ECE6102 Final Report

Iman Salehi, Ryan Saltus

Abstract

In this course project, we investigated the reinforcement learning based approach for imposing barrier constraints on a state space as outlined in [9]. This approach transforms system states into a constrained space, and puts constraints on full-state and control input simultaneously. An actor-critic based reinforcement learning technique is combined with the barrier transformation to learn the optimal control policy in the constrained space that also stabilizes the original system in the sense of Lyapunov. In addition to recreating the authors' results, we have implemented several supplementary simultations to further explore the performance of the Actor-Critic-Barrier approach. We have also implemented this approach on a Baxter research robot in order to test the practicality of this method on a physical system.

I. Introduction

IN the real world, solutions to control problems have to be designed in a specific way to take safety constraints into account. In a controls sense, this means that the state space of the system must be bounded, and the control input must be upper bounded. One of the state-of-the-art approaches to ensuring that the state space of a system is bounded is to transform it using a barrier function. The barrier Lyapunov method is a way to deal with control problems that have a constrained state [3], [7]. In [9], the authors formulate the constrained state problem as an optimal control problem, and develop a reinforcement learning algorithm to approximate the optimal policy for a barrier transformed system. In this final project, we implement that approach in simulation, and on an actual robot.

The rest of this report is organized as follows. Section II describes the mathematics involved in this approach, including the derivation of the barrier function transformation, the associated optimal control problem, and the online actor-critic barrier learning method. Section III presents the results of the additional simulations, as well as the robotic experimentation. Section IV presents the final conclusions drawn.

II. Methodology

A. Barrier Function

Consider a continuous nonlinear DS of the following form

$$\begin{aligned} \dot{x}_i(t) &= x_{i+1}(t) \quad \forall i = 1, \cdots, n-1 \\ \dot{x}_n(t) &= f\left(x(t)\right) + g\left(x(t)\right) u, \end{aligned} \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}$ are locally Lipschitz continuous nonlinear functions, $x(t) = [x_1(t) \ \cdots \ x_n(t)] \in \mathcal{X} \subset \mathbb{R}^n$ with $x_i \in \mathbb{R}$ is the state of the system, and $u \in \mathbb{R}$ is the control input. [1]

The objective is to find a control policy $u : \mathbb{R}^n \to \mathbb{R}^m$ such that the closed-system in (1) has an asymptotic stable equilibrium while the control input satisfies

$$\|u\| \leq \lambda, \ \forall i = 1, \cdots, m \tag{2}$$

and the state $x = [x_1 \ \cdots \ x_n]^T$ satisfies

$$\begin{aligned} x_1 &\in (a_1, A_1) \\ &\vdots \\ x_n &\in (a_n, A_n) \end{aligned} \tag{3}$$

Definition 1. *Given the continuous system (1), a real-valued function $B\colon \ \mathcal{X} \to \mathbb{R}$ is said to be barrier Lyapunov function (BLF) for an open set $\mathcal{S}$ containing the origin, that is differentiable with its argument and satisfies the following properties $\forall x \in \mathcal{X}$*

[1] Note that throughout this report, for ease of notation, we abbreviate $x(t)$ by x, unless necessary for clarity.

$$B(x) \leq \gamma, \qquad \lim_{x \to \partial \mathcal{S}} B(x) = \infty, \tag{4}$$

where γ is some positive constant.

The following BLF candidate $B(z,t) : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$ is considered in [9]:

$$B(z; a, A) = \log\left(\frac{A}{a}\frac{a-z}{A-z}\right), \forall z \in (a, A) \tag{5}$$

where $\log(\cdot)$ is a natural logarithm, a and A are lower and upper bounds of each state, and $z \in \mathbb{R}$ is arbitrary input argument that is used as a place holder. Using (5), the system can be transformed into a constrained state $s = [s_i \ \cdots \ s_n] \in \mathbb{R}^n$ as follows

$$s_i = B\left(x_i; a_i, A_i\right), \tag{6}$$

$$x_i = B^{-1}\left(s_i; a_i, A_i\right), \qquad \forall i = 1, \cdots, n \tag{7}$$

where,

$$B^{-1}(s_i; a_i, A_i) = \frac{a_i A_i \left(e^{\frac{-s_i}{2}} - e^{\frac{s_i}{2}}\right)}{A_i e^{\frac{-s_i}{2}} - a_i e^{\frac{s_i}{2}}}. \tag{8}$$

Using the chain rule of differentiation, i.e., $\frac{dx_i}{dt} = \frac{\partial x_i}{\partial s_i}\frac{ds_i}{dt}$, where,

$$\frac{\partial x_i}{\partial s_i} = \frac{A_i a_i^2 - a_i A_i^2}{a_i^2 e^{s_i} - 2a_i A_i + A_i^2 e^{-s_i}} \tag{9}$$

after some algebraic manipulations the new dynamics in s yields

$$\begin{aligned} \dot{s}_i &= \frac{B^{-1}(s_{i+1})}{\frac{\partial x_i}{\partial s_i}} \\ &= F_i\left(s_i, s_{i+1}\right), \forall i = 1, \cdots, n-1 \end{aligned} \tag{10}$$

$$\begin{aligned} \dot{s}_n &= \frac{f(x) + g(x)u}{\frac{\partial x_n}{\partial s_n}} \\ &= F_n(s) + g(s)u, \end{aligned} \tag{11}$$

where,

$$F_n(s) = kf\left(\begin{bmatrix} B^{-1}(s_1) & \cdots & B^{-1}(s_n) \end{bmatrix}\right) \tag{12}$$

$$g(s) = kg\left(\begin{bmatrix} B^{-1}(s_1) & \cdots & B^{-1}(s_n) \end{bmatrix}\right) \tag{13}$$

and

$$k = \frac{\left(a_n^2 e^{s_n} - 2a_n A_n + A_n^2 e^{-s_n}\right)}{A_n a_n^2 - a_n A_n^2} \tag{14}$$

then the constrained system in terms of s can be expressed in a compact form as follows

$$\dot{s} = F(s) + G(s)u, \tag{15}$$

where, $F(s) = \left[F_1\left(s_1, s_2\right), \ \cdots \ , F_n(s)\right]^T$, $G(s) = [0, \ \cdots \ , 0, g(s)]^T$.

Assumption 1. *The function F(s) is Lipschitz continuous with Lipschitz constant L_F and bounded in $\mathcal{S}$ with $\|F(\cdot)\| < \bar{F}$, where $\bar{F}$ is a positive scalar. Moreover, $G(s)$ is also bounded in $\mathcal{X}$, i.e., $\|G(\cdot)\| < \bar{G}$, where $\bar{G}$ is a positive scalar.*

To stabilize the original system (1) with input saturation and full-state constraints, an optimal control problem with constraints of input saturation is formulated for the constrained system defined in (15).

B. Optimal Control Problem

Provided the aforementioned assumptions and Lemma hold, the value function is defined as

$$V(S_0) = \int_{t_0}^{\infty} U(s,u), \tag{16}$$

which is to be minimized with the constraints of input saturation given in (2), where $U(s,u) = Q(s(t)) + \Theta(u(t))$ is the reward function, λ is the bound of control input, $Q(s)$ is a positive definite monotonically increasing function and $\Theta(u)$ is a positive definite integrand function.

Lemma 1 proves that if the initial state is within the prescribed bound, a controller can be learned for the full-state constrained system such that it stabilizes the original system in the sense of Lyapunov (1) given that the initial state of the system $x_0 \in \mathcal{X}$

Lemma 1. *Suppose that u^* solves (16) for the constrained system, then the same controller can also solve stabilizes the original dynamics in (1) provided that the initial state x_0 of the system satisfies the state boundary constraints in (4).*

Proof. See proof of ([7], Lemma 1) □

To deal with the input saturation, the nonquadratic penalty function, i.e., an inverse of a hyperbolic tangent, in [1] is adopted

$$\Theta(u) = 2\int_0^u [\lambda \tanh^{-1}\left(\frac{v}{\lambda}\right)]^T R dv, \tag{17}$$

where $R = \mathrm{diag}([r_1 \ \cdots \ r_m])$ with the positive penalty weight $r_i \in \mathbb{R}^+, \ \forall i \in 1, \cdots, m$.

