward functions is added for the convenience and there is no difference with those in the chapter.

B-1 Hybrid automaton for overshoot performance

The mass-damper system evolves according to the equation of motion given in appendix A-1 in all modes. The memory state evolves as

$$z_{k+1} = z_k = q_0.$$

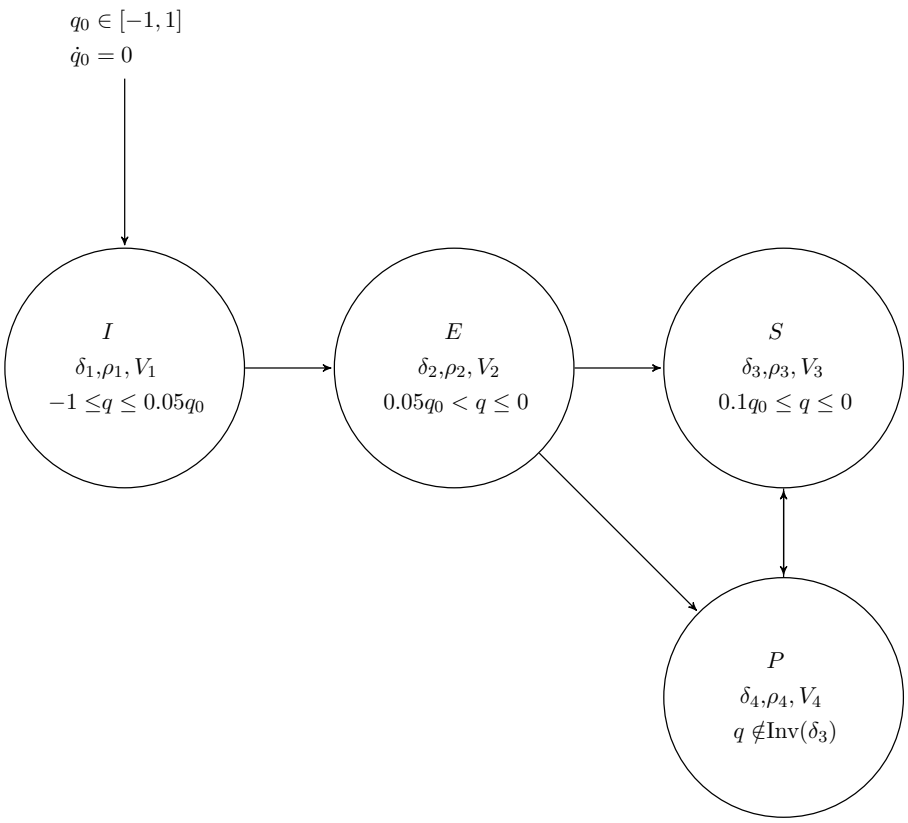


Figure B-1: Hybrid automaton for over-/undershoot performance $M_p = 0$.

$$
\begin{aligned}
\mathrm{Init} &= \{(q_0, \dot{q}_0) \mid q_0 \in [-1, 1] \wedge \dot{q}_0 = 0\} \\
\mathrm{Inv}(\delta_1) &= \begin{cases} \{(q, \dot{q}) \mid q \leq 0.05q_0\} \text{ if } q_0 < 0 \\ \{(q, \dot{q}) \mid q \geq 0.05q_0\} \text{ if } q_0 \geq 0 \end{cases} \\
\mathrm{Inv}(\delta_2) &= \begin{cases} \{(q, \dot{q}) \mid 0.05q_0 < q \leq 0\} \text{ if } \wedge\, q_0 < 0 \\ \{(q, \dot{q}) \mid 0.05q_0 > q \geq 0\} \text{ if } q_0 \geq 0 \end{cases} \\
\mathrm{Inv}(\delta_3) &= \begin{cases} \{(q, \dot{q}) \mid 0.1q_0 \leq q \leq 0\} \text{ if } q_0 < 0 \\ \{(q, \dot{q}) \mid 0.1q_0 \geq q \geq 0\} \text{ if } q_0 \geq 0 \end{cases} \\
\mathrm{Inv}(\delta_4) &= \begin{cases} \{(q, \dot{q}) \mid q < 0.05q_0 \wedge q > 0\} \text{ if } q_0 < 0 \\ \{(q, \dot{q}) \mid q > 0.05q_0 \wedge q < 0\} \text{ if } q_0 \geq 0 \end{cases}
\end{aligned}
\tag{B-1}
$$

