DISSERTATION

A SYNTHESIS OF REINFORCEMENT LEARNING AND ROBUST CONTROL THEORY

Submitted by
R. Matthew Kretchmar
Department of Computer Science

In partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
Colorado State University
Fort Collins, Colorado
Summer 2000

ABSTRACT OF DISSERTATION

A SYNTHESIS OF REINFORCEMENT LEARNING AND ROBUST CONTROL THEORY

The pursuit of control algorithms with improved performance drives the entire control research community as well as large parts of the mathematics, engineering, and artificial intelligence research communities. A fundamental limitation on achieving control performance is the conflicting requirement of maintaining system stability. In general, the more aggressive is the controller, the better the control performance but also the closer to system instability.

Robust control is a collection of theories, techniques, and tools that form one of the leading edge approaches to control. Most controllers are designed not on the physical plant to be controlled, but on a mathematical model of the plant; hence, these controllers often do not perform well on the physical plant and are sometimes unstable. Robust control overcomes this problem by adding uncertainty to the mathematical model. The result is a more general, less aggressive controller which performs well on both the model and the physical plant. However, the robust control method also sacrifices some control performance in order to achieve its guarantees of stability.

Reinforcement learning based neural networks offer some distinct advantages for improving control performance. Their nonlinearity enables the neural network to implement a wider range of control functions, and their adaptability permits them to improve control performance via on-line, trial-and-error learning. However, neuro-control is typically plagued by a lack of stability guarantees. Even momentary instability cannot be tolerated in most physical plants, and thus, the threat of instability prohibits the application of neuro-control in many situations.

In this dissertation, we develop a stable neuro-control scheme by synthesizing the two fields of reinforcement learning and robust control theory. We provide a learning system with many of the advantages of neuro-control. Using functional uncertainty to represent the nonlinear and time-varying components of the neural networks, we apply the robust control techniques to guarantee the stability of our neuro-controller. Our scheme provides stable control not only for a specific fixed-weight, neural network, but also for a neuro-controller in which the weights are changing during learning. Furthermore, we apply our stable neuro-controller to several control tasks to demonstrate that the theoretical stability guarantee is readily applicable to real-life control situations. We also discuss several problems we encounter and identify potential avenues of future research.

R. Matthew Kretchmar
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 2000

ACKNOWLEDGEMENTS

An enormous credit is due to Dr. Charles Anderson of the Computer Science Department. Chuck served as my advisor for four and a half years, introduced me to reinforcement learning and nurtured me through my graduate career. Without his patient assistance, close collaboration, and careful guidance this dissertation would not have come to fruition.

Special thanks is due to Dr. Peter Young and Dr. Douglas Hittle of the Electrical Engineering and Mechanical Engineering departments, respectively. This dissertation would not have been possible without Peter's expertise in control theory and robust control theory, and without Doug's thorough knowledge of HVAC control systems. I would like to thank Dr. Adele Howe and Dr. Darrell Whitley of the Computer Science Department. While serving as committee members, they offered excellent technical advice to make the dissertation a more coherent document. Thanks is due to all five members of the committee for countless hours of document reading and editting.

From a financial standpoint, thanks is due to the National Science Foundation. The work in this dissertation is funded by grants CMS-9401249 and CMS-980474. Thanks is due to Charles Anderson, Douglas Hittle, and Peter Young who served as the principle investigators of these NSF grants and provided my research assistantship. Financial assistance was also provided by the Department of Computer Science in the form of computing equipment and a teaching assistantship. The Colorado State University is also thanked for additional scholarship funds.

I wish to thank my parents, R. Scott and Janet Kretchmar, and my sister, Jennifer Kretchmar, who provided support and motivation with a "dissertation race".

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Problem Statement

Automated controllers provide control signals to a plant in an attempt to cause the plant to exhibit a desired behavior. Here we use "plant" as a generic term for a device that is capable of being controlled. Together, the plant, the controller, and their interconnection comprise the system. The design of controllers is complicated by system instability which results in at least improper plant operation, and possibly, significant damage to equipment and/or injury to people. Fortunately, a properly designed controller prevents the system from operating in a dangerous, unstable mode. Therefore, it is imperative that the controller be engineered with stable operation as a primary goal; performance is a secondary design consideration to be pursued after stability is assured.

The design of such controllers can be approached from a number of design philosophies. In this dissertation, we focus on two diametric design philosophies. The first design philosophy, *robust control*, exploits significant a priori system knowledge in order to construct a high-performing controller that still guarantees stability. The other design philosophy, *reinforcement learning*, builds a controller assuming little initial knowledge of the system but is capable of learning and adapting to find better control functions. The robust controller uses extensive system knowledge but is fixed and rigid for all time; the reinforcement learning controller uses limited system knowledge but is capable of continuous adaptation to find better control schemes. The opposition of these two design approaches is illustrated in Figure 1.1.

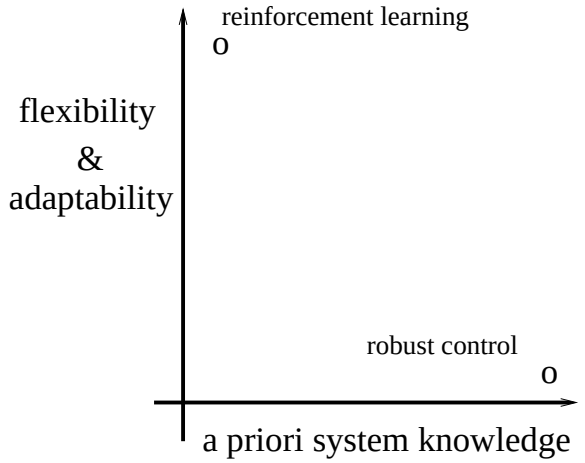


Figure 1.1: Controller Design Philosophies

In robust control, we analyze the dynamics of the plant in an attempt to build a controller that

is mathematically guaranteed to provide stable control behavior. Often we discover an entire set of stable controllers; we then select the best performing controller from this stable set. However, most plants of practical interest possess enough complexity to prohibit the precise specification of the plant dynamics; usually we are forced to compute a mathematical model of the plant to serve as an approximation to the real plant. Consequently, the robust control design process is complicated because we must not only construct a controller that is stable for our mathematical model, but is also stable for the real plant. Necessarily, this limits the aggressiveness of the controller design and thus, results in suboptimal control performance.

On the other hand, reinforcement learning assumes little about the dynamics of the system. Instead, it develops a good control function through on-line, trial and error learning. The challenge of this approach is to establish a framework with enough flexibility to allow the controller to adapt to a good control strategy. However, this flexibility may result in numerous undesirable control strategies; the engineer must be willing to allow the controller to temporarily assume many of these poorer control strategies as it searches for the better ones. It is important to note that many of the undesirable strategies may provide unstable control behavior. We can envision the implausibility of the reinforcement learning approach in designing a controller for a helicopter rotor; the adaptive controller may crash thousands, even tens of thousands of helicopters before it finds a stable control function. However, once the reinforcement learner settles in on reasonably good control strategies, it can refine its search and often discover an exemplary controller that is not only stable, but also outperforms controllers designed using robust control techniques.

Thus, the problem this dissertation examines can be summarized as follows. We desire a robust control approach because this approach guarantees stable control behavior and exploits known system knowledge to achieve relatively good initial control performance. But, a robust control approach sacrifices some control performance in order to achieve the stability guarantee. A reinforcement learning approach is attractive because it is able to discover excellently performing controllers via trial-and-error search, but might temporarily implement a variety of unstable control functions. The problem is to combine these two techniques to guarantee stability and also perform safe trial-and-error search in order to adaptively improve control performance. We now present this problem in more detail by defining plants, stability, performance and numerous other terms.

1.2 Problem Details

Figure 1.2 shows the basic components of a typical system. The plant, G, is the device to be controlled. A controller, K, produces the control signal, u, used to modify the behavior of the plant. In this dissertation, we focus on a broad category of tasks known as *tracking tasks*: the controller must provide control signals so that the plant output, y, mimics an external, time-varying, input signal called the reference input, r. Performance is measured by the error signal, e, which is the difference between the reference signal and the plant output: $e = r - y$. We also require that the controller maintain system stability: for a finite and static reference signal, r, we require the system signals, u and y, remain finite and furthermore move asymptotically toward stable fixed points, $\bar{u}$ and $\bar{y}$. We are more precise about the notion of stability in subsequent chapters.

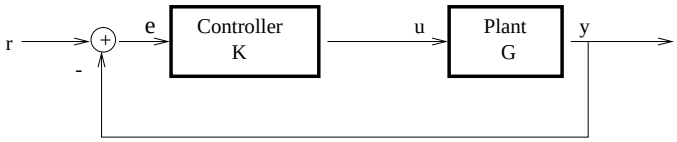


Figure 1.2: Nominal System

The vast majority of research in control theory applies to systems which are linear, time-invariant (LTI). The simple mathematics of LTI systems enables the application of the mature and extensive body of linear systems theory. Consequently, the design of stable controllers is straightforward. However, the LTI conditions place restrictive limits on the class of controllers available for use.

A non-LTI controller is often able to achieve greater performance because it is not saddled with the limitations of LTI. Two classes of non-LTI controllers are particularly useful for control; nonlinear controllers implement a wider range of control functions, and adaptive controllers self-modify to better match the system characteristics. However, nonlinear and adaptive controllers are difficult, and often impossible, to study analytically. Thus, the guarantee of stable control inherent in LTI designs is sacrificed for non-LTI controllers.