Definition 2. *A control policy $\mu(s)$ is said to be admissible with respect to the reward function $U(s,u)$ defined in (16) on $\mathcal{X} \in \mathbb{R}^n$, denoted by $\mu(s) \in \pi(\mathcal{X})$, if*

- *$\mu(s)$ is continuous on $\mathcal{X}$*
- *$\mu(0) = 0$,*
- *$u(s) = \mu(s)$ stabilizes (15) on $\mathcal{X}$,*
- *$V(s)$ is finite $\forall s \in \mathcal{X}$.*

Given an admissible u, the Hamiltonian is defined as

$$\mathcal{H}\left(s, u, \frac{\partial V}{\partial s}\right) = \left(\frac{\partial V}{\partial s}\right)^T \ [F(s) + G(s)u] + U(s,u), \tag{18}$$

which inserting (17) yields to the following Bellman equation

$$0 = \mathcal{H}\left(s, u, \frac{\partial V}{\partial s}\right) \tag{19}$$

$$= \left(\frac{\partial V}{\partial s}\right)^T \ [F(s) + G(s)u] + U(s,u), \tag{20}$$

Consider the optimal value function given by

$$V^*(s(t)) = \min_{u(\cdot)\in\pi(\mathcal{X})} \int_t^{\infty} [Q(s(\tau)) + \Theta(u(\tau))] d\tau. \tag{21}$$

Based on optimal control theory, the stationary condition in the Hamiltonian yields the optimal control policy

$$u^*(s) = -\lambda \tanh\left(\frac{1}{2\lambda} R^{-1} G^T(s) \frac{\partial V^*(s)}{\partial s}\right), \tag{22}$$

Utilizing (22) yields the Hamiltonian-Jacobi-Bellman (HJB) equation

$$0 = \left[\frac{\partial V^*(s)}{\partial t}\right] + \mathcal{H}\left(s, u, \frac{\partial V^*}{\partial s}\right), \tag{23}$$

which is a nonlinear partial differential equation and extremely difficult to solve. To this end, an online algorithm with an actor-critic structure is presented to find an approximate solution to the HJB equation.

C. Online Actor-Critic Barrier Learning

In this section, the details of the online algorithm with an actor-critic-barrier structure to learn the optimal control policy is explained. First, the barrier function is employed to transform the original system in (1) to the system in (15). Based on the constrained dynamics (15), two neural networks (NN) are setup. An actor NN and a critic NN. The critic NN is used to approximate the Bellman equation (20) and the actor network is used to approximate the optimal policy (22). Figure 1 shows the structure of the actor-critic-barrier methodology.

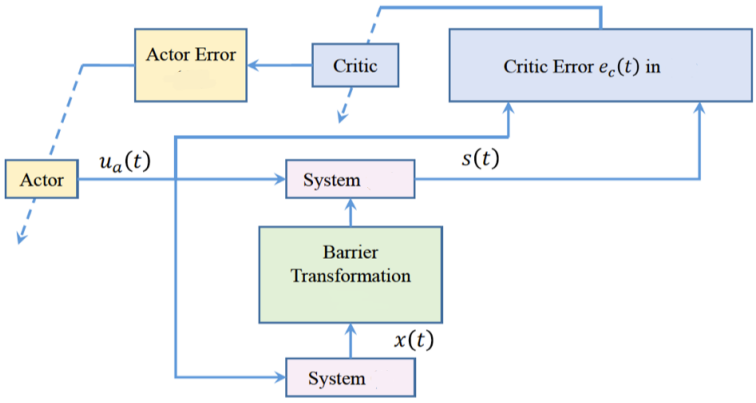

Figure 1: Actor-Critic-Barrier Implementation Structure. [9]

Assumption 2. *There exists positive definite and smooth solution $V(x)$ to the Bellman equation (20).*

Assumption 3. *There exist a single-layer NN such that the value function $V(s)$ and its gradient $\nabla V(s)$ can be uniformly approximated with a critic NN, within a set $\mathcal{X} \subseteq \mathbb{R}^n$ that contains the origin as*

$$V^*(s) = W^T \phi(s) + \epsilon(s) \tag{24}$$

$$\nabla V^*(s) = [\nabla \phi(s)]^T W + \nabla \epsilon(s), \tag{25}$$

where $W \in \mathbb{R}^N$ is the critic weight, $\phi(\cdot) : \mathbb{R}^n \to \mathbb{R}^N$ is the critic basis, $\epsilon(s)$ and $\nabla \epsilon(s)$ are the bounded approximation errors satisfying $\|\epsilon(s)\| \leq b_\epsilon$ and $\|\nabla \epsilon(s)\| \leq b_{d\epsilon}$, $\phi(s)$ and $\nabla \phi(s)$ satisfying $\|\phi(s)\| \leq b_\phi$ and $\nabla \phi(s) \leq b_{bd\phi}$ for all $s \in \mathcal{X}$.

For the optimal control policy $u^*(s)$, the Bellman equation (20) approximation error using the value function approximation (24) can be expressed as

$$\epsilon_B = U(s, u^*) + W^T \sigma \tag{26}$$

where σ is an N-dimentional vector signal defined as

$$\sigma := \nabla \phi(s) \left[F(s) + G(s) u^* \right] \tag{27}$$

Considering the value gradient approximation (25), one can obtain that the Bellman approximation error ϵ_B results results from the value gradient approximation error $\nabla \epsilon(s)$, i.e.,

$$\epsilon_B = - \left[\nabla \epsilon(s) \right]^T \left[F(s) + G(s) u^* \right] \tag{28}$$

From Assumption 2, there exists a constant b_B such that $\|\epsilon_B\| \leq b_B$. The HJB equation (23) can be approximated using the value function approximation (24) as

$$\epsilon_{hjb}(s) = W^T \nabla \phi(s) F(s) + \lambda^2 R \ln \left[\mathbf{I}_m - tanh^2(D) \right] + Q(s), \tag{29}$$

where $D = \frac{1}{2\lambda} R^{-1} G^T(s) (\nabla \phi)^T W$. It is assumed that the ideal value function approximation (24) guarantees $\|\epsilon_{hjb}\| \leq b_{hjb}$.

1) Value Function Approximation: The ideal weight, W in (24), provides the best approximate to the optimal value function $V^*(s)$ on the compact set $\mathcal{X}$ and is unknown. Therefore, the estimation of W is implemented by the critic network with the approximations of the value function and value gradient

$$\hat{V}(s) = W_c^T \phi_c(s) \tag{30}$$

$$\nabla \hat{V}(s) = [\nabla \phi_c(s)]^T W_c, \tag{31}$$

where $\phi_c(s) \in \mathbb{R}^n \to \mathbb{R}^N$ is the critic basis. For a given policy, the residual of Bellman equation approximation using the identifier NN and the critic can be determined as

$$e_c(t) := U(s(t), u(t)) + \hat{W}_c^T(t)\sigma(t) \tag{32}$$

Defining the critic weight approximation error

$$\tilde{W}_c = W - W_c \tag{33}$$

Following from (26), the relation between Bellman residual e_c and the Bellman equation approximation error ϵ_B can be written in terms of the critic weight error $\tilde{W}_c$ as

$$e_c = \epsilon_B - \tilde{W}_c^T \sigma, \tag{34}$$

The policy evaluation for an admissible control policy $u(\cdot)$ can be formulated as adapting the critic weight $\hat{W}_c$ to minimize the objective function [8]

$$E_c = \frac{1}{2} \frac{[e_c(t)]^2}{2(1 + \sigma^T \sigma)} \tag{35}$$

then $e_c \to \epsilon_B$ as $W_c \to W$. Using the chain rule yields the gradient decent algorithm for minimizing the critic weight cost function, i.e., E_c, and is given by [8]

$$\dot{W}_c = -\alpha \frac{\partial E_c}{\partial W_c} = -\alpha_c \frac{\sigma}{(1 + \sigma^T \sigma)^2} [\sigma^T W_c + U(s, u)], \tag{36}$$

2) Actor Learning: Online Synchronous Policy Iteration: Consider the value gradient approximation using the critic weight W_c in (31), the policy can be determined as

$$u_c(s) = -\lambda \tanh(\mathrm{D_c}), \tag{37}$$

where D_c is the approximation of D^* evaluated using critic weights W_c. However, this policy improvement does not guarantee the stability of the system [4], [8]. Therefore, to ensure stability in a Lyaponuv sense the policy applied to the system is implemented by the actor network as

$$u_a(s) = -\lambda \tanh(\mathrm{D_a}), \tag{38}$$

where $D_a = \frac{1}{2\lambda} R^{-1} G^T(s) (\nabla \phi)^T W_a$. The policy evaluation for an admissible control policy can be formulated as adapting the critic weight W_a to minimize the following objective function

$$E_u = \frac{1}{2} e_u^T R e_u, \tag{39}$$

where $e_u = u_c - u_a = \lambda[\tanh(D_a) - \tanh(D_c)]$. Using the chain rule yields the gradient descent algorithm for minimizing E_u. It is proven in [9] that if the following gradient-descent based actor learning update rule

$$\dot{W}_a = -\alpha_a [\nabla \phi G e_u + \nabla \phi G \tanh^2(D_a) e_u + Y W_a], \tag{40}$$

where $Y > \frac{M_a M_a^T}{2}$ and $M = \nabla \phi G \lambda [\tanh(\kappa D_a) - \tanh(D_a)]$ is applied to system and the control input u_c as an approximation of the optimal control policy in (22) with value approximation by (31), then under Assumptions 1-3 and suppose the signal $\frac{\sigma_a(t)}{1 + \sigma_a^T(t)\sigma_a(t)}$ satisfies the persistency of excitation condition the closed-loop system states and the critic and actor NN errors are uniformly ultimately bounded for a sufficiently large number of NN basis. [9]

III. Experimental Evaluations

In this section, the performance and robustness of the online Actor-Critic-Barrier algorithm is evaluated in simulated settings, and then the effectiveness of the method in practice is demonstrated in Baxter robot.