Table B-1: Table explaining when transitions occur in the hybrid automaton for overshoot performance.

transition	transition condition
mode I $\rightarrow$ mode E	$(q > 0.05q_0 \mid q_0 < 0) \vee (q < 0.05q_0 \mid q_0 > 0)$
mode E $\rightarrow$ mode S	$(\dot{q} < 0 \mid q_0 < 0) \vee (\dot{q} > 0 \mid q_0 > 0)$
mode E $\rightarrow$ mode P	$(q > 0 \mid q_0 < 0) \vee (q < 0 \mid q_0 > 0)$
mode S $\rightarrow$ mode P	$(q > 0 \vee q < 0.1q_0 \mid q_0 < 0) \vee (q < 0 \vee q > 0.1q_0 \mid q_0 > 0)$
mode P $\rightarrow$ mode S	$(0.1q_0 < q < 0 \mid q_0 < 0) \vee (0 < q < 0.1q_0 \mid q_0 > 0)$

Table B-2: Mode-wise reward functions for mass-damper stabilization with overshoot performance $M_p = 0$.

Mode	Reward function
I	$\rho_1 = \begin{cases} 0 & \text{if } \dot{q} > 0 \mid q_0 < q_{\text{des}} \text{ (overshoot)} \\ 0 & \text{if } \dot{q} < 0 \mid q_0 > q_{\text{des}} \text{ (undershoot)} \\ -3 & \text{else} \end{cases}$
E	$\rho_2 = -3\lvert q_{\text{des}} - q \rvert$
S	$\rho_3 = -3\lvert q_{\text{des}} - q \rvert + \begin{cases} 3 & \text{if } \delta = \delta' = \text{S} \\ 0 & \text{else} \end{cases}$
P	$\rho_4 = -9$

B-2 Hybrid automaton for rise time performance

Again, for mass-damper system evolves according to the equation of motion. Thereby the memory state evolves in mode E as

$$z_{k+1} = \begin{bmatrix} q_0 \\ t_{k+1} \end{bmatrix} = \begin{bmatrix} q_0 \\ t_k + T_s \end{bmatrix}$$

whereby T_s is the sample time. For other modes, time will not evolve and the memory state becomes

$$z_{k+1} = \begin{bmatrix} q_0 \\ t_{k+1} \end{bmatrix} = \begin{bmatrix} q_0 \\ t_k \end{bmatrix}$$