Neural networks, or neuro-controllers, constitute much of the recent non-LTI control research. Because neural networks are both nonlinear and adaptive, they often realize far superior control compared to LTI. However, dynamic analysis of neuro-controllers is mostly intractable thereby prohibiting control engineers from ascertaining their stability status. As a result, the use of neuro-controllers is primarily restricted to academic experiments; most industrial applications require guarantees of stable control which have not been possible with neural networks.

The stability issue for systems with neuro-controllers encompasses two aspects. *Static stability* is achieved when the system is proven stable provided that the neural network weights are constant. *Dynamic stability* implies that the system is stable even while the network weights are changing. Dynamic stability is required for networks which learn on-line in that it requires the system to be stable regardless of the sequence of weight values learned by the algorithm.

1.3 Objective

The primary objective of this work is to develop a framework in which we can ensure the stability of neuro-controllers. We then use this framework to prove the stability of both static and dynamic neuro-controllers. This objective is mainly a theoretical goal. While a few recent published results have shown some success with the static stability problem, we will provide a different proof that strengthens existing solutions. We also offer the first neuro-control scheme proven to solve the dynamic stability problem; the neuro-controller we propose is guaranteed to provide stable control even while the network is training.

As a secondary objective, we demonstrate that the theoretical stability proofs are practical to implement on real neuro-control systems tackling difficult control problems. Many theoretical results in artificial intelligence are not amenable to practical implementation; often technical assumptions of the theorems are violated in order to construct "reasonable and working" systems in practice. We show the assumptions of our stability theorems do not place unreasonable limitations on the practical implementation of a neuro-control system.

To demonstrate that we have achieved these objectives we will provide the following.

- A formal proof of the stability of a neuro-controller with fixed weights (static neural network).
- A formal proof of the stability of a neuro-controller undergoing weight changes during learning (dynamic neural network).
- A neural network architecture and a learning algorithm suitable for use in a general class of control tasks.
- A series of case studies showing that the neuro-control architecture and stability proofs are amenable to practical implementation on several control tasks.

1.4 Approach

In an effort to achieve our primary goal of verifying the stability of static and dynamic neuro-controllers, we employ an approach that combines reinforcement learning and robust control. We draw upon the reinforcement learning research literature to construct a learning algorithm and a neural network architecture that are suitable for application in a broad category of control tasks. Robust control provides the tools we require to guarantee the stability of the system.

Figure 1.3 depicts the high-level architecture of the proposed system. Again, r is the reference input to be tracked by the plant output, y. The tracking error is the difference between the reference

signal and the plant output: $e = r - y$. A *nominal* controller, K, operates on the tracking error to produce a control signal u_c. A learning agent is included that also acts on the tracking error to produce a control signal $\hat{u}$. The two component control signals are added to arrive at the overall control signal: $u = u_c + \hat{u}$. Again, the goal of the controller(s) is twofold. The first goal is to guarantee system stability. The second goal is to produce the control signals to cause the plant to closely track the reference input over time. Specifically, this latter performance goal is to learn a control function to minimize the mean squared tracking error over time.

Importantly, the learning agent does not replace the nominal controller; rather, it adds to the control signal in an attempt to improve performance over the nominal LTI controller. This approach, retaining the LTI controller as opposed to replacing it, offers two advantages. First, if the neuro-controller fails, the LTI controller will still provide good control performance; the neuro-controller can be turned off without greatly affecting the system. Second, the control performance of the system is improved during the learning process. If the neuro-controller were operating alone, its initial control performance would most likely be extremely poor; the neural network would require substantial training time to return to the level of performance of the nominal LTI controller. Instead, the neuro-controller starts with the performance of the nominal controller and adds small adjustments to the control signal in an attempt to further improve control performance. The neuro-controller starts with an existing high-performance controller instead of starting tabula rasa.

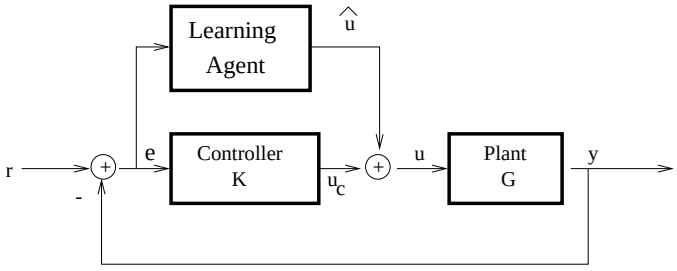


Figure 1.3: Nominal System with Learning Agent Controller

Because the learning agent is implemented with a neural network that contains non-LTI features, we must solve the static stability problem: we must ensure that a network with a fixed set of weights implements a stable control scheme. Since exact stability analysis of the nonlinear neural network is intractable, we need to modify the network to fit into the LTI framework. To accomplish this, we treat the nonlinear hidden units of the neural network as sector-bounded, nonlinear uncertainties. The techniques of robust control are developed around the concept of treating system nonlinearities as uncertainties. Thus, we can apply the techniques of robust control to determine the stability status of the neuro-controller. Specifically, we use either μ-analysis or IQC-analysis which are two robust control tools that determine the stability of systems with uncertainty. In this way, we solve the static stability problem.

Along with the nonlinearity, the other powerful feature of using a neural network for the learning agent is its adaptability; the agent can learn to provide better control. In order to accommodate an adaptive learning agent, we must solve the dynamic stability problem: the control system must be proven stable while the neural network is learning. To solve the dynamic stability problem we require two components of uncertainty. As we did in the static stability analysis, we use a sector-bounded uncertainty to cover the neural network's nonlinear hidden layer. Additionally, we add uncertainty in the form of a slowly time-varying scalar to cover weight changes during learning. Again, we apply μ-analysis and IQC-analysis to determine whether the network (with the weight uncertainty) forms a stable controller.

To understand the details of how we employ μ-analysis and IQC-analysis, envision the current neural network weight values as a point in the high-dimensional weight space of the network. By adding a small perturbation to each individual network weight, we form region around the current weight space point. We employ μ-analysis or IQC-analysis to determine the largest set of neural

network weight perturbations that the system can tolerate while still being stable. In effect, the region formed by the weight perturbations acts as a "safety region" in the network's weight space in which the learning algorithm can operate; any network weight values within this region produce stable control. We then apply standard reinforcement learning to adapt the network weights until they move outside the stable safety region. In this way, we solve the dynamic uncertainty problem for stability during network training.

In summary, our approach in meeting the primary objective, a theoretical result demonstrating the stability of a neural network controller, is to convert the neural network, with nonlinearity and the adaptive weight changes, to a linear, time-invariant form by using uncertainty regions. Once the network has been recast in the LTI form, then we apply the stability analysis tools of μ and IQC in order to determine the stability status of the neuro-control system.

Our secondary objective is to demonstrate the practical application of the stability theorems to challenging control problems. To accomplish this goal, we pursue two paths. In the first path, we design a suitable learning agent to address the following primary considerations: the selection of an appropriate learning algorithm, the construction of a suitable high-level architecture to fulfill the dual roles of controller and learning agent, and the design of a low-level architecture that satisfactorily accomplishes the first two considerations. The second path that we pursue is to present case studies demonstrating the application of our theory to four control tasks. The intent of these case studies is to illustrate the application of the static and dynamic stability theorems to practical control situations; these case studies are not intended to be an empirical analysis comparing this approach with other control algorithms.

In the first path outlined above, the design of a suitable learning agent, we address three primary considerations. The first consideration is selecting a learning algorithm for the agent. The choice of a learning algorithm is mostly orthogonal to the constraints of robust stability; we have considerable freedom in selecting a learning algorithm that is geared primarily to control performance. We chose reinforcement learning for our learning agent, because it is well suited to the limited information of the system (a performance metric) and the algorithm also is ideal at optimizing functions over extended time horizons. Reinforcement learning implements the trial-and-error search required to find good control functions.

The second consideration is designing a high-level architecture to accommodate the dual role of the learning agent. The agent must act as a controller by providing real-time control signals in response to tracking error input signals and must act as a reinforcement learner by accumulating value functions and using them to adjust the control policy. To fulfill the possibly conflicting requirements of each role, we turn to a dual network architecture known as the actor-critic design. We find this arrangement is suitable for not only balancing the demands of our learning algorithm and control function, but also for the analysis required by robust stability.

The third consideration in designing a learning agent is designing a low-level neural architecture to over come neurodynamic problems that occur with weight changes during learning. Reinforcement learning agents can be implemented in a variety of representations. The selection of a representation affects the performance of the agent in simulations and real-world applications. We discuss specific neurodynamic problems we encounter and discuss how specific neural network representations overcome these difficulties.

The second path is to demonstrate the applicability of our robust, stable, reinforcement learning agent by using the agent in a series of example case studies. We select four example control problems for our case studies. The first two control tasks are trivial from a control design standpoint, but they allow the reader to fully understand the dynamics of the system and to compute the desired optimal control law. These two tasks serve primarily to illustrate the application of the theory to control problems. The third example case study involves a simulated distillation column. This control problem offers sufficient complexity to warrant state of the art control solutions. We apply our stable neuro-controller to this task to illustrate the ease of application to challenging control problems and to demonstrate that the stable neuro-controller is able to realize performance improvements over robust controllers. The final example control task involves a model of an HVAC heating coil. We use this case study to demonstrate that the HVAC domain is a solid candidate for the application of our stable neuro-controller, but that an improper application could limit the effectiveness of a

stable neuro-controller.

1.5 Contribution and Significance

The work in this dissertation has significant implications for the control community. We provide a new approach to proving the stability of a fixed-weight neural network. Most importantly, our methodology is the first that guarantees stability during the network training process. With stability guarantees for neuro-control, the control community can utilize the adaptive nonlinear power of neural network controllers while still ensuring stability in critical control applications. We also contribute a neural architecture and learning agent design to overcome numerous, non-trivial, technical problems. Much of the previous theoretical work in neuro-control does not address implementation details, which can render the theory inapplicable. In this dissertation, we develop a learning agent that is suitable for application, and we provide detailed analysis of the stable reinforcement learning agent as it tackles four difficult control problems. In this section, we discuss each of these contributions in more detail.