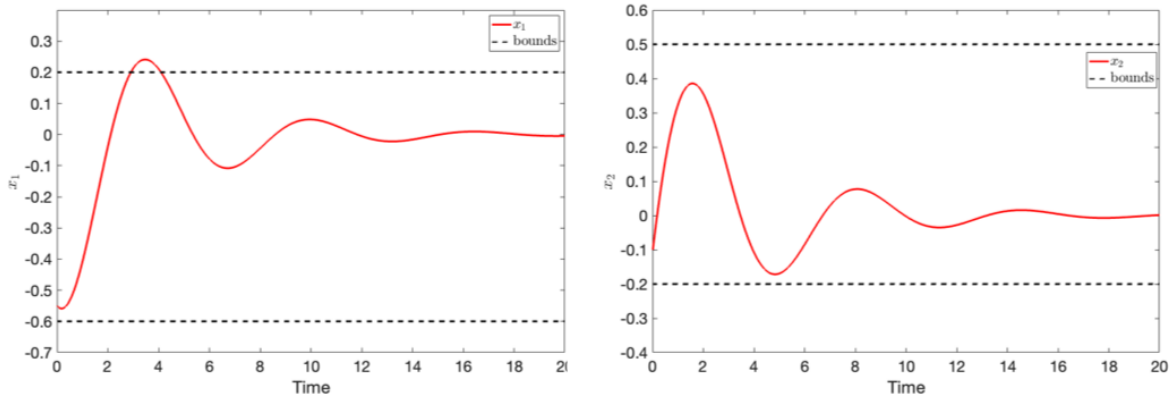

Figure 2: State trajectories when using the converse HJB approach [5].

A. Simulations

The simulations were conducted using a desktop computer with an Intel $i5$ processor and 16 GB RAM. The method was coded using MATLAB $2019a$, and it was tested on the controlled Van der Pol oscillator with the dynamics (41). Similar to [9], it was desired that the state $x = [x_1 \ x_2]^T$ satisfied the following constraints,

$$x_1 \in (-0.6, 0.2) \qquad\qquad x_2 \in (-0.2, 0.5)$$

The method was compared against the converse HJB method [5] when the performance parameters were selected as $Q = I_{2\times 2}$, $R = 1$, and the optimal controller $u = -x_1 x_2$. When applying this optimal control policy to the system (41), the evolution is shown in Figure 2, where the solid lines represent the state evolutions and the dashed lines denote the asymmetric bounds for the states. The objective was to stabilize the dynamics without violating the safety constraints, but as it is seen in Figure 2, even though the states were regulated to the origin asymptotically, the full state constraint objective was not met.

Remark 1. *The choice of a basis function for both actor and critic networks are essential for the algorithm to converge. It is also clear that the approximation error in both networks can be reduced by increasing the number of basis functions. [2], [8]*

To setup the actor-critic network, polynomial function approximation with $N = 3$ neurons as well as radial basis function (RBF) with $N = 9$ and $N = 25$ neurons were selected.

$$\phi(s) = [s_1^2, \ s_1 s_2, \ s_2^2]^T$$

$$\phi_i(s) = e^{\left(-\frac{1}{2\sigma_i^2}(s-c_i)^2\right)}.$$

The means for the RBF kernels were arranged in uniform $\sqrt{N} \times \sqrt{N}$ grid between state bounds. The covariance matrix was initialized to $\sigma_i^2 = 0.3 I_{2\times 2}$, all the NN weights were randomly initialized in $[-1, 1]$, and the states were initialized to $x(0) = [-0.55, -0.1]$.

When the Actor-Critic-Barrier algorithm was applied to the system, the state trajectories converged to the origin without crossing the predefined bounds on the states. The corresponding results of the polynomial and RBF kernels can be seen in Figures 3-5. It is desired to drive the system to the origin without violating the boundaries of each states, and as one can observe in Figure 9, when the online actor-critic method with barrier transformation was used the states approach to the origin without violating the saftey constraints.

Convergence of the actor-critic parameters to the optimal parameters are presented in Figures 6-8. The system state trajectories in two-dimensional space are shown in Figure 9. Figure 10 illustrates the evolution of the reward funtion between RBF and polynomial kernels for the duration of the experiment. As it is mentioned in Remark 1 the choice of kernels are essential for the algorithm to converge. Comparing the evolution of the reward function between RBF kernels, verifies that by increasing the number of basis functions the function approximation error reduces thus no further compensation is required while minimizing the cost.

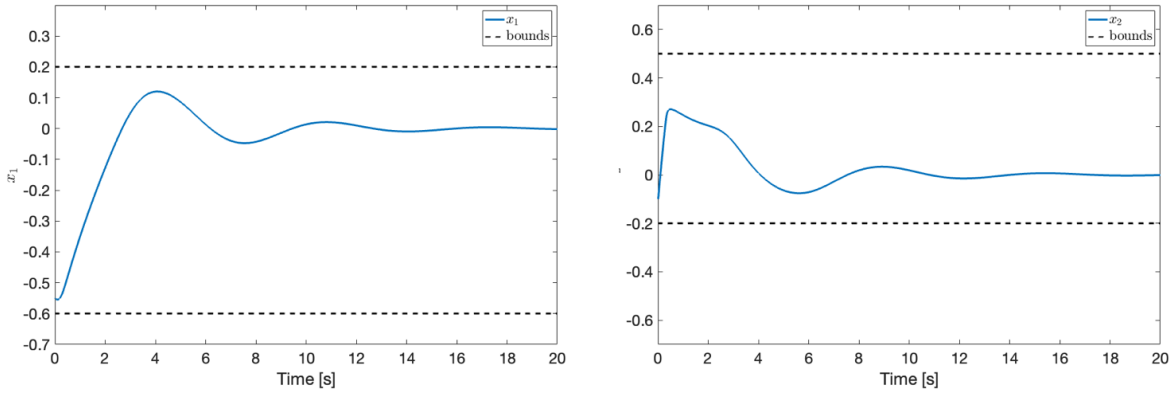

Figure 3: State trajectories with Actor-Critic-Barrier approach when $N = 3$ polynomail kernels were selected.

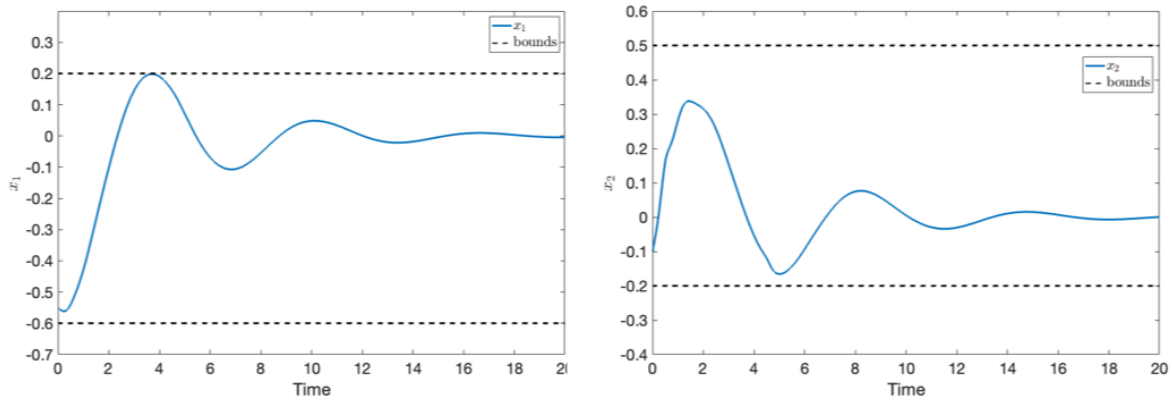

Figure 4: State trajectories with Actor-Critic-Barrier approach when $N = 9$ RBF kernels were selected.