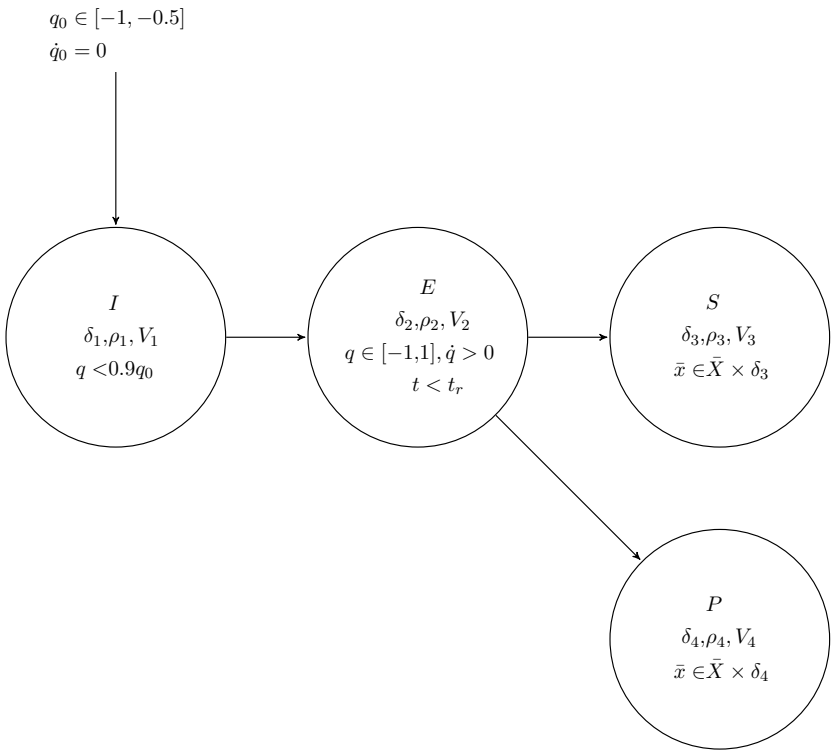


Figure B-2: Hybrid automaton for rise time performance $t_r = [0.6, 0.8]$.

$$
\begin{aligned}
\text{Init} &= \{(q_0, \dot{q}_0) \mid q_0 \in [-1, -0.5] \wedge \dot{q}_0 = 0\} \\
\text{Inv}(\delta_1) &= \{(q, \dot{q}) \mid q < 0.9q_0\} \\
\text{Inv}(\delta_2) &= \{(q, \dot{q}) \mid 0.9q_0 < q \leq 0.1q_0 \wedge t < 0.8\} \\
\text{Inv}(\delta_3) &= \bar{X} \times \delta_3 \\
\text{Inv}(\delta_4) &= \bar{X} \times \delta_4
\end{aligned}
\tag{B-2}
$$

Table B-3: Table explaining when transitions occur in the hybrid automaton for rise time performance.

transition	transition condition
mode I $\rightarrow$ mode E	$q > 0.9q_0$
mode E $\rightarrow$ mode S	$(q > 0.1q_0 \mid 0.6 < t < 0.8)$
mode E $\rightarrow$ mode P	$\dot{q} < 0 \vee (q > 0.1q_0 \mid t < 0.6) \vee (q < 0.1q_0 \mid t > 0.8)$

Table B-4: Mode-wise reward functions for mass-damper stabilization with rise time performance $t_r = 0.8$.

Mode	Reward function
I	$\rho_1 = \begin{cases} 0 & \text{if } \dot{q} > 0 \mid q_0 < q_{\text{des}} \\ 0 & \text{if } \dot{q} < 0 \mid q_0 > q_{\text{des}} \\ -3 & \text{else} \end{cases}$
E	$\rho_2 = -3(0.1q_{\text{des}} - q)$
S	$\rho_3 = -3\lvert q_{\text{des}} - q\rvert + \begin{cases} 3 & \text{if } \delta = \delta' = \text{S} \\ 0 & \text{else} \end{cases}$
P	$\rho_4 = -9$