The first contribution is our solution to the static stability problem: given a neural network controller with fixed weights, we have developed a method which conclusively proves the stability of the controller. A few other research groups have arrived at similar results using other approaches. The most significant of these other static stability solutions is the NLq research group of Suykens and DeMoor [Suykens and Moor, 1997]. Our approach is similar to the NLq group in the treatment of the nonlinearity of the neural network, but we differ in how we arrive at the stability guarantees. Our approach is also graphical and thus amenable to inspection and change-and-test scenarios.

By far, our most significant contribution is a solution to the dynamic stability problem. Our approach is the first to guarantee the stability of the neuro-controller while the network is experiencing weight changes during learning. We extend the techniques of robust control to transform the network weight learning problem into one of network weight uncertainty. With this key realization, a straightforward computation guarantees the stability of the network during training.

An additional contribution is the specific architecture amenable to the reinforcement learning / control situation. As already mentioned, we build upon the early work of actor-critic designs as well as more recent designs involving Q-learning. Our dual network design features a computable policy (this is not available in Q-learning) which is necessary for robust analysis. The architecture also utilizes a discrete value function to mitigate difficulties specific to training in control situations; we demonstrate its effectiveness in our four case studies.

The work in this dissertation paves the way for further research in adaptive neuro-control. This initial solution to the static and dynamic stability problems is a large step forward in allowing industry to utilize neuro-controllers in their products.

1.6 Overview of Dissertation

This dissertation synthesizes two diverse bodies of research. From the artificial intelligence community we use reinforcement learning and neural networks. From the control community, we employ the recently developed robust control theory. Here we outline the major components of each chapter in this dissertation.

In Chapter 2, we present an overview of the research literature contributing to reinforcement learning and robust control.

Chapter 3 introduces the various concepts of stability. This chapter progresses from the simple mathematical definitions of stability toward the more complex notions of system stability. Although the details are lengthy, we present the entire progression as it is fundamental to the stability proofs in later chapters. We start with several key definitions of stability and different interpretations for each definition. Because the stability definitions are typically not applicable for complex systems, we also introduce Liapunov's direct method of ascertaining system stability. These concepts are then applied to system stability. We discuss robust stability before concluding the chapter with a brief discussion of μ-analysis and IQC-analysis theories.

In Chapter 4, we present our solution to the static and dynamic stability problems. The chapter begins by introducing the stability problems and motivates the need for a solution. The chapter then splits into two parallel lines of development; the first line assumes that μ-analysis is used as the robust stability tool while the second line assumes that IQC-analysis is the robust stability tool. In each line, we provide the details of how to convert a nonlinear neural network into an LTI system. By covering the nonlinearity of the neural network with uncertainty, we can apply the tools of robust control to arrive at static stability guarantees. Chapter 4 then extends our solution of the static stability problem to overcome the dynamic stability problem. We discuss how to add additional uncertainty to the neural network weights. This allows us to define a safety region in the network weight space; the safety region permits the reinforcement learning algorithm to adjust the neural network weights while maintaining a guarantee of stable control. The chapter concludes with a sketch of the stable reinforcement learning algorithm general enough to recombine the parallel lines of μ-analysis and IQC-analysis.

Chapter 5 details the design of the learning agent. We start by examining why the reinforcement learning algorithm is better suited to the control domain than other learning algorithms. Our scheme has an important requirement in that the agent must play a dual role as both a reinforcement learner and as a controller. We devise a high-level architecture to resolve difficulties in this dual role. We then discuss the design of the low-level architecture by considering various choices of particular neural networks to be used in the learning agent. The network-as-controller introduces difficult neuro-dynamic problems not typically encountered in neural network training or in reinforcement learning.

We then put the algorithm and stability proofs to the test in Chapter 6. Here we present four example control problems as case studies in the application of the stability proofs to real control problems. Importantly, the intent of this chapter is not to empirically compare our stable reinforcement learning method with other control strategies; instead, the purpose of Chapter 6 is to demonstrate that our theoretical contributions are applicable to real-life control situations without violating the presuppositions of the stability proofs. While trivial from a control perspective, the first two control tasks are simple enough for the reader to easily visualize the application of static and dynamic stability to the control design. We also apply our learning agent to a complex distillation column process. This example serves to illustrate the necessity of robust control over optimal control techniques and also to demonstrate how the neuro-control agent is able to regain control performance lost to the robust control design. The HVAC (Heating, Ventilation and Air Conditioning) research community is currently examining ways to employ neuro-controllers in their field. The complex dynamics of heating coils, cooling towers, building systems and other HVAC systems offer difficult control tasks that represent the cutting edge in neuro-control research. We tackle the control of a heating coil to understand how the the theory and techniques developed in this work are applied to this domain.

Chapter 7 summarizes the dissertation, iterates our contributions to the fields of reinforcement learning and control, discusses the the successes and difficulties of our approach, and finally introduces some avenues of future research in the arena of stable neuro-control. Appendix A provides a brief tutorial on the use of the μ and IQC tools. Appendix B lists the code used in the case studies.

Chapter 2

Literature Review

In this chapter, we review the significant contributions in the research literature to neuro-control, reinforcement learning, and robust control. These key papers serve as the basis for the theoretical and experimental advances outlined in this dissertation. We identify several historically important papers and also discuss recent papers with direct relevance to our goal of a stable neuro-control algorithm.

2.1 Robust Control

Despite the fact that the techniques of robust control are relatively new, there are a large number of recent publications in this field. We identify a few key researchers and their seminal papers which set the stage for the current success of robust control theory. Among the earliest works in robust control is the stability theory of the 1970s. This early work built upon the mature body of research in linear systems to extend stability theorems for systems with a very specific and limited set of nonlinear components. In particular, the development of the circle criterion and the small gain theorem in works by Zames [Zames, 1966], Desoer and Vidyasagar [Desoer and Vidyasagar, 1975], and Vidyasagar [Vidyasagar, 1978] provide sufficient conditions to prove the stability of systems with nonlinear elements in the feedback path.

Another major step introduces uncertainty to handle systems with general nonlinear components. The novel idea in this work is to structure the type of uncertainty and then provide stability theorems for any nonlinearities in the system meeting the criteria of the structured uncertainty. The advent of a structured singular value metric, μ, is of paramount importance to robust control. Doyle [Packard and Doyle, 1993] pioneered much of the early work in robust control and provided the connection between μ and structured uncertainty. Also important is the formal development of the general LFT (Linear Fractional Transform) framework along with advances in computing LMIs (Linear Matrix Inequalities) with polynomial time algorithms [Packard and Doyle, 1993]. Young [Young and Dahleh, 1995; Young, 1996] extends μ-analysis to other types of uncertainty including parametric uncertainty. Notable progress in robust control also includes the availability of commercial software for performing the complex computations required [Balas et al., 1996]. Matlab's μ-Tools toolbox makes robust control theory accessible to the control engineer without investing years to translate difficult theory into practical code.

2.2 Traditional Adaptive Control

While actual physical plants contain dynamics which are not LTI (linear, time-invariant), the analysis of non-LTI systems is mostly intractable. Much of modern control theory is based upon linear, time-invariant (LTI) models. Because LTI dynamics are not as rich in their functional expression, control performance is often sacrificed by the limitations imposed by LTI dynamics. One of the early attempts by the control community to seek alternatives to LTI control design is adaptive control.

Franklin [Franklin and Selfridge, 1992] gives the following definition of adaptive control: "Adaptive control is a branch of control theory in which a controlled system is modeled, typically by means of a set of linear difference or differential equations, some of whose parameters are unknown and have to be estimated". The primary purpose of adaptive control is to form a model of a physical plant by adapting parameters in the model. Adaptive control typically proposes a *model structure* for the plant a priori; the parameters of this model are altered. By presupposing a model structure, the adaptive control scheme is limited in its representational flexibility [Franklin and Selfridge, 1992]. Here we introduce the general approach by describing two popular and representative adaptive control schemes.

One of the simpler attempts at traditional adaptive control is the Self-Tuning Regulator (STR) [Astrom and Wittenmark, 1973]. The STR uses an equation with unspecified parameters to model the system. The parameters are updated from on-line sampling to better fit empirical data on the system. As the parameters are updated, the controller is re-tuned to provide better control for the updated system model [Franklin and Selfridge, 1992]. The major advantage of this approach is the STR's predication on the rich and well developed theory of least-squares parameter estimation. Namely, tight bounds on the system model error can be computed readily. Major drawbacks include unguaranteed system stability, especially during initial learning of the system model parameters, and a requirement for a priori knowledge of the correct structure for the system equations.

A more sophisticated approach to adaptive control is the Model Reference Adaptive Controller (MRAC) [Parks, 1966]. The MRAC uses an externally supplied *ideal* model of the closed loop system which exhibits the desired characteristics. The parameters of the controller are updated dynamically in an attempt to make the closed-loop system act like the reference model. The adaptive algorithm is posed as a Liapunov function; this has the advantage that the output error is bounded and is asymptotically stable. That is, the output of the closed-loop system will eventually move to track the reference model with non-increasing error. Again, a major disadvantage is that the structure of the reference model must match the system dynamics. Additionally, the reference model is limited to being linear in order for the Liapunov stability analysis to hold.