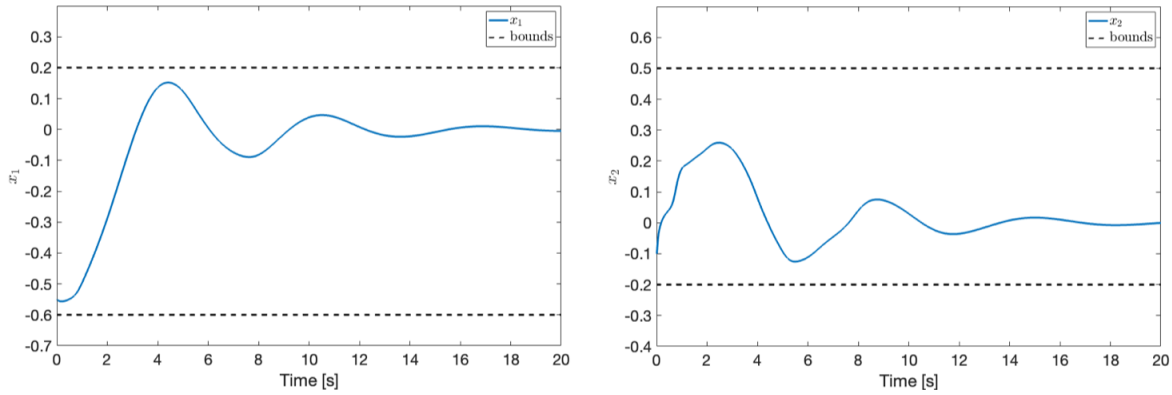

Figure 5: State trajectories with Actor-Critic-Barrier approach when $N = 25$ RBF kernels were selected.

B. Robot Implementation

This section presents the experimental results for the implementation of the Actor-Critic-Barrier algorithm on Rethink Robotics' Baxter Research Robot. We demonstrate the effectiveness of the Actor-Critic-Barrier algorithm in a similar scenario to those results presented in the paper, but on a real robot. The goal of this exercise it to determine the utility of the Actor-Critic-Barrier algorithm in a realistic manufacturing scenario. The experiment was run using a Baxter research robot and a Lenovo Desktop Computer with Intel Quad-Core $i5-4570$ CPU with 3.20 GHz processors and 8 GB RAM. The Operating System used was Ubuntu 14.04. In order to represent the barrier, a set of oversized legos were used.

The experiment portion of the project was coded using Python 2.7, and the main file can be found in A. The

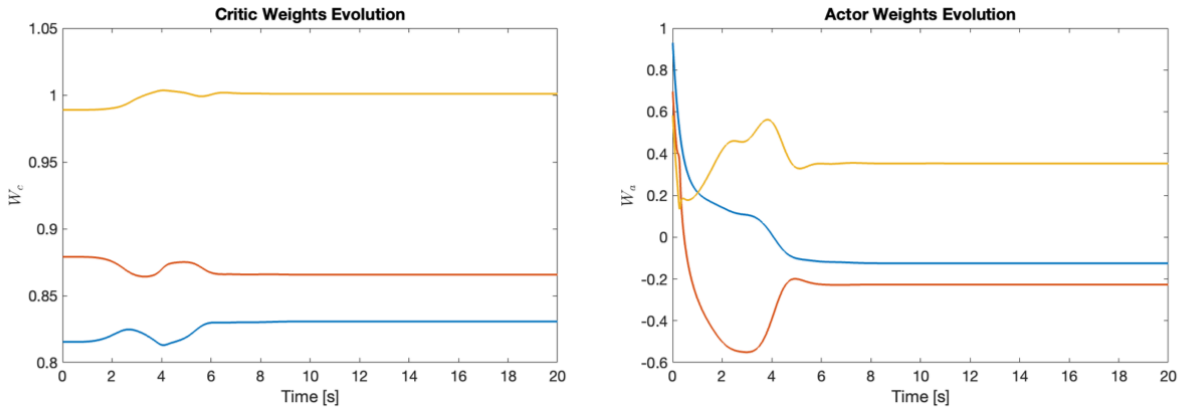

Figure 6: Evolution of the actor-critic weights for the duration of the experiment, when $N = 3$ polynomail kernels were selected.

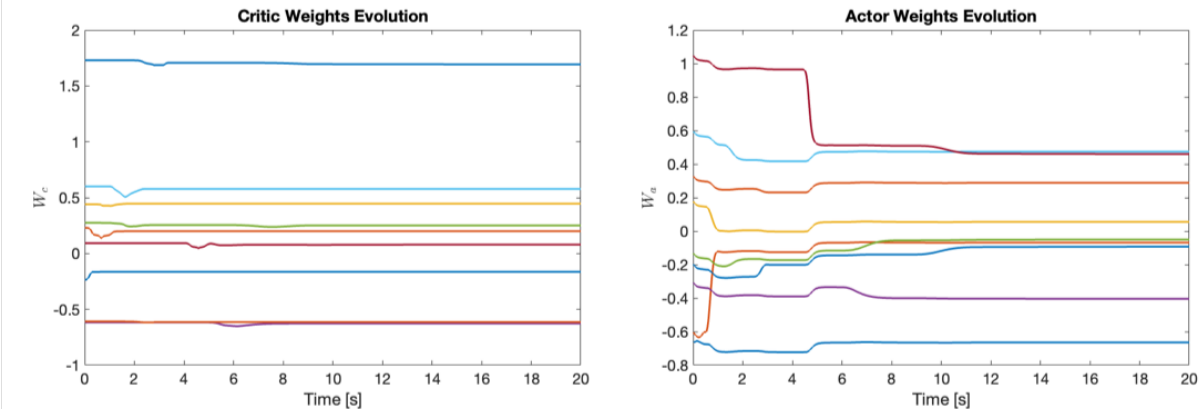

Figure 7: Evolution of the actor-critic weights for the duration of the experiment, when $N = 9$ polynomail kernels were selected.

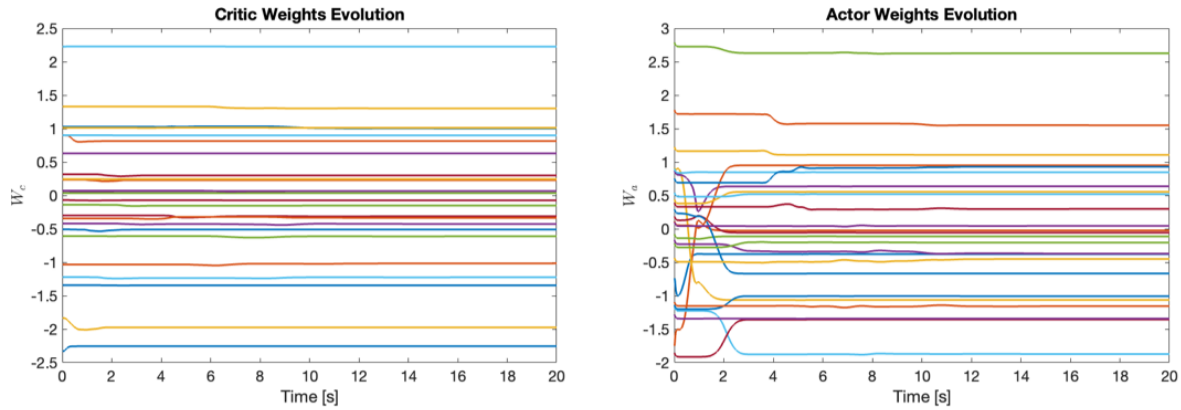

Figure 8: Evolution of the actor-critic weights for the duration of the experiment, when $N = 25$ polynomail kernels were selected.