B-3 Hybrid automaton for overshoot and rise time performance

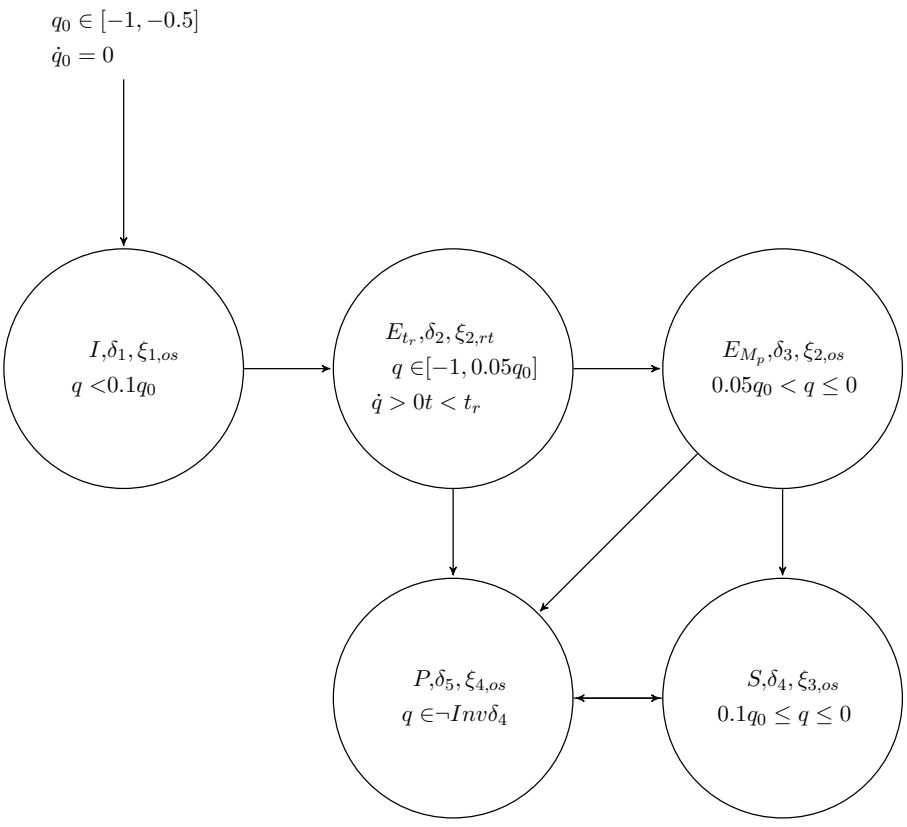


Figure B-3: Hybrid automaton for rise time performance $t_r \in [0.6, 0.8]$ and $M_p = 0$.

$$\begin{aligned}
\text{Init} &= \{(q_0, \dot{q}_0) \mid q_0 \in [-1, -0.5] \wedge \dot{q}_0 = 0\} \\
\text{Inv}(\delta_1) &= \{(q, \dot{q}) \mid q < 0.9q_0\} \\
\text{Inv}(\delta_2) &= \{(q, \dot{q}) \mid 0.9q_0 < q \leq 0.1q_0 \wedge t < 0.8\} \\
\text{Inv}(\delta_3) &= \{(q, \dot{q}) \mid 0.1q_0 \leq q \leq 0\} \\
\text{Inv}(\delta_4) &= \{(q, \dot{q}) \mid 0.1q_0 \leq q \leq 0\} \\
\text{Inv}(\delta_5) &= \{(q, \dot{q}) \mid q < 0.05q_0 \wedge q > 0\}
\end{aligned} \tag{B-3}$$

Table B-5: Table explaining when transitions occur in the hybrid automaton for rise time and overshoot performance.

transition	transition condition
mode I $\rightarrow$ mode E_{t_r}	$q > 0.9q_0$
mode E_{t_r} $\rightarrow$ mode E_{M_p}	$(q > 0.1q_0 \mid 0.6 < t < 0.8)$
mode E_{t_r} $\rightarrow$ mode P	$\dot{q} < 0 \vee (q > 0.1q_0 \mid t < 0.6) \vee (q < 0.1q_0 \mid t > 0.8)$
mode E_{M_p} $\rightarrow$ mode S	$\dot{q} < 0$
mode E_{M_p} $\rightarrow$ mode P	$q > 0$
mode S $\rightarrow$ mode P	$(q > 0 \vee q < 0.1q_0 \mid q_0 < 0) \vee (q < 0 \vee q > 0.1q_0 \mid q_0 > 0)$
mode P $\rightarrow$ mode S	$(0.1q_0 < q < 0 \mid q_0 < 0) \vee (0 < q < 0.1q_0 \mid q_0 > 0)$

Appendix C

Petri net models

In this appendix the composite change matrix and the transition conditions of the petri nets used in simulation experiments of chapter 4 are denoted. The figure of the petri net is added for the convenience and there is no difference with the figures in the chapter. The marking of the petri net represents the initial marking used in the problems.