2.3 Neuro-control

Neuro-control originated as a special branch of adaptive control. Originally, neural networks were employed as adaptive control agents to model the dynamics of a plant [Kalkkuhl et al., 1997]. However, neural networks are much broader than the specific models of traditional adaptive control and their use quickly spread to other aspects of control theory.

The application of connectionist computation (neural networks) to the area of control is not new. Much of the past research can be categorized into a few distinct approaches; we review the most common applications of neural networks to control. It is particularly noteworthy that most of these approaches are based upon supervised learning. Reinforcement learning, having been developed a decade later than supervised learning, remains mostly the focus of the AI community. We review reinforcement learning approaches to neuro-control in Section 2.5.

System Identification

As stated above, among the earliest applications of neural networks to control is system identification which is also known as parameter estimation by the optimal control community [Werbos, 1992; Barto, 1992]. Essentially, the neural network is trained to imitate the plant as shown in Figure 2.1.

System identification, in the manor of adaptive control, is useful for plant classification. Suppose that we possess a control solution that is effective for a particular class of plants – plants which possess a certain dynamic form. We construct a neural network model that is capable of learning this form. We then train the network on the unknown physical plant using supervised learning. If we can reduce the approximation error below a prespecified tolerance, then we conclude that the neural network is capable of approximating this plant. Thus, the plant must possess dynamics which belong to this particular class of plants. We then apply our known control solution because the control solution is effective for all plants of this class. Conversely, if the neural network is not

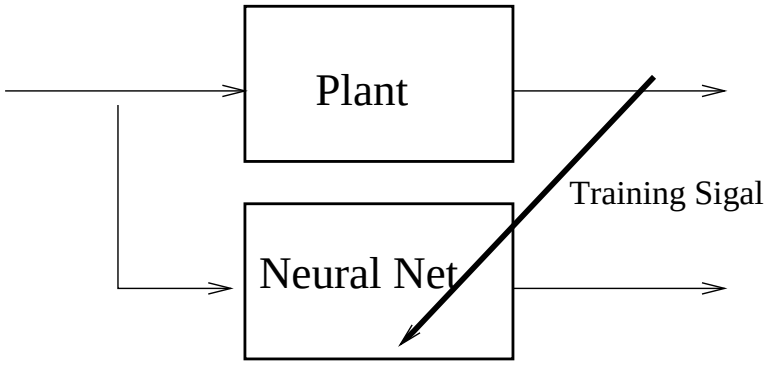


Figure 2.1: Neuro-control: System Identification

able to reduce the approximation error below the tolerance level, then the plant in question does not belong to the known class of plants. We cannot apply our control solution.

While system identification provided the early bridge from pure adaptive control to neuro-control, this method's primary utility became one of providing a training signal for a double neural network arrangement. This arrangement is discussed shortly.

Imitate an Existing Controller

Also among the early applications of neural networks to control is modeling an existing controller. This arose from early neural network research involving the newly "rediscovered" back propagation algorithm [Werbos, 1974; Rumelhart et al., 1986a] and the neural network's keen ability to model a nonlinear function. The network can be trained to imitate any nonlinear mapping given sufficient resources (hidden units) [Hassoun, 1995]. The architecture for such an arrangement is sketched in Figure 2.2. The neural network receives the same inputs as the controller and attempts to produce the same outputs. The error is back propagated through the net to adjust the weights.

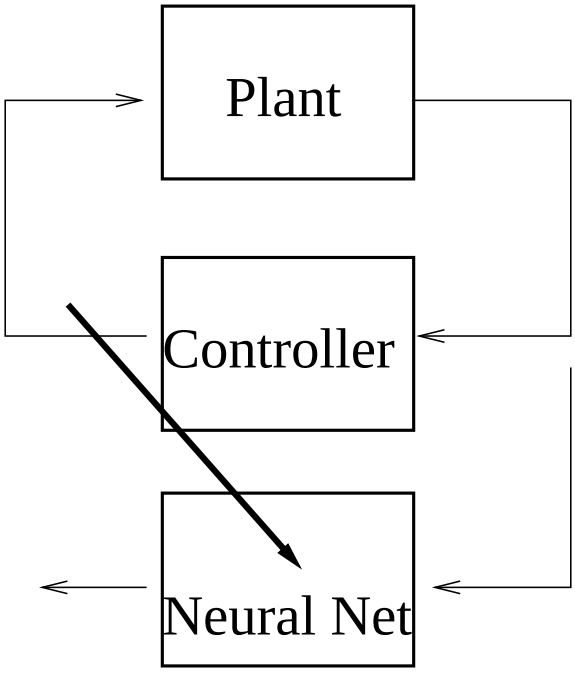


Figure 2.2: Neuro-control: Imitate Existing Controller

An obvious question arises as to the utility of training such a network if we already have an existing controller. There are several reasons why we would require the neuro-controller. The existing controller might be impractical or expensive. An example is a controller that is human; the network is trained to mimic a person's control decisions. A second reason is that the neural controller may be easier to analyze; it might be possible to gain insights to a "control rule" [Barto, 1992]. Training a neural network using an existing controller is also useful as a method to "seed" the network off-line. Once the network mimics the controller, it can be placed on-line and make intelligent control decisions immediately. The learning process continues so that the network potentially surpasses the existing controller's performance. However, this requires more than the supervised training algorithm; it requires the techniques of reinforcement learning.

System Inverse Identification

The neural network is trained to model the inverse dynamics of the plant. The network receives the plant's output and attempts to reproduce the plant inputs accurately. This arrangement is depicted in Figure 2.3. Naturally, this approach is limited if the plant is not invertible as is the case when the forward plant function is not injective [Barto, 1992].

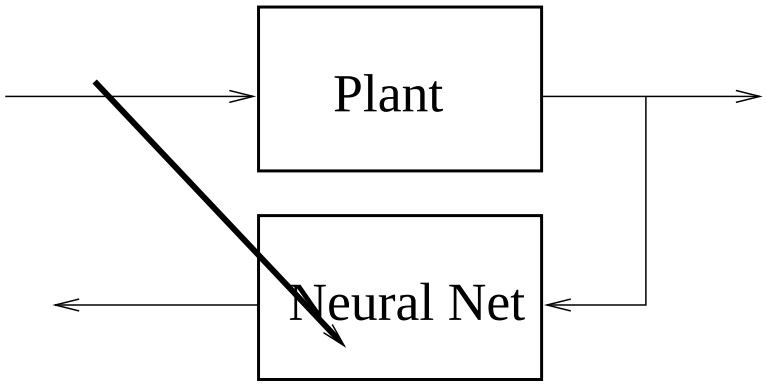


Figure 2.3: Neuro-control: Learn an Inverse Plant

Many control problems are reference tracking problems in which the plant output attempts to track a reference input signal. Having the plant inverse as a controller obviously implements this function nicely. This arrangement is shown in Figure 2.4. However, in practice there is often difficulty in learning a well-formed plant inverse. Subtle errors and new input vectors can cause the inverse controller to produce widely different responses [Barto, 1992]. Also, the addition of a feedback loop complicates the practicality of such a controller. Kawato [Kawato, 1992] makes extensive use if this architecture in his *feedback error training* method. Instead of using the actual plant output as an input to the network, Kawato substitutes the desired plant response.

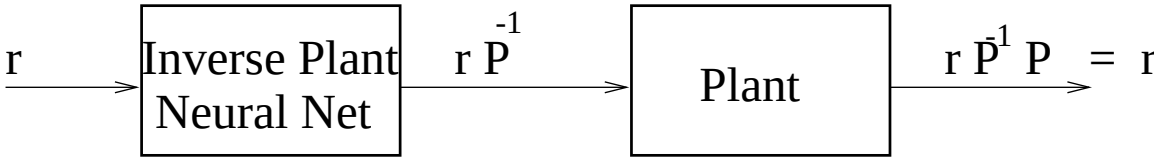


Figure 2.4: Neuro-control: Inverse Plant as Controller

Differential Plant

This method begins with the same basic arrangement of a neural network used for system identification as in Figure 2.1. We train one neural network to mimic the forward plant. As shown in Figure 2.5, we include a second neural network as a controller. The immediate benefit of such

an arrangement is that we can now train the controller off-line (i.e. we can use the plant-network instead of the plant). But the benefit is more profound than this.

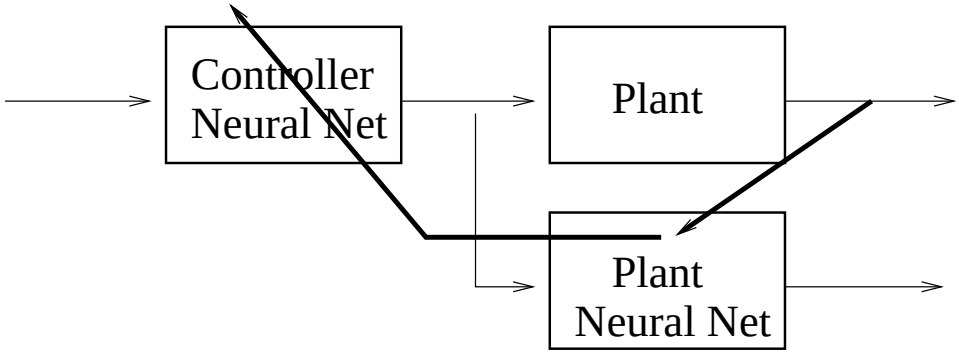


Figure 2.5: Neuro-control: Differential Plant

The problem of using a neural network directly as a controller is that the neural network is trained using supervised learning. That is, the neural network acts upon a plant state (usually the tracking error) to produce a control signal for the plant. In order to train the network, we must know the correct, or optimal, control signal that the network should have produced. It is often extremely difficult to examine the plant's output response and compute what the optimal control signal should have been. We require a method which relates the tracking error at the plant output with the control signal at the controller output (plant input).