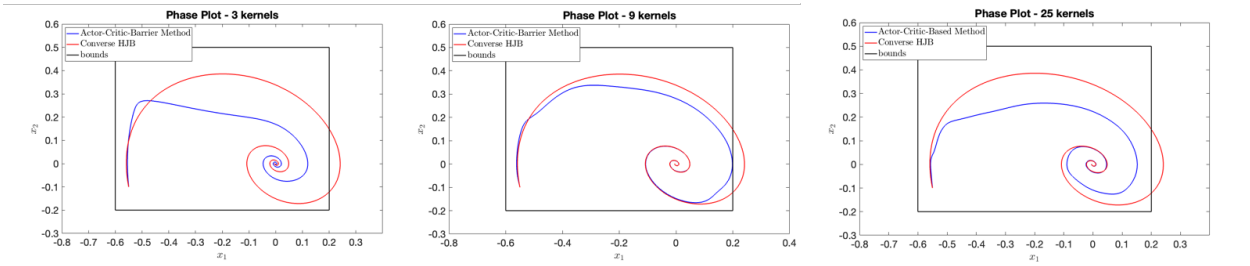

Figure 9: Two-dimensional phase plot of state trajectories using the converse HJB approach [5] and the actor-critic-barrier algorithm [9]. The actor-critic-barrier algorithm was tested on three different choices of activation functions. The black box denotes the safety region.

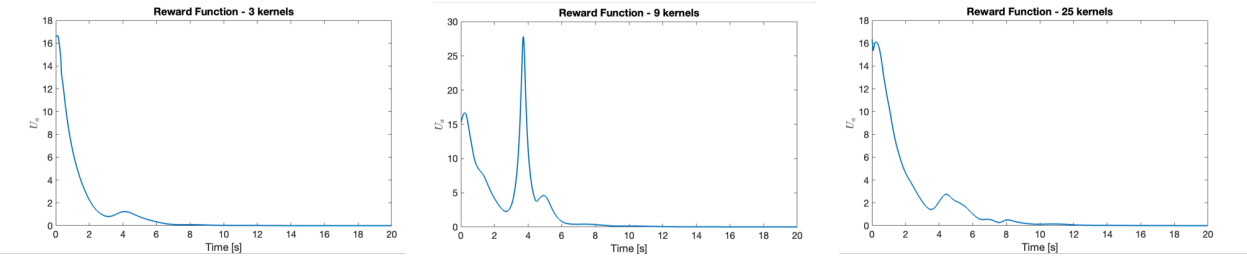

Figure 10: Evolution of the reward functions of polynomial and RBF kernels for the duration of the experiment.

project was coded using object oriented programming techniques, and the main parts are isolated by functionality: objects are created for the barrier, for the Actor-Critic, and for the Robot Controller separately. This way, the function calls for each independent part are grouped together, and can be reused for different experiments and simulations. All interaction between the processor and the Robot are handled by ROS, the Robot Operating System.

The implementation on the physical system has several major differences over the general implementation of Figure 1. The adapted implementation can be seen in Figure 11. While the reinforcement learning actor-critic model (green block) remains largely unchanged, the major change happens with the inclusion of the physical system itself. The red blocks are added parts to the framework in order to it to the real life implementation. After the actor computes the estimated rate of change of the transformed state, the signal is then transformed back to the unconstrained domain by using (10) and (11). From here, we get a velocity vector in the Cartesian space, which is then multiplied by a scalar gain, which was determined experimentally. The inverse Jacobian is then solved for and multiplied by the velocity vector, which translates the velocity vector into individual joint velocities which can then be sent to the robot to be executed. After that, the new position feedback from the robot is taken as the next state. This state is then transformed back into the constrained domain, and is fed back into the actor-critic system.

In the experiment, a table is set up alongside Baxter, and an approximate barrier is constructed for viewing purposes with the legos. The states x_1 and x_2 are chosen as the robot end-effector's Cartesian coordinates in x and y with respect to the robot's base frame. The barrier specifications are set to $a_1 = -0.4$, $a_2 = -0.2$, $A_1 = 0.1$, and $A_2 = 0.2$. The other parameters are set as $\lambda = 8.0$, $\alpha_a = 1.5$, $\alpha_c = 0.1$, $\kappa = 1.0$, $c = 1.0$, $R = 1.0$, $Q = I(2)$. The system is chosen to follow the dynamics of a Van der Pol oscillator:

$$\dot{x} = \begin{bmatrix} x_2 \\ -x_1 - 0.5\left(1 - x_1^2\right) x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ x_1 \end{bmatrix} u \tag{41}$$

In the first part of the experiment, the robot is run with the optimal controller determined by the reverse HJB method [5], which does not take the barrier into account. According to the HJB method, the optimal controller for these dynamics and parameters is given by $u^*(x) = -x_1 x_2$. The results of this experiment can be seen in Figure 12. As can be seen in the fourth and fifth frames, the robotic arm clearly breaks the barrier, exiting the safety region. This is considered an unsafe motion. The state evolution can be seen in Figure 13, as well as the combined phase plot in Figure 16.

In the second part of the experiment, the safe reinforcement learning framework is applied to the controller. The framework is applied with the same dynamics, and same initial conditions as in the previous experiment. The results of this experiment can be seen in Figure 14. We can see that the robot maintains the safety condition throughout the

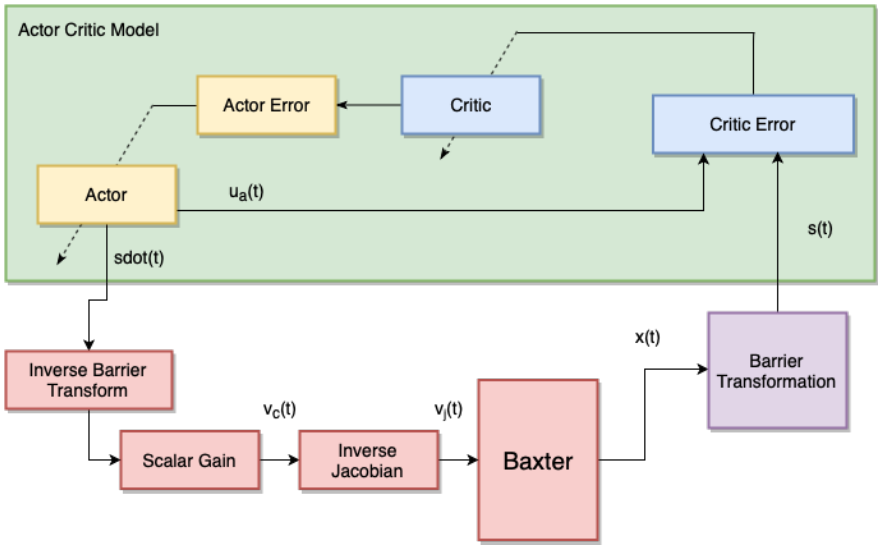

Figure 11: Actor-Critic-Barrier structure adapted for robot experimentation.

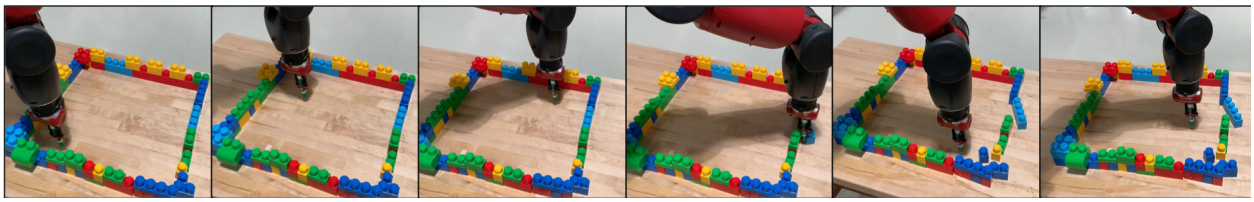

Figure 12: Robot experiment without barrier. The safety region is denoted by the lego bricks.

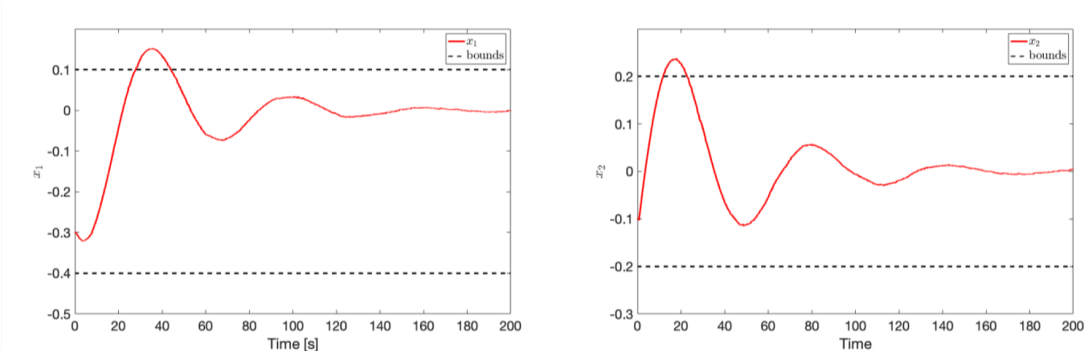

Figure 13: State evolution of x_1 (left) and x_2 (right) for the Converse HJB robot implementation.