Mention for the transition conditions, it is under assumption that the transition is enabled, i.e. the input place contains a token. Given that the transition is enabled, t becomes 1 if the the written condition is true and otherwise t is 0.

C-1 Problem 1: $\Diamond\Box\kappa$

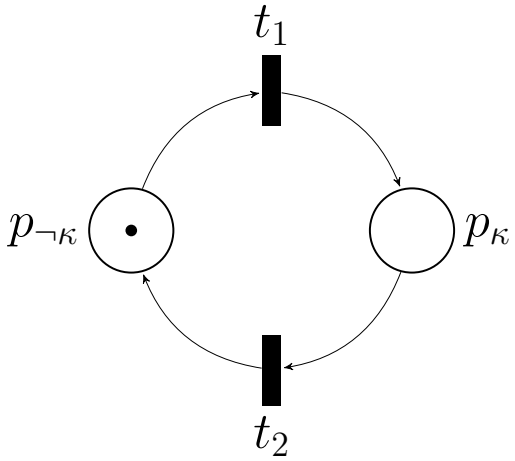


Figure C-1: Petri net for $\Diamond\Box\kappa$

$$\begin{aligned} \kappa &= -0.1 < q < 0.1 \\ M &= \begin{bmatrix} p_\kappa & p_{\neg\kappa} \end{bmatrix} \end{aligned} \tag{C-1}$$

$$D = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}^T = \begin{cases} \kappa \\ \neg\kappa \end{cases}$$

C-2 Problem 2: $\Diamond\Box\kappa \wedge \Box\neg\psi$

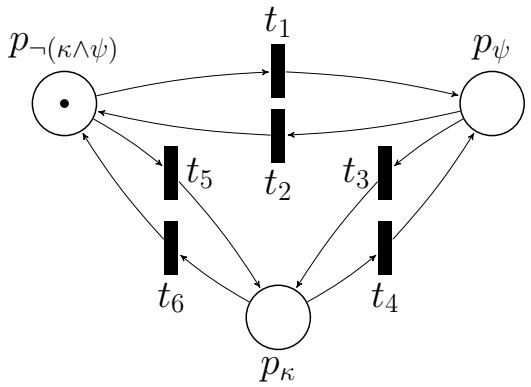


Figure C-2: Petri net for $\Diamond\Box\kappa \wedge \Box\neg\psi$

$$\begin{aligned} \kappa &= -0.3 < q < 0.3 \\ \psi &= 0.3 < q < \frac{\pi}{2} + 0.5 \\ M &= \begin{bmatrix} p_{\neg(\kappa\wedge\psi)} & p_{\neg\psi} & p_\kappa \end{bmatrix} \end{aligned}$$

$$D = \begin{bmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & -1 & 1 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{bmatrix}^T = \begin{cases} \psi \\ \neg(\kappa \wedge \psi) \\ \kappa \\ \psi \\ \kappa \\ \neg(\kappa \wedge \psi) \end{cases}$$