With the dual neural network differential plant arrangement, we now have a mathematical model of the plant which is differentiable. When an reference input signal is applied to the neural controller, it produces a control signal for the plant. The plant acts upon the control signal to produce a plant-output. We can now use the error between the actual plant output and the desired plant output to back propagate the error through the network-plant to control signal. Jordan shows that this arrangement is equivalent to the transpose of the plant's Jacobian evaluated at the control signal (plant input) [Jordan, 1988]. The control signal error can now be used to train the controller network. By using this "double back propagation" method we can train the neural controller with supervised learning. The advent of reinforcement learning largely obviates a need for this arrangement. However, double back propagation still has uses for special neural architectures; we borrow the concept for use in training our neuro-controller.

2.4 Stable, Robust Neuro-control

Despite the relative frequency with which "robust", "stable", and "neural network" appear in paper titles populating conference proceedings and journal publications, there exist relatively few attempts at devising a truly stable neural network controller. Much of the confusion arises from the many interpretations given to the terms "robust" and "stable". In the AI community, stability implies many different concepts including: sensitivity to fault tolerance, relative ease of learning, and the dynamics of weight changes. Even in the control community, robustness assumes many interpretations; robustness generally means insensitivity to something. The problem is that there are many "somethings" to be insensitive toward. In this dissertation, robustness describes a controller that is insensitive to differences in the plant model and physical plant. We use the term stability to denote the formal mathematical definition of stable control presented in Chapter 3.

Along these lines, there have been very few true attempts to construct neuro-controllers which are stable. There have been even fewer attempts to add the robustness criteria. As we shall discuss thoroughly in subsequent chapters, the dearth of successful research in this area is due to the inherent intractability of analyzing a neural network's dynamic contribution to a control system. Despite the lack of directly relevant publication in this area, there are a few noteworthy efforts we outline here.

The unique, direct approach of Slotine and Sanner [Sanner and Slotine, 1992] utilizes hardware-realizable, analog, neural networks to implement a real-time, stable, adaptive controller for continuous-time, nonlinear plants. To achieve this result, they do require certain a priori assumptions about the class of nonlinearities which characterize the plant. Slotine and Sanner present a proof based upon a Liapunov stability criterion: the neural network architecture is stable in the sense that the control error will converge monotonically to a minimum.

This type of "robust stability proof" is common in the earlier work in stable neuro-control. The Liapunov stability analysis (or Liapunov equation) is used to show that network learning algorithms cause the network outputs to asymptotically converge upon a stable solution. We outline Liapunov stability analysis in Section 3.3. However, this type of proof is not robust in the modern sense of robust control concerning the differences between physical plant dynamics and plant model dynamics.

The approach of Bass and Lee [Bass and Lee, 1994] is noteworthy for two reasons. They propose an architecture in which the neural network is treated as unstructured plant uncertainty, and they also attempt to limit the magnitude of the neural network by having it learn only the nonlinear component of the inverse plant. Their approach uses the neural network to *linearize* a nonlinear plant. Thus, the neural network does not provide any control; simply, the neural network is used to learn the "uncertainty" between the true plant and an LTI model of the plant. Then, a standard robust controller is designed to control this plant.

Bass and Lee attempt to reduce the magnitude of the neural network weights by reducing the uncertainty that the network represents. To accomplish this, they first compute a linear model of the plant (off-line) using standard control techniques. Then, the neural network is trained to compute the difference between the linear model and the plant. Once the network is trained, then they can compute a bound on the magnitude of the neural network. They combine this bound with a variant of the small gain theorem (see Section 3.5) to prove stability. Bass and Lee do provide among the first discussions of how to adapt the network's learning algorithm to achieve a bounded norm on the network output. The work in this dissertation is a significant step forward in that we use the neural network to provide direct control signals. We employ the full nonlinear, adaptive, power of the neural network to realize improved control.

Narendra and Levin [Levin and Narendra, 1993] propose a neuro-control scheme involving *feedback linearization*. An LTI model of a nonlinear plant is devised. Then, the neural network is used in several different ways to stabilize the system about a fixed point. The stability guarantees are only valid for local trajectories about the fixed point. The local validity region may be arbitrarily small. Furthermore, their method relies upon substantial state information that may not be observable from the plant.

The *NLq* research group of Suykens, De Moor, Vandewalle and others [Suykens et al., 1996; Suykens and Moor, 1997; Suykens et al., 1997; Verrelst et al., 1997; Suykens and Bersini, 1996; Suykens et al., 1993b; Suykens et al., 1993a; Suykens et al., 1995] is among the most well-developed efforts at stable, robust neuro-control. An NLq representation is a series of alternating linear and nonlinear computational blocks. The format is generic enough to include most classes of neural networks as well as many other architectures such as parameterized systems and Kalman filters. NLq is significant in its formal generalization of categorizing neural architectures in a framework that is suitable for application to control theory.

The second contribution of NLq theory is a series of proofs demonstrating the internal stability of systems containing NLq components. The authors present three variations of the proof, all of which are based upon solving convex optimization problems known as LMIs (linear matrix inequalities). The LMIs are configured to arrive at stability via a quadratic Liapunov-type function. The NLq team is also among the first to attempt to devise a neural network learning algorithm based upon stability constraints. They alter Narendras's *Dynamic Back Propagation Algorithm* [Narendra and Parthasarathy, 1990] to provide stability assurances for their neuro-controller. However, their algorithm only provides "point-wise" stability assurances: each network weight value obtained during learning implements a stable system. But, the *dynamic* system, in which the network weights change, is not guaranteed to produce stable control. In this dissertation, we distinguish the two approaches as the static stability problem and the dynamic stability problem. Furthermore, the supervised learning algorithm of the NLq group does not utilize the excellent advantages of rein-

forcement learning. Still, their work is clearly the leading pioneering effort in the field of stable neuro-control.

2.5 Reinforcement Learning for Control

In this section we review the most significant contributions of reinforcement learning with emphasis on those directly contributing to our work in robust neuro-control. Sutton and Barto's text, *Reinforcement Learning: An Introduction* presents a detailed historical account of reinforcement learning and its application to control [Sutton and Barto, 1998]. From a historical perspective, Sutton and Barto identify two key research trends that led to the development of reinforcement learning: the trail and error learning from psychology and the dynamic programming methods from mathematics.

It is no surprise that the early researchers in reinforcement learning were motivated by observing animals (and people) *learning* to solve complicated tasks. Along these lines, a few psychologists are noted for developing formal theories of this "trial and error" learning. These theories served as spring boards for developing algorithmic and mathematical representations of artificial agents learning by the same means. Notably, Roger Thorndike's work in *operant conditioning* identified an animal's ability to form associations between an action and a positive/negative reward that follows [Thorndike, 1911]. The experimental results of many pioneer researchers helped to strengthen Thorndike's theories. Notably, the work of Skinner and Pavlov demonstrates "reinforcement learning" in action via experiments on rats and dogs respectively [Skinner, 1938; Pavlov, 1927].

The other historical trend in reinforcement learning arises from the "optimal control" work performed in the early 1950s. By "optimal control", we refer to the mathematical optimization of reinforcement signals. Today, this work falls into the category of dynamic programming and should not be confused with the optimal control techniques of modern control theory. Mathematician Richard Bellman is deservedly credited with developing the techniques of dynamic programming to solve a class of deterministic "control problems" via a search procedure [Bellman, 1957]. By extending the work in dynamic programming to stochastic problems, Bellman and others formulated the early work in Markov decision processes.

Barto and others combined these two historical approaches in the field of reinforcement learning. The reinforcement learning agent interacts with an environment by observing states, s, and selecting actions, a. After each moment of interaction (observing s and choosing a), the agent receives a feedback signal, or reinforcement signal, r, from the environment. This is much like the trial-and-error approach from animal learning and psychology. The goal of reinforcement learning is to devise a control algorithm, called a *policy*, that selects optimal actions (a) for each observed state (s). By optimal we mean those actions which produce the highest reinforcements (r) not only for the immediate action, but also for future actions not yet selected. The mathematical optimization techniques of Bellman are integrated into the reinforcement learning algorithm to arrive at a policy with optimal actions.

A key concept in reinforcement learning is the formation of the value function. The value function is the expected sum of future reinforcement signals that the agent receives and is associated with each state in the environment. Thus $V(s)$ is the value of starting in state s and selecting optimal actions in the future; $V(s)$ is the sum of reinforcement signals, r, that the agent receives from the environment.

A significant advance in the field of reinforcement learning is the Q-learning algorithm of Chris Watkins [Watkins, 1989]. Watkins demonstrates how to correctly associate the the value function of the reinforcement learner with both the state and action of the system. With this key step, the value function can now be used to directly implement a policy without a model of the environment dynamics. His Q-learning approach neatly ties the theory into an algorithm which is both easy to implement and demonstrates excellent empirical results.

Reinforcement learning algorithms must have some construct to store the value function it learns while interacting with the environment. These algorithms often use a function approximator to store the value function; the performance of the algorithm depends upon the selection of a function approximation scheme. There have been many attempts to provide improved control of a reinforcement

learner by adapting the function approximator which learns/stores the Q-value function. Anderson adds an effective extension to Q-learning by applying his "hidden restart" algorithm to the difficult pole balancer control task [Anderson, 1993]. Moore's Parti-Game Algorithm [Moore, 1995] dynamically builds an approximator through on-line experience. Sutton [Sutton, 1996] demonstrates the effectiveness of discrete local function approximators in solving many of the neuro-dynamic problems associated with reinforcement learning control tasks. We turn to Sutton's work with CMACs (Cerebellar Model Articular Controller) to solve some of the implementation problems for our learning agent. Anderson and Kretchmar have also proposed additional algorithms that adapt to form better approximation schemes such as the Temporal Neighborhoods Algorithm [Kretchmar and Anderson, 1997; Kretchmar and Anderson, 1999].