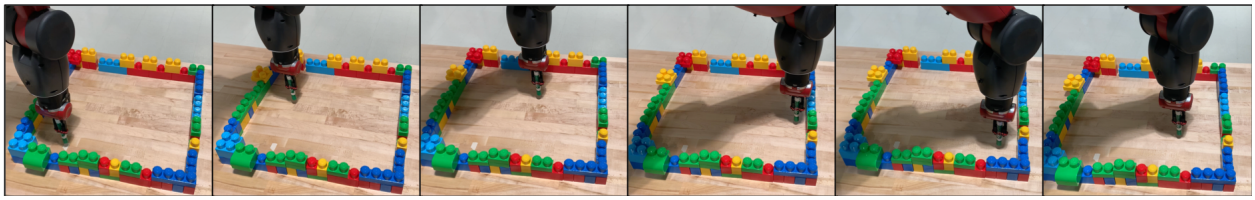

Figure 14: Robot experiment with barrier. The safety region is denoted by the lego bricks.

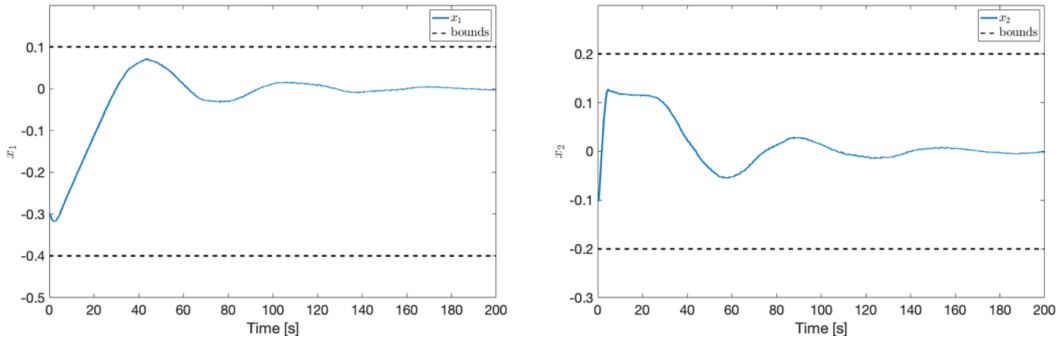

Figure 15: State evolution of x_1 (left) and x_2 (right) for the Actor-Critic-Barrier robot implementation.

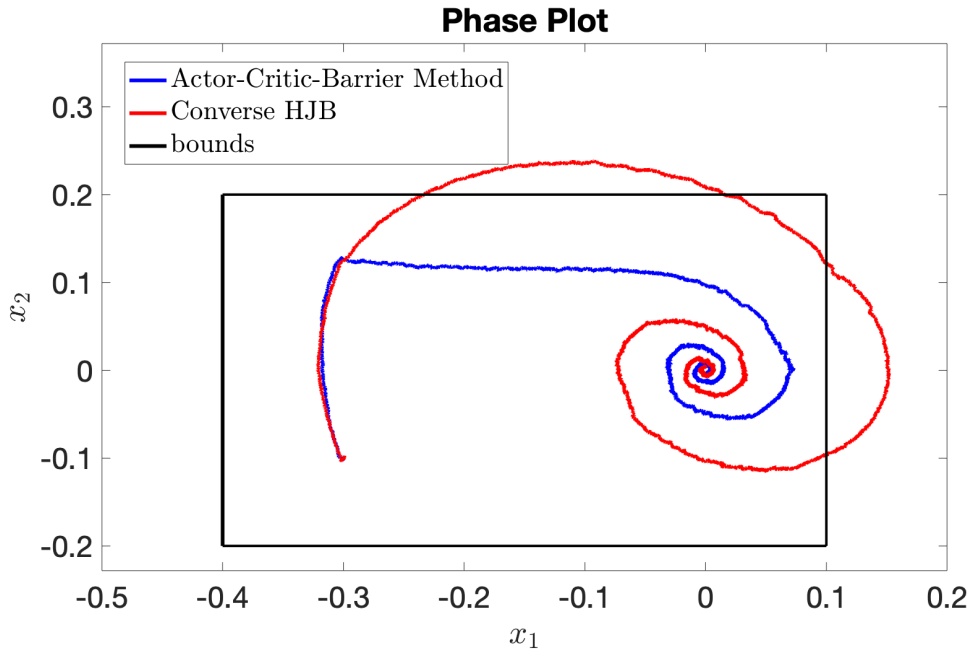

Figure 16: Robot experiment with barrier. The safety region is denoted by the lego bricks.

entire trial. Looking at frames 4 and 5, in the places where the converse HJB method fails, the robot stays within the boundary. The combined phase plot of the two experiments can be seen in Figure 16, and the phase evolution of the Actor-Critic-Barrier can be seen in Figure 15. The evolution of the actor and critic weights can be seen in Figure 18, and the evolution of the reward function can be seen in 17.

During the experimentation portion, a major difference between the practical implementation and computer simulation was immediately apparent. Due to the distributed nature of the system, a great deal of latency was introduced when the approach was tried on the robot. During the simulation, the system goes through an iteration every 1ms. On the robot system, the time in between iterations was found to be approximately 24ms, so there is a much greater delay in the weight updates. At full speed, in the first second, rather than receiving 1000 updates as in simulation, the actor-critic system only receives around 40 updates. In our initial experiments, we found that the latency had such an effect that the initial velocity vector would carry the end effector outside the barrier almost immediately, due to the weights not having enough time to update. To address this issue, we introduced a scalar gain to the controller before solving for the inverse Jacobian. This scalar gain is set between 0 and 1, and scales down the velocity output so that execution slows down, and the actor-critic system is allowed enough iterations to learn the system. In our experiments we chose 0.1 as this gain.

The limiting gain addition helped to recreate the result observed in simulation, but is not an adequate solution to allow this method to be used in an industrial setting with our current setup. In order to allow for this type of learning

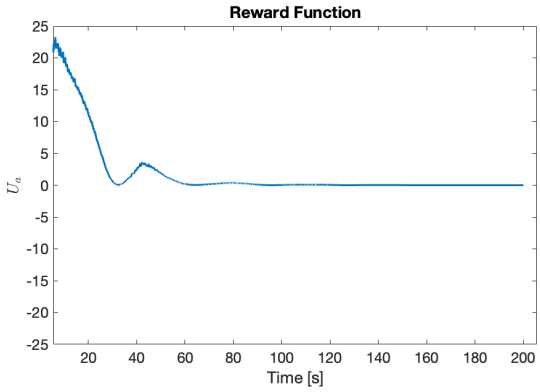

Figure 17: Reward function evolution with time.

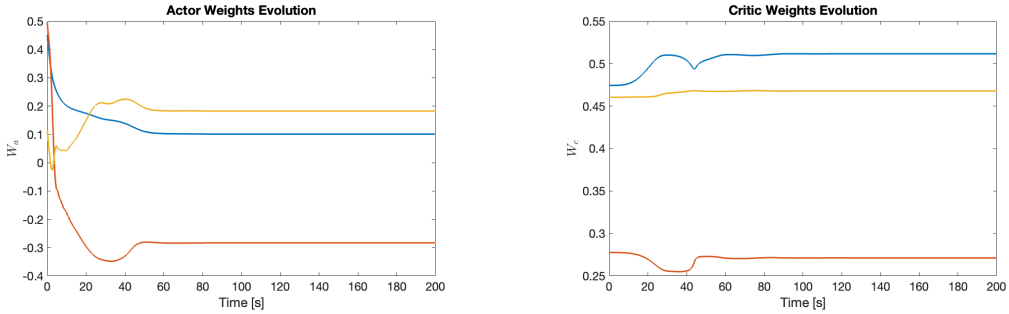

Figure 18: Evolution of actor-critic weights with time.

system to be used, the feedback between the robot and the controller must have low latency to allow the weights of the system enough time to update, or the original dynamics of the system must be relatively slow. Another possible solution is to use a more sophisticated reinforcement learning system, like in [6], where the authors use a PD controller in parallel with an actor-critic system to improve the tracking performance of the nominal feedback controller.