C-3 Problem 3: $\Diamond\Box\kappa \wedge \Box\neg\psi \wedge \Box\upsilon$

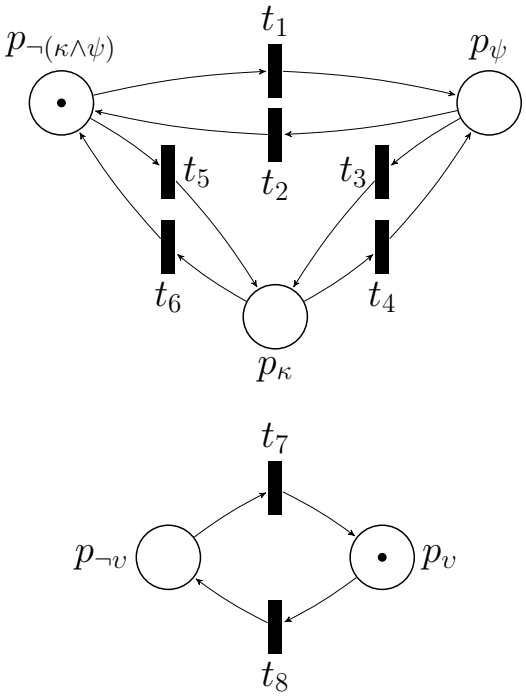


Figure C-3: Petri net for $\Diamond\Box\kappa \wedge \Box\neg\psi \wedge \Box\upsilon$

$$
\begin{aligned}
\kappa &= -0.3 < q < 0.3 \\
\psi &= 0.3 < q < \frac{\pi}{2} + 0.5 \\
\upsilon &= \mid \dot{q} \mid < 10 \\
M &= \begin{bmatrix} p_{\neg(\kappa\wedge\psi)} & p_{\neg\psi} & p_\kappa & p_{\neg\upsilon} & p_\upsilon \end{bmatrix}
\end{aligned}
$$

$$
D = \begin{bmatrix}
-1 & 1 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 0 \\
0 & -1 & 1 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 \\
1 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & -1 & 1 \\
0 & 0 & 0 & 1 & -1
\end{bmatrix}
\qquad
\begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{bmatrix}^T
= \begin{cases}
\psi \\
\neg(\kappa \wedge \psi) \\
\kappa \\
\psi \\
\kappa \\
\neg(\kappa \wedge \psi) \\
\upsilon \\
\neg\upsilon
\end{cases}
$$

C-4 Problem 4: $\square\diamond\kappa \wedge \square\diamond\upsilon \wedge \square\neg\psi$

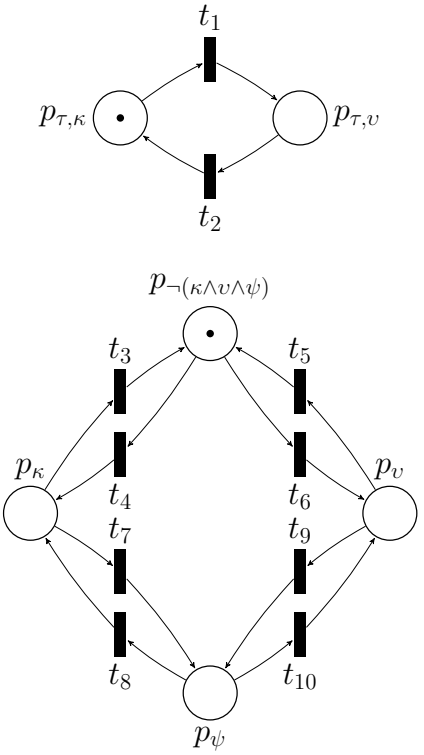


Figure C-4: Petri net for $\square\diamond\kappa \wedge \square\diamond\upsilon \wedge \square\neg\psi$

$$\kappa = -\frac{\pi}{2} - 0.5 < q < -\frac{\pi}{2} + 0.3$$
$$\upsilon = \frac{\pi}{2} - 0.3 < q < \frac{\pi}{2} + 0.5$$
$$\psi = -\pi < q < -\frac{\pi}{2} - 0.5 \vee \frac{\pi}{2} + 0.5 < q < \pi$$
$$M = \begin{bmatrix} p_{\tau,\kappa} & p_{\tau,\upsilon} & p_\kappa & p_\upsilon & p_\psi & p_{\neg(\kappa\wedge\upsilon\wedge\psi)} \end{bmatrix}$$

$$D = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \end{bmatrix} \qquad \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \\ t_9 \\ t_{10} \end{bmatrix}^T = \begin{cases} \kappa \\ \upsilon \\ \neg(\kappa \wedge \upsilon \wedge \psi) \\ \kappa \\ \neg(\kappa \wedge \upsilon \wedge \psi) \\ \upsilon \\ \psi \\ \kappa \\ \psi \\ \upsilon \end{cases}$$