Among the notable research of reinforcement learning is the recent success of reinforcement learning applications on difficult and diverse control problems. Crites and Barto successfully applied reinforcement learning to control elevator dispatching in large scale office buildings [Crites and Barto, 1996]. Their controller demonstrates better service performance than state-of-the-art, elevator-dispatching controllers. To further emphasize the wide range of reinforcement learning control, Singh and Bertsekas have out-competed commercial controllers for cellular telephone channel assignment [Singh and Bertsekas, 1996]. There has also been extensive application to HVAC control with promising results [Anderson et al., 1996]. An earlier paper by Barto, Bradtke and Singh discussed theoretical similarities between reinforcement learning and optimal control; their paper used a race car example for demonstration [Barto et al., 1996]. Early applications of reinforcement learning include world-class checker players [Samuel, 1959] and backgammon players [Tesauro, 1994]. Anderson lists several other applications which have emerged as benchmarks for reinforcement learning empirical studies [Anderson, 1992].

Chapter 3

Stability Theory Overview

As the primary goal of this dissertation is to guarantee stability for systems with neuro-controllers, it is important to rigorously develop the various concepts of stability. If a neural network is to be incorporated as part of the control scheme, we must ascertain the requirements on the neuro-controllerk such that signals propagating through the system do not "blow up" to infinity for reasonable system inputs. In this chapter, we present a thorough overview of the various aspects of stability with special emphasis on the components of stability theory contributing to the stability proofs for our neuro-controllers in LTI systems.

This chapter serves two purposes. First, we collect the definitions and theorems that we will require to prove neuro-control stability. We select and organize those concepts germane to neuro-controllers and present them in a coherent frame of reference. The secondary objective of this chapter is to provide a high-level overview of stability. For theoreticians trained in classical electrical engineering, much of this material can be skimmed. For the mathematically literate non-engineer, this chapter provides enough of the fundamental parts of stability theory necessary for understanding the proofs we present for neuro-control stability. Numerous references are cited so that interested readers may pursue a more comprehensive treatment of the material.

In many ways, stability theory is much like the three blind men who touch various parts of an elephant and each form widely different conclusions about what must constitute an elephant. Similarly, we can approach stability theory from a number of different ways, each of which seems to be uniquely distinct from the others. However, because each approach to stability theory concerns with the same fundamental definitions, each of these different approaches can be shown to be roughly equivalent. Despite their equivalence, they do offer a different view of the stability of dynamic systems; each approach has advantages over the others in some situations. Thus, this chapter will present a number of the more common treatments of stability. Where appropriate, we will demonstrate the equivalence of these approaches or will direct the reader to references showing the equivalence.

The first three sections of this chapter introduce different, but equivalent, interpretations of the basic stability definitions. Section 3.2 lists the fundamental definitions that form the basis of stability theory. An equivalent but alternative formulation is provided by Liapunov's direct method introduced in Section 3.3. While definitional stability and Liapunov stability offer an "internal view" of stability, a third interpretation in Section 3.4, referred to as BIBO stability, presents stability from a system input/output perspective.

The final three sections examine feedback systems; the topics here are fundamental to standard control theory. In Section 3.6, we present nominal stability for feedback systems with pure LTI components. Section 3.7 builds upon feedback stability to arrive at the robust stability of systems with non-LTI components. We present two techniques within robust control to arrive at stability guarantees for non-LTI systems: μ-analysis techniques of robust control are based upon the structured singular value while integral quadratic constraints (IQC) derive from Liapunov theory.

3.1 Dynamic Systems

We begin by describing a generic continuous-time dynamic system. All of the results in this chapter are extendible to discrete-time dynamic systems; in fact, the implementation of a neural network controller will be on a digital computer. Linear dynamic systems are easily converted between the continuous-time and discrete-time domain. Additionally, continuous-time is the standard way of presenting stability theory, thus we will follow that convention here. There are several excellent texts on introductory control theory. Most of the definitions in this section are paraphrased from [Rugh, 1996; Vidyasagar, 1978; Desoer and Vidyasagar, 1975]; exceptions to these references are cited directly in the text.

Consider the general system specified by an n-dimensional state vector, $x(t)$, accepting an m-dimensional input vector $u(t)$, and producing the ℓ-dimensional output vector $y(t)$. The dynamics of such a system are described by the set of differential equations:

$$\dot{x}(t) = f_1[t, x(t)] + g_1[t, u(t)] \tag{3.1}$$
$$y(t) = f_2[t, x(t)] + g_2[t, u(t)] \tag{3.2}$$

where f_1, f_2, g_1, and g_2 are all continuous vector functions. To steer toward our goal of neuro-dynamic control stability, we immediately restrict ourselves to studying a subset of this most general system. Namely, we concentrate on *autonomous* systems, often referred to as *time-invariant* systems, in which the functions f_1, f_2, g_1, and g_2 do not depend upon t. Also we restrict ourselves to *strictly proper* systems in which $g_2 = 0$. Thus, we arrive at:

$$\dot{x}(t) = f_1[x(t)] + g_1[u(t)] \tag{3.3}$$
$$y(t) = f_2[x(t)] \tag{3.4}$$

A vast majority of the research in stability theory involves *linear* systems. Traditionally, the study of nonlinear system stability has been an exercise in theoretical mathematics; the intractable nature of nonlinear system analysis yields little practical guidelines for constructing stable nonlinear systems. Since linear system analysis is tractable, a significant body of applied research has been developed for such systems. In the remainder of this section, we concentrate primarily upon *linear, time-invariant* (LTI) systems, returning to nonlinear systems only to note contrasts.

A linear, time-invariant system is governed by the differential equations:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{3.5}$$
$$y(t) = Cx(t) + Du(t) \tag{3.6}$$

in which A, B, C, and D are constant matrices of the appropriate dimensions and $D = 0$ for strictly proper systems. This situation is depicted graphically in Figure 3.1.

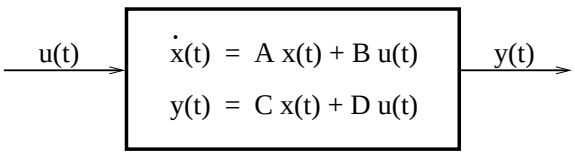


Figure 3.1: LTI Continuous-time System

At this point we should make a brief comment about the existence and uniqueness of solutions to (3.5) where a solution is an explicit function for $x(t)$. In seeking these solutions, we primarily concentrate on the "zero-input" case given by:

$$\dot{x}(t) \quad = \quad Ax(t) \tag{3.7}$$

The general solution to (3.7) is of the form $x(t) = \Phi(t, t_0)x_0$, where $\Phi(t, t_0)$ is known as the *homogeneous solution* and also the *state transition matrix*. For the linear case given by (3.7) solutions typically exist and are unique. The mathematically interested reader is directed to [Rugh, 1996] for a complete analysis of the uniqueness and existence of $\Phi(t, t_0)$.

Returning to the full input/output system of (3.5), we arrive at the complete solution given in (3.8). Notice the solution is linear in x_0 and $u(t)$, and is composed of a *zero-input response* given as the first term on the right hand side and a *zero-state response* in the integral portion of the right hand side.

$$x(t) \quad = \quad C\Phi(t, t_0)x_0 + \int_{t_0}^{t} C\Phi(t, t_0)Bu(\tau)d\tau \tag{3.8}$$

In most systems of practical interest, the solution given in (3.8) cannot be computed analytically. Often we can recover an approximate solution via numerical methods; however, any conclusions we then draw regarding stability are not absolute. Despite these difficulties, we can still propose some useful stability properties by examining the state matrix A. In the next section, we introduce various definitions of stability and look as the trivial cases in which we can verify stability.

3.2 Basic Stability Definitions and Theorems

Again, let us consider the linear, time-invariant, continuous time dynamic system with zero-inputs. This system is restated here in (3.9):

$$\dot{x}(t) = Ax(t) \tag{3.9}$$

Definition 1 *We say that $\bar{x}$ is an **equilibrium point** of (3.9) if*

$$\dot{x}(t) = A\bar{x} = 0 \tag{3.10}$$

We can assume that this equilibrium point is the zero vector ($\bar{x} = 0$) without loss of generality, because we can perform a simple change of coordinates to produce an equivalent system in which $\bar{x} = 0$ without affecting the stability analysis. This shift simplifies the definitions and proofs concerning stability.

Thus, for a system started in state $x(t_0) = x_0 = 0$ at time t_0, we have that $x(t) = 0$ for all t. An interesting question to ask is what happens if the system is started in a state that is some arbitrarily small deviation away from $\bar{x}$? Or if the system is currently at state $\bar{x}$ and is then perturbed to some nearby state, what is the resulting trajectory of $x(t)$? If the system is "stable", we would like to know that the system will settle into the equilibrium state or, at least, that the system will not move arbitrarily far away from the equilibrium point (will not blow-up). Before we jump into a mathematically formal statement of stability, we briefly discuss the concept of a norm. A norm is a metric to measure the size of something. We have norms for vectors, time-varying signals, and matrices. A formal mathematical description of norms is given in [Skogestad and Postlethwaite, 1996]. A common example of a vector norm is the 2-norm given as

$$\|x(t)\|_2 = x^T(t)x(t) \tag{3.11}$$

which is also the inner product of vector and is sometimes written as $\|x(t)\|$ without the 2 subscript. Now we are ready to present our first set of stability definitions:

Definition 2 *The equilibrium point* 0 *of the continuous-time LTI system given by (3.9) is* ***stable*** *at time* t_0 *if for each* $\epsilon > 0$ *there exists a* $\delta(\epsilon) > 0$ *such that*

$$\|x(t_0)\| < \delta \Longrightarrow \|x(t)\| < \epsilon \quad \forall t \geq t_0 \tag{3.12}$$

Notice that this definition does not specify convergence upon the fixed point. It simply states that given any distance ϵ, we can find a starting point, $x(t_0) < \delta$, such that the system remains within ϵ of the fixed point for all time. An example of a system that is *stable* is a simple linear system with purely imaginary eigenvalues; the trajectory is an orbit of fixed distance from the origin. Notice also that $\|\ \ \|$ is any norm on $\Re^n$ as they are all topologically equivalent [Vidyasagar, 1978].