IV. Conclusion

In this project, we investigated a reinforcement learning based approach for imposing barrier constraints on a state space. We recreated the original authors' results in a simulation settings, and then added more simulations to further explore the performance of the approach. We also adapted and implemented this approach for use on a Baxter research robot, and compared the results to another implementation method without barrier constraints. We found that the Actor-Critic-Barrier system performed as expected when the weights of the reinforcement learning system were allowed enough time to update, but the approach failed when run at full speed due to the latency included in the physical system. To allow the sytem enough time to update, a fractional scalar gain was introduced before the inverse Jacobian was applied, in order to scale down the velocity vector commanded to the robot, thereby slowing down the execution of the system.

References

[1] Murad Abu-Khalaf and Frank L Lewis. Nearly optimal control laws for nonlinear systems with saturating actuators using a neural network hjb approach. *Automatica*, 41(5):779–791, 2005.

[2] Shubhendu Bhasin, Rushikesh Kamalapurkar, Marcus Johnson, Kyriakos G Vamvoudakis, Frank L Lewis, and Warren E Dixon. A novel actor–critic–identifier architecture for approximate optimal control of uncertain nonlinear systems. *Automatica*, 49(1):82–92, 2013.

[3] Wei He, Yuhao Chen, and Zhao Yin. Adaptive neural network control of an uncertain robot with full-state constraints. *IEEE transactions on cybernetics*, 46(3):620–629, 2015.

[4] Hamidreza Modares, Frank L Lewis, and Mohammad-Bagher Naghibi-Sistani. Adaptive optimal control of unknown constrained-input systems using policy iteration and neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 24(10):1513–1525, 2013.

[5] Vesna Nevistić and James A Primbs. Constrained nonlinear optimal control: a converse hjb approach. 1996.

[6] Yudha P Pane, Subramanya P Nageshrao, and Robert Babuška. Actor-critic reinforcement learning for tracking control in robotics. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 5819–5826. IEEE, 2016.

[7] Keng Peng Tee, Shuzhi Sam Ge, and Eng Hock Tay. Barrier lyapunov functions for the control of output-constrained nonlinear systems. *Automatica*, 45(4):918–927, 2009.

[8] Kyriakos G Vamvoudakis and Frank L Lewis. Online actor–critic algorithm to solve the continuous-time infinite horizon optimal control problem. *Automatica*, 46(5):878–888, 2010.

[9] Yongliang Yang, Yixin Yin, Wei He, Kyriakos G Vamvoudakis, Hamidreza Modares, and Donald C Wunsch. Safety-aware reinforcement learning framework with an actor-critic-barrier structure. In *2019 American Control Conference (ACC)*, pages 2352–2358. IEEE, 2019.