C-5 Problem 5: $\neg\kappa\, \mathcal{U}_{\{1,2\}}\ (\kappa\, \mathcal{U}_{\{2,\text{inf}\}}\ \Box\neg\kappa)$

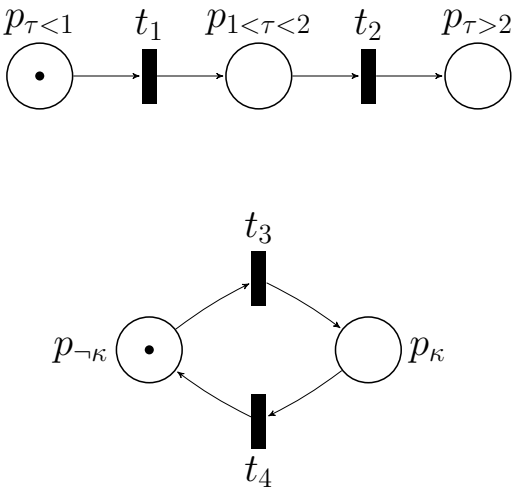


Figure C-5: Petri net for $\neg\kappa\, \mathcal{U}_{\{1,2\}}\ (\kappa\, \mathcal{U}_{\{2,\text{inf}\}}\ \Box\neg\kappa)$

$$\kappa = -\pi < q < -\pi + 0.5 \vee \pi - 0.5 < q < \pi$$
$$M = \begin{bmatrix} p_{\tau<1} & p_{1<\tau<2} & p_{\tau>2} & p_{\neg\kappa} & p_\kappa \end{bmatrix}$$

$$D = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \qquad \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{bmatrix}^T = \begin{cases} \tau > 1 \\ \tau > 2 \\ \kappa \\ \neg\kappa \end{cases}$$

Bibliography

[1] L. Buşoniu, "Reinforcement learning and dynamic programming for control - lecture notes." Master ICAF, 2012.

[2] I. Grondman, L. Busoniu, G. Lopes, R. Babuska, *et al.*, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 2012.

[3] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series, Mit Press, 1998.

[4] E. Schuitema, *Reinforcement learning on autonomous humanoid robots*. TU Delft, Delft University of Technology, 2012.

[5] J. Randløv and P. Alstrøm, "Learning to drive a bicycle using reinforcement learning and shaping," 1998.

[6] S. B. Thrun, "Efficient exploration in reinforcement learning," tech. rep., 1992.

[7] M. Wiering, "Explorations in efficient reinforcement learning," *1999*, February 1999.

[8] L. Busoniu, D. Ernst, B. de Schutter, and R. Babuska, "Approximate reinforcement learning: An overview," *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pp. 1–8, May 2011.

[9] R. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, February 1988.

[10] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *SIAM Journal on Control and Optimization*, pp. 1008–1014, MIT Press, 2000.

[11] M. Riedmiller, J. Peters, and S. Schaal, "Evaluation of policy gradient methods and variants on the cart-pole benchmark," in *Approximate Dynamic Programming*

and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on, pp. 254 –261, april 2007.

[12] M. Mataric *et al.*, "Reward functions for accelerated learning," in *Proceedings of the Eleventh International Conference on Machine Learning*, vol. 189, San Francisco, 1994.

[13] V. Gullapalli, *Reinforcement learning and its application to control*. PhD thesis, Citeseer, 1992.

[14] A. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," *MACHINE LEARNING-*, 1999.

[15] A. Ng and S. Russell, "Algorithms for inverse reinforcement learning," *... international conference on machine learning*, 2000.