Often, this same definition is given in another formulation. An alternative but equivalent formulation is [Rugh, 1996]:

Definition 3 *The equilibrium point* 0 *of the continuous-time dynamic LTI system given by (3.9) is* ***stable*** *at time* t_0 *if there exists a finite constant* $\gamma > 0$ *such that*

$$\|x(t)\| < \gamma \|x(t_0)\| \quad \forall t \geq t_0 \tag{3.13}$$

This definition is sometimes referred to as the boundedness result because the norm of $x(t)$ is bounded by a multiple of the norm of the initial state x_0. Consider $\gamma = \frac{\delta}{\epsilon}$ to see compatibility with Definition 2.

While stability, as stated in the above two definitions, implies that the trajectory will not move arbitrarily far away from a fixed point, it does not imply that the trajectory will converge upon the fixed point. Here, we present a stronger stability definition possessing this convergence behavior:

Definition 4 *The equilibrium point* 0 *of the continuous-time dynamic LTI system given by (3.9) is* ***asymptotically stable*** *at time* t_0 *if it is stable at time* t_0 *and there exists a* $\delta > 0$ *such that*

$$\|x_0\| < \delta \Longrightarrow \lim_{t \to \infty} \|x(t)\| = 0 \tag{3.14}$$

This stability definition requires that for all initial states close enough to the fixed point, the system will converge to the equilibrium point eventually. Asymptotic stability is also referred to as *exponential stability* [1]. An equivalent restatement of this definition is:

Definition 5 *The equilibrium point* 0 *of the continuous-time dynamic LTI system given by (3.9) is* ***asymptotically stable*** *at time* t_0 *if it is stable at time* t_0 *and there exists a* $\gamma > 0$ *and* $\lambda > 0$ *such that*

$$\|x(t)\| \leq \gamma e^{-\lambda(t-t_0)} \|x_0\| \quad t \geq t_0 \tag{3.15}$$

Thus, the trajectory of $x(t)$ is not only bounded by a constant γ of the initial state but also by an exponentially decreasing function $e^{-\lambda(t-t_0)}$. This ensures that all trajectories will converge to the fixed point as $t \to \infty$.

These two definitions (and their subtle variations) form the core of stability theory for LTI systems. In the time-invariant case we consider here, the analysis of linear systems is fairly straightforward. The stability of these systems depends directly on the eigenvalues of the A matrix:

[1] To be precise, there are subtle differences between asymptotic stability and exponential stability; but they are equivalent for the autonomous case. See [Rugh, 1996] for more details.

Theorem 1 *The continuous-time LTI dynamic system given by (3.9) is stable iff* $Re(\lambda_i) \leq 0$ *where* λ_i *is an eigenvalue of* A*. That is, the real parts of the eigenvalues must be less than or equal to zero. Furthermore, the system is asymptotically stable iff* $Re(\lambda_i) < 0$*.*

Thus, ascertaining the stability of an LTI system amounts to checking the real parts of the eigenvalues of the state matrix A. For the class of linear, time-invariant dynamic systems, stability determination is rather straightforward.

For linear, time-varying systems (LTV), however, the eigenvalue condition is insufficient; instead, one must satisfy certain constraints on the solutions Φ of the differential equations. As stated previously, computing Φ exactly is often an intractable task; therefore, ascertaining the stability of LTV systems is problematic. Liapunov's direct method, discussed in the next section, provides an alternative way to arrive at stability that overcomes these difficulties.

3.3 Liapunov's Direct Method

In this section, we outline a different approach to determine the stability of a continuous-time dynamic system. It is important to note that we are not redefining our notion of stability; instead Liapunov's direct method is merely an alternative way of arriving at the same conclusions regarding system stability: two different paths to the same destination (we are now the blind man pulling on the elephant's trunk instead of its tail). Any system that is proven stable via Liapunov's direct method will also be stable according to the definitions presented in Section 3.2, but for many systems, it is easier to reach the stability conclusions using Liapunov's method rather than meeting the requirements specified in the definitions. Again, the definitions in this section are taken primarily from [Rugh, 1996; Vidyasagar, 1978; Desoer and Vidyasagar, 1975].

For this section, we return our attention to the generic formulation for a linear, possibly time-varying, continuous-time, dynamic system with no input/output signals:

$$\dot{x}(t) \quad = \quad A(t)x(t) \tag{3.16}$$

Liapunov's direct method (also known as Liapunov's second method) uses a function defined on the state space. If this function meets certain criteria, then it is referred to as a Liapunov function.

Definition 6 *Consider the continuous-time dynamical system given by (3.16) with an equilibrium point at* $\bar{x}$*. A function* V *defined on a region* Ω *(*$\Omega \subset \Re^n$*) of the state space which includes* $\bar{x}$ *is a* ***Liapunov function candidate*** *if*

1. V *is continuous and continuously differentiable.*
2. $V(x) > V(\bar{x}) \quad \forall x \in \Omega, x \neq \bar{x}$

Essentially, a Liapunov function candidate is a "bowl-shaped" function with a unique minimum at the fixed point.

Definition 7 *Consider the continuous-time dynamical system given by (3.16) with an equilibrium point at* $\bar{x}$*. A Liapunov function candidate,* V*, is a* ***Liapunov function*** *if*

$$\dot{V}(x) \leq 0 \quad \forall x \in \Omega \tag{3.17}$$

A Liapunov function meets the requirements of being a candidate and it also is monotonically non-increasing along trajectories of $x(t)$. Returning to our visual image of the bowl function, $x(t)$ indicates how the state x moves along the surface of the bowl according to the dynamic system given by(3.16). A Liapunov Function, then, has trajectories which never move up the surface of the bowl – they always move downward on the bowl surface. It is also convenient to think of V as an energy function. In fact, for many physical systems, V is conveniently chosen to be the total energy of the system. The definition of a Liapunov function facilitates several stability theorems.

Theorem 2 *If V is a Liapunov function for the system specified by (3.16), then the system is stable. Furthermore, if $\dot{V}(x) < 0$, then the system is asymptotically stable.*

The proof of this theorem is complex and thus the reader is directed to [Rugh, 1996; Vidyasagar, 1978; Aggarwal and Vidyasagar, 1977] for various versions of the proof. Often, there is some confusion in the literature as to whether the time-derivative of the Liapunov function (3.17) is non-positive (stability) or strictly negative (asymptotic stability); when reading other texts, the reader should be aware of which version of Liapunov function the author assumes.

One particular choice of Liapunov functions is the square of the 2-norm. If we can be assured that $\|x\|^2$ decreases with t, then we have satisfied the requirements for asymptotic stability:

$$V(x(t)) = \|x(t)\|^2 = x^T(t)x(t) \tag{3.18}$$

$$\dot{V}(x(t)) = \frac{d}{dt}\|x(t)\|^2 = \dot{x}^T(t)x(t) + x^T(t)\dot{x}(t) \tag{3.19}$$

$$= x^T(t)[A^T(t) + A(t)]x(t) \tag{3.20}$$

(3.20) is actually a restrictive version of a more general formulation. To arrive at the more general formulation, we recast (3.18) as a *quadratic form* which is also referred to as a *quadratic Liapunov form* by:

$$x^T(t)Q(t)x(t) \tag{3.21}$$

We require that the quadratic Liapunov function, $Q(t)$, be symmetric and positive-definite. It can be shown that these stipulations on $Q(t)$ are equivalent to the conditions to the Liapunov candidate function as stated in Definition 6. Taking the time-derivative of the quadratic form, we arrive at:

$$\frac{d}{dt}[x^T(t)Q(t)x(t)] = x^T(t)[A^T(t)Q(t) + Q(t)A(t) + \dot{Q}(t)]x(t) \tag{3.22}$$

Theorem 3 *The continuous-time dynamical system given by (3.16) is stable iff*

$$\eta I \leq Q(t) \leq \rho I \tag{3.23}$$

$$A^T(t)Q(t) + Q(t)A(t) + \dot{Q}(t) \leq 0 \tag{3.24}$$

where η, ρ are positive constants. Furthermore, the system is asymptotically stable iff (3.24) is replaced by

$$A^T(t)Q(t) + Q(t)A(t) + \dot{Q}(t) \leq \nu I \tag{3.25}$$

where ν is a positive constant.

These quadratic forms are yet another way to arrive at the same stability conclusions. As we shall see in Section 3.9, quadratic forms provide general and powerful analysis tools for stability.

In summary, Liapunov's direct method is an alternative way to conclude stability results for a dynamic system. The system may be of sufficient complexity that a straight forward application of the definitions given in Section 3.2 may be exceedingly difficult or impossible. This is mostly due to the fact that one must solve the differential equation (3.7) of the system in order to meet the definitional requirements. Instead, if one can find suitable Liapunov functions for these systems, then one may be able to prove the stability of such systems. However, there is no step-by-step method which generates Liapunov functions. Often, one must rely upon intuition and experience to formulate Liapunov function candidates and then find the candidates (if any) which meet the trajectory requirements to become full Liapunov functions. Despite this limitation, Liapunov functions have been found for a large number of dynamical systems. A thorough treatment of the subject can be found in [Vidyasagar, 1978].