APPENDIX A
PROJECT CODE

```
#!/usr/bin/env python

"""
File: main.py
Author: Ryan Saltus and Iman Salehi
Email: ryan.saltus@uconn.edu and iman.salehi@uconn.edu
Description: The main code for the OMPC Project.
"""
import numpy as np
from numpy import matmul
from math import exp, log, atanh, tanh
import matplotlib.pyplot as plt

import scipy.io as sio

import rospy
import baxter_interface
from baxter_pykdl import baxter_kinematics
from baxter_core_msgs.msg import EndpointState

class barrier(object):

   """A class for implementing barrier transformations based on VdP Oscillator
       dynamics"""

   def __init__(self, a1, a2, A1, A2):
      self.a = np.array([[a1], [a2]])
      self.A = np.array([[A1], [A2]])

   def log_transform(self, x):
      """Logarithmic barrier function transformation."""

      s = np.log(np.multiply(self.A, (self.a-x))/np.multiply(self.a, self.A-x))
      return s

   def inverse(self, s):
      """Inverts the state in constrained state back to the original space."""

      pre = np.multiply(self.a, self.A)
      num = np.exp(s/2)-np.exp(-s/2)
      den = np.multiply(self.a, np.exp(s/2)) - np.multiply(self.A, np.exp(-s/2))

      x = np.multiply(pre, num)
      x = np.divide(x, den)

      return x

   def dot_inverse(self, s):
      """Calculates the time derivative of the inverse of the barrier function"""
      num = np.multiply(self.A, self.a**2) - np.multiply(self.a, self.A**2)
      den = np.multiply(self.a**2, np.exp(s)) - 2*np.multiply(self.a, self.A) \
                              + np.multiply((self.A**2), np.exp(-s))
      return np.divide(num, den)

   def dyn_transform(self, s):
      """Applies a logarithmic barrier function transformation to unconstrained state
      dynamics, to transform the dynamics into a constrained space."""

      num = (self.A[-1,0]**2)*exp(-s[-1,0]) - 2*self.a[-1,0]*self.A[-1,0] \
                             + (self.a[-1,0]**2)*exp(s[-1,0])
```

```
        den = self.A[-1,0]*(self.a[-1,0]**2) - self.a[-1,0]*(self.A[-1,0]**2)
        multiplier = num/den

        x = self.inverse(s)
        dotinv = self.dot_inverse(s)[0,0]
        fx = -x[0,0] - 0.5*x[-1,0]*(1-(x[0,0]**2))

        # Transformation of state dynamics
        Gs = np.array([[0], [multiplier*x[0,0]]])
        Fs = np.array([[x[-1,0]/dotinv], [multiplier*fx]])

        return Fs, Gs

class ActorCritic(object):

    """A class implementing the Actor Critic network. """

    def __init__(self, itera):

        self.lmbda = 8.0
        self.alpha_a = 1.5
        self.alpha_c = .1
        self.kappa = 1.0
        self.c = 1.0
        self.dt = .001

        self.R = 1.0
        self.Q = np.identity(2)

        self.num_kernels = 3
        self.Wa = np.random.rand(self.num_kernels,1)
        self.Wc = np.random.rand(self.num_kernels,1)

        self.iterations = itera
        self.s_array = np.zeros((2, self.iterations))
        self.x_array = np.zeros((2, self.iterations))
        self.wa_array = np.zeros((self.num_kernels, self.iterations))
        self.wc_array = np.zeros((self.num_kernels, self.iterations))
        self.Ua_array = np.zeros((1, self.iterations))
        self.xdot = np.zeros((2,1))

    def record_data(self, iteration, s, x, Ua):
        self.s_array[:,iteration] = s[:,0].copy()
        self.x_array[:,iteration] = x[:,0].copy()
        self.wa_array[:,iteration] = self.Wa[:,0].copy()
        self.wc_array[:,iteration] = self.Wc[:,0].copy()
        self.Ua_array[:,iteration] = Ua

    def offline_update(self, iteration, s, barrier, dt):

        #self.dt = dt
        old_x = barrier.inverse(s)
        Fs, Gs = barrier.dyn_transform(s)
        dphi = np.array([[2.0*s[0,0], s[1,0], 0],[ 0, s[0,0], 2.0*s[1,0]]]).T

        Da = (1/(2*self.lmbda))*(1/self.R)*matmul(matmul(Gs.T, dphi.T), self.Wa)
        ua = -self.lmbda*np.tanh(Da)

        sdot = Fs + Gs*ua
        Dc = (1/(2.0*self.lmbda))*(1/self.R)*matmul(matmul(Gs.T, dphi.T), self.Wc)
        uc = -self.lmbda*np.tanh(Dc)

        eu = uc - ua
        sigma_a = matmul(dphi, sdot)
```

```
        a = 2.0*self.lmbda*self.R*(0.5*self.lmbda*log(self.lmbda**2-ua**2)
           +ua+atanh(ua/self.lmbda))
        b =
            2.0*self.lmbda*self.R*(0.5*self.lmbda*log(self.lmbda**2)+atanh(0/self.lmbda))
        Ua = matmul(matmul(s.T, self.Q), s) + (a-b)
        M = matmul(dphi, Gs)*self.lmbda*(np.tanh(self.kappa*Da)-np.tanh(Da))
        Y = matmul(M, M.T)/2.0 + self.c*np.ones((self.num_kernels, self.num_kernels))

        Wa_dot = -self.alpha_a*(matmul(dphi, Gs)*eu + matmul(dphi, Gs)*(tanh(Da)**2)*eu
            + matmul(Y, self.Wa))
        self.Wa = self.Wa + self.dt*Wa_dot

        Wc_dot = -self.alpha_c*(np.divide(sigma_a,((1+matmul(sigma_a.T,
            sigma_a))**2))*(Ua+matmul(self.Wc.T,sigma_a)))
        self.Wc = self.Wc + self.dt*Wc_dot

        s = s + sdot * self.dt
        x = barrier.inverse(s)
        self.xdot = np.multiply(sdot, barrier.dot_inverse(s))

        self.record_data(iteration, s, x, Ua)

        return s, sdot, self.xdot

class RobotController(object):

    def __init__(self, limb):
        self._control_arm = baxter_interface.limb.Limb(limb)
        self._kin = baxter_kinematics(limb)

    def command_velocity(self, command):
        rate = rospy.Rate(100)

        control_joint_names = self._control_arm.joint_names()
        jacob_i = self._kin.jacobian_pseudo_inverse()
        self._vel_command = np.transpose([command[0,0], command[1,0], 0, 0, 0, 0])
        vel = matmul(jacob_i, self._vel_command)
        self._joint_command = {'left_s0':vel[0,0], 'left_s1':vel[0,1],
            'left_e0':vel[0,2],
                        'left_e1':vel[0,3], 'left_w0':vel[0,4], 'left_w1':vel[0,5],
                        'left_w2':vel[0,6]}
        self._control_arm.set_joint_velocities(self._joint_command)
        rate.sleep()

def test_barrier():
    nn = ActorCritic()
    btest = barrier(-0.6, -0.2, 0.2, 0.5)
    x = np.array([[-0.55], [-0.1]])
    state = btest.log_transform(x)
    inv_state = btest.inverse(state)
    dyn = btest.dyn_transform(state)
    newstate = nn.online_update(state, btest)

def test_ac():

    nn = ActorCritic(100000)
    #bar = barrier(-0.6, -0.2, 0.2, 0.5)

    #x = np.array([[-0.3], [0.4]])
    bar = barrier(-0.4, -0.4, 0.1, 0.2)

    x = np.array([[-0.3], [-0.1]])
    state = bar.log_transform(x)
```

```
    dt = .001
    count = 0

    while count < nn.iterations:
        state, state_dot, _ = nn.offline_update(count, state, bar, dt)
        count += 1

    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.plot(nn.x_array[0,:], nn.x_array[1,:])

    l_x, l_y = [bar.a[0], bar.a[0]], [bar.a[1], bar.A[1]]
    r_x, r_y = [bar.A[0], bar.A[0]], [bar.a[1], bar.A[1]]
    t_x, t_y = [bar.a[0], bar.A[0]], [bar.A[1], bar.A[1]]
    b_x, b_y = [bar.a[0], bar.A[0]], [bar.a[1], bar.a[1]]

    ax1.plot(l_x, l_y, r_x, r_y, t_x, t_y, b_x, b_y, marker='o', color='k')
    ax1.set_xlim([-0.8, 0.3])
    ax1.set_ylim([-0.3, 0.6])

    plt.show()


def test_system():
    rospy.init_node('RL_barrier', anonymous=True)
    nn = ActorCritic(10000)
    #bar = barrier(-0.4, -0.2, 0.3, 0.2)
    #bar = barrier(-0.2, -0.6, 0.1, 0.6)
    bar = barrier(-0.4, -0.2, 0.1, 0.2)

    x = np.array([[-0.3], [-0.1]])
    #x = np.array([[-0.3], [0.1]])

    rc = RobotController('left')

    initial = rospy.wait_for_message('/robot/limb/left/endpoint_state', EndpointState)
    init_x = initial.pose.position.x
    init_y = initial.pose.position.y
    then = rospy.get_rostime()
    then = then.to_sec()
    then = float(then)
    prev_time = initial.header.stamp.secs + initial.header.stamp.nsecs*(10**-9)

    transform_vector = np.array([[init_x - x[0,0]],[init_y-x[1,0]]])
    state = bar.log_transform(x)
    count = 0

    while count < nn.iterations:

        curr_state = rospy.wait_for_message('/robot/limb/left/endpoint_state',
            EndpointState)
        now = rospy.get_rostime()
        now = now.to_sec()
        now = float(now)
        feedback_x = curr_state.pose.position.x
        feedback_y = curr_state.pose.position.y
        dt = now - then

        feedback_state = np.array([[feedback_x],[feedback_y]])
        feedback_state = feedback_state - transform_vector
        feedback_state = bar.log_transform(feedback_state)

        _, _, xdot = nn.offline_update(count, feedback_state, bar, dt)
```

```
        #xdot = xdot/10.0

        rc.command_velocity(xdot)
        count += 1
        then = now

    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.plot(nn.x_array[0,:], nn.x_array[1,:])
    l_x, l_y = [bar.a[0], bar.a[0]], [bar.a[1], bar.A[1]]
    r_x, r_y = [bar.A[0], bar.A[0]], [bar.a[1], bar.A[1]]
    t_x, t_y = [bar.a[0], bar.A[0]], [bar.A[1], bar.A[1]]
    b_x, b_y = [bar.a[0], bar.A[0]], [bar.a[1], bar.a[1]]

    ax1.plot(l_x, l_y, r_x, r_y, t_x, t_y, b_x, b_y, marker='o', color='k')
    ax1.set_xlim([-0.8, 0.3])
    ax1.set_ylim([-0.3, 0.6])

    fig2 = plt.figure()

    ax2 = fig2.add_subplot(111)
    ax2.plot(nn.wa_array[0,:])
    ax2.plot(nn.wa_array[1,:])
    ax2.plot(nn.wa_array[2,:])

    ax3 = fig2.add_subplot(111)
    ax3.plot(nn.wc_array[0,:])
    ax3.plot(nn.wc_array[1,:])
    ax3.plot(nn.wc_array[2,:])

    fig3 = plt.figure()
    ax1.plot(nn.Ua_array[0,:])

    plt.show()

    sio.savemat('critic_weights.mat', {'wc':nn.wc_array})
    sio.savemat('actor_weights.mat', {'wa':nn.wa_array})
    sio.savemat('demo_w_barrier.mat', {'x_bar':nn.x_array})
    sio.savemat('reward.mat', {'Ua':nn.Ua_array})

def test_dynamics():
    rospy.init_node('RL_barrier', anonymous=True)
    #bar = barrier(-0.4, -0.2, 0.3, 0.2)
    #bar = barrier(-0.2, -0.6, 0.1, 0.6)
    bar = barrier(-0.4, -0.4, 0.1, 0.3)

    x = np.array([[-0.3], [-0.1]])
    #x = np.array([[-0.3], [0.1]])

    rc = RobotController('left')

    initial = rospy.wait_for_message('/robot/limb/left/endpoint_state', EndpointState)
    init_x = initial.pose.position.x
    init_y = initial.pose.position.y
    then = rospy.get_rostime()
    then = then.to_sec()
    then = float(then)
    prev_time = initial.header.stamp.secs + initial.header.stamp.nsecs*(10**-9)

    transform_vector = np.array([[init_x - x[0,0]],[init_y-x[1,0]]])
    state = bar.log_transform(x)
    count = 0
    iterations = 10000
    record_array = np.zeros((2,iterations))
```

```
    while count < iterations:

        curr_state = rospy.wait_for_message('/robot/limb/left/endpoint_state',
            EndpointState)
        now = rospy.get_rostime()
        now = now.to_sec()
        now = float(now)
        feedback_x = curr_state.pose.position.x
        feedback_y = curr_state.pose.position.y
        dt = now - then

        feedback_state = np.array([[feedback_x],[feedback_y]])
        feedback_state = feedback_state - transform_vector

        record_array[:, count] = feedback_state[:,0]

        sy, sx = feedback_state[1,0], feedback_state[0,0]

        ustar = -sy*sx
        x1_d = sy
        x2_d = -sx-.5*(1-sx**2)*sy + sx*ustar;

        xdot = np.array([[x1_d],[x2_d]])
        xdot = xdot/10.0

        rc.command_velocity(xdot)
        count += 1
        then = now

    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.plot(record_array[0,:], record_array[1,:])
    l_x, l_y = [bar.a[0], bar.a[0]], [bar.a[1], bar.A[1]]
    r_x, r_y = [bar.A[0], bar.A[0]], [bar.a[1], bar.A[1]]
    t_x, t_y = [bar.a[0], bar.A[0]], [bar.A[1], bar.A[1]]
    b_x, b_y = [bar.a[0], bar.A[0]], [bar.a[1], bar.a[1]]

    ax1.plot(l_x, l_y, r_x, r_y, t_x, t_y, b_x, b_y, marker='o', color='k')
    ax1.set_xlim([-0.8, 0.3])
    ax1.set_ylim([-0.3, 0.6])

    plt.show()
    sio.savemat('demo_wo_barrier.mat', {'x_nbar':record_array})

if __name__ == "__main__":
    test_system()
    #test_dynamics()
    #test_ac()
```