[16] P. Abbeel and A. Ng, "Apprenticeship learning via inverse reinforcement learning," *...-first international conference on Machine learning*, 2004.

[17] R. S. Sutton, D. Precup, S. Singh, *et al.*, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1, pp. 181–211, 1999.

[18] D. Precup, "Temporal abstraction in reinforcement learning," 2000.

[19] J. Morimoto and K. Doya, "Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning," *Robotics and Autonomous Systems*, vol. 36, no. 1, pp. 37–51, 2001.

[20] D. Sadigh, E. S. Kim, S. Coogan, S. S. Sastry, S. Seshia, *et al.*, "A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications," in *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pp. 1091–1096, IEEE, 2014.

[21] M. Svorenova, I. Cerna, and C. Belta, "Optimal control of mdps with temporal logic constraints," in *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*, pp. 3938–3943, IEEE, 2013.

Glossary

List of Acronyms

RL	Reinforcement Learning
MDP	Markov Decision Process
AC	Actor-Critic
SAC	Standard Actor-Critic
DP	Dynamic Programming
TD	Temporal Difference
TL	Temporal Logic
MITL	Metric Interval Temporal Logic
IRL	Inverse Reinforcement Learning
PID	Proportional-Integral-Derivative
HyRL	Hybrid Reinforcement Learning
HRL	Hierarchical Reinforcement Learning
SACPN	Standard Actor-Critic with Petri Net

List of Symbols

General

$\hat{a}$	approximation of a variable or function a
a^T	transpose of vector/matrix a
$\mathbb{R}$	set of real numbers

$\mathcal{N}$	normal distribution
$\sim$	draw from distribution
σ	standard deviation
$\nabla_a f$	gradient of function f with respect to a

Reinforcement Learning

e	eligibility trace
f	transition function
k	discrete time
r	instantaneous reward
u	action
x	state
J	cost function
Q	state-action value function
R	return
U	action space
V	state value function
X	state space
α	learning rate
δ	temporal difference (error)
γ	discount factor
λ	eligibility trace decay rate
ϕ	basis function
π	policy
ρ	reward function
θ	value function parameter vector
ξ	policy parameter vector
Δu	exploration

Control performance languages

c	constant for discrete reward function
C	cost matrix for absolute error reward function
Q	cost matrix for quadratic error reward function

e_{ss}	steady-state error
t_r	rise time
t_s	settling time
M_p	overshoot ratio

a	atomic proposition
AP	set of atomic propositions
I	time interval
$\Diamond$	eventually, **F**
$\Box$	globally, **G**
$\neg$	negation
$\top$	true
$\mathcal{U}$	until, **U**
φ	temporal formula
$\wedge$	conjunction

(Hybrid) Automata

f	continuous state transition function
D	set of modes (discrete states)
E	set of edges
F	discrete state transition function
G	guard condition
H	hybrid automaton
$P(X)$	power set of X
R	reset map
U	set of inputs
X	set of continuous states
Init	set of initial states
Inv	invariant set

Petri net

p	place/number of tokens in place
t	transition
A	set of arcs
D	composite change matrix
D^+	composite change matrix describing outputs
D^-	composite change matrix describing inputs
M	marking matrix
P	set of places
T	set of transitions

Hybrid Reinforcement Learning

$\bar{x}$	augmented state
m	memory state transition function

t	time
z	memory state
T	transition relation
δ	mode
η	switching function
ϱ	set of reward functions
Γ	transition function of hybrid automaton

Standard Actor-Critic with Peri Net

$\tilde{p}$	number of tokens in place at previous time step
r	reward function
$\bar{x}$	augmented state
τ	time
κ, υ, ψ	atomic propositions
$\varkappa, \Upsilon, \Psi$	subsets of state space
Γ	petri net
Σ	total system augmented by petri net
Θ	value function parameter matrix
Ξ	policy parameter matrix