3.4 Input-Output Stability

In the previous two sections, we primarily consider the notion of stability for the zero-input case; these sections present sufficient conditions for the stability of systems with external inputs set to 0. In this section, we return to the case where external inputs are non-zero. As yet another view of stability, we can approach the topic by looking at conditions on the input/output behavior of the system. For clarity, we restate the linear, time-invariant system with inputs and outputs here:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{3.26}$$
$$y(t) = Cx(t) \tag{3.27}$$

Definition 8 *Consider the continuous-time, LTI system given by (3.26, 3.27). The system is* ***bounded-input bounded-output stable (BIBO)*** *if there exists a finite constant η such that for any any input $u(k)$ we have*

$$\sup_{t \geq t_0} \|y(t)\| \leq \eta \sup_{t \geq t_0} \|u(t)\| \tag{3.28}$$

where *sup* stands for supremum: a variant of *max* with mathematically subtle differences [2]. This definition provides the familiar "bounded-input, bounded-output" (BIBO) stability notion: if the input signal is finite, then the output signal must also be finite. The constant, η, is often called the *gain* of the system. Using the 2-norm, we consider BIBO in the practical terms of energy. The ratio of the output signal energy to the input signal energy is a finite constant (i.e. the energy gain is finite); the system cannot inject an infinite amount of energy, and thus, is stable. A useful analogy is served by water in a sink. When the tap is running, energy is being added to the system (the water will continue to swirl around the basin). When the tap is turned off the water will eventually drain from the basin and the sink will settle to its equilibrium point of no energy (no water). If the amount of water added from the tap is finite, then the amount of water draining from the basin will also be finite; the sink cannot produce water.

Even though we have stated BIBO stability for an LTI system, there is nothing specific about LTI that affects the definition of stability. In fact, the same definition applies equally to other systems such as time-varying and nonlinear systems.

In the next theorem about BIBO stability we use two technical terms from the control field: controllability and observability. We briefly define them here; a more detailed definition can be found in [Skogestad and Postlethwaite, 1996; Rugh, 1996; Vidyasagar, 1978]. *Controllability* is the ability to direct the system from any arbitrary starting state, x_1, to any arbitrary ending state, x_2, in a finite amount of time by only manipulating the input signal $u(t)$. If it is not possible to direct the system in this manor, then the system is said to be uncontrollable. Similarly, if we can reconstruct the system state, $x(t)$, by watching only the system output, $y(t)$, then the system is said to be *observable*. An unobservable system may have hidden dynamics that affect internal states of the system but do not affect the system output. For our purposes, we require the system to be controllable and observable so that we can relate BIBO stability to "internal stability" as stated in the previous two sections.

Theorem 4 *Consider the (controllable and observable) LTI system given by (3.26,3.27). The system is asymptotically stable iff it is BIBO stable.*

We see that BIBO stability is mathematically equivalent to the asymptotic stability discussed early in the definitions and in Liapunov's direct method. Intuitively this makes sense. If we inject a finite amount of energy into the system ($\|u\| < \infty$), and the output energy of the system is finite ($\|y\| < \infty$), then necessarily the internal signals of the system must asymptotically decay to 0. The BIBO perspective defines stability in terms of the input/output behavior of the system. This is the first step towards examining the more complex stability arrangements of feedback systems.

[2] Maximum represents the largest value while supremum is defined as the least upper bound – the actual bound may not be reached but only approached asymptotically [Skogestad and Postlethwaite, 1996]

3.5 Feedback Stability

To this point, we have been considering continuous-time LTI systems in isolation (refer back to Figure 3.1). Most control systems of interest are composed of multiple LTI systems combined in an array of fashions such as parallel connections, serial connections and particularly feedback connections. Generally, LTI systems combine to form other LTI systems. Figure 3.2 depicts a typical feedback arrangement.

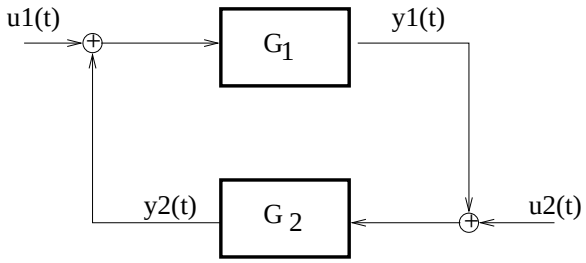


Figure 3.2: Feedback System

There are a multitude of theorems that guarantee the stability of this type of system for particular classes of systems for G_1 and G_2. We begin with the most general formulation in which there are no restrictions on G_1 and G_2, then we introduce additional stability proofs involving specific, but common, restrictions on these subsystems.

The main theorem of this section is the *Small Gain Theorem*. Suppose we determine that the systems, G_1, and G_2, are both individually BIBO stable with gains of $\eta_1 = 2$ and $\eta_2 = 5$, respectively. Consider a finite amount of energy injected into this system at input u_1. System G_1 will magnify the energy by a factor of 2. Next, system G_2 will magnify the energy by a factor of 5. As the energy circulates around the feedback loop, it is easy to see that it will be magnified by 10 for each pass around loop; the energy will grow without bound. Even though both systems, G_1 and G_2, are BIBO stable, the *feedback interconnection* of the components renders the overall system unstable. Thus, the connectedness of the components requires further restrictions on the component gains to make the overall system stable. We formalize this notion with the following statement of the *Small Gain Theorem* [Desoer and Vidyasagar, 1975].

Theorem 5 *Consider the system depicted in Figure 3.2. The equations of this system are given by:*

$$y_1 = G_1(u_1 + y_2) \tag{3.29}$$
$$y_2 = G_2(u_2 + y_1) \tag{3.30}$$

Suppose that G_1 and G_2 are causal and BIBO stable systems:

$$\|G_1 e_1\| \leq \eta_1 \|e_1\| + \beta_1 \tag{3.31}$$
$$\|G_2 e_2\| \leq \eta_2 \|e_2\| + \beta_2 \tag{3.32}$$

If $\eta_1 \eta_2 < 1$ then,

$$\|y_1\| \leq \eta_1 \|e_1\| \leq \frac{\eta_1(\|u_1\| + \eta_2\|u_2\| + \beta_2 + \eta_2\beta_1)}{(1 - \eta_1\eta_2)} \tag{3.33}$$
$$\|y_2\| \leq \eta_2 \|e_2\| \leq \frac{\eta_2(\|u_2\| + \eta_1\|u_1\| + \beta_1 + \eta_1\beta_2)}{(1 - \eta_1\eta_2)} \tag{3.34}$$

Although (3.33) and (3.34) are somewhat complex, this theorem essentially states that if the input energy in u_1 and u_2 is finite and the loop gain is less than unity, then the output energy of y_1 and y_2 is also finite. The small gain theorem is a very broad statement about feedback system stability. It has been stated and proved in many different formulations. Several references provide this and other formulations in more detail [Desoer and Vidyasagar, 1975; Vidyasagar, 1978; Megretski and Rantzer, 1997a; Megretski and Rantzer, 1997b].

3.6 Nominal Stability

In the previous four sections, we have addressed stability from a mathematician's perspective. While this perspective is both valid and useful, a control engineer casts a different interpretation on stability. The engineer's perspective addresses specific systems encountered frequently in control analysis and design. In this section, we remove our mathematicians hat and replace it with that of the engineer. We examine stability from the utility of control analysis.

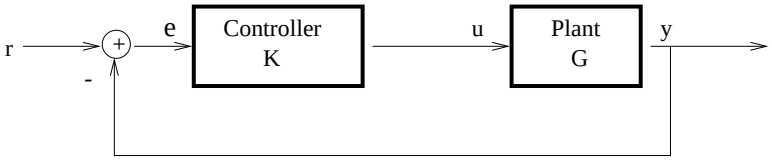


Figure 3.3: Typical System

Consider the dynamic system typically encountered by a control engineer as shown in Figure 3.3. Notice that this diagram could easily be recast into the feedback arrangement of Figure 3.2; we simply combine the series blocks of the controller, K, and the plant, G, to arrive at G_1. The feedback system, G_2, is a unity gain. We could then apply the small gain theorem to prove stability for a particular controller and plant. However, the small gain theorem is a rather blunt tool to apply in this case; more precise stability tools have been developed for this specific arrangement, because it is encountered so frequently.

First, the system in Figure 3.3 is typically expressed in three different representations: the state space equation, a time function, and the transfer function. The *state space* representation is a set of differential equations describing the behavior of a system; this is the representation that we have been dealing with so far in this chapter (see Equation 3.5 for an example). In simpler systems, we can solve the set of differential equations to obtain an exact solution to the system. This solution is the *time function* representation. Both the state space and time function representations are expressions in the time domain. In contrast, the *transfer function* representation expresses the system in the frequency domain. One must use the Laplace transform to move back and forth between the two domains.

To illustrate the three representations, we show an example system in all three formulations. Consider the system in Figure 3.4. In the top portion of this figure is the control system of Figure 3.3 with a specific controller and plant. We draw a dotted box around this control system and compute the transfer function as seen from input r to output y. This is depicted in the lower portion of Figure 3.4.

Most control engineering analysis is performed in the frequency domain; hence, we start here. The controller, $K(s)$, is unity and the plant, $G(s)$, is an integrator expressed as:

$$K(s) = 1 \tag{3.35}$$

$$G(s) = \frac{1}{s} \tag{3.36}$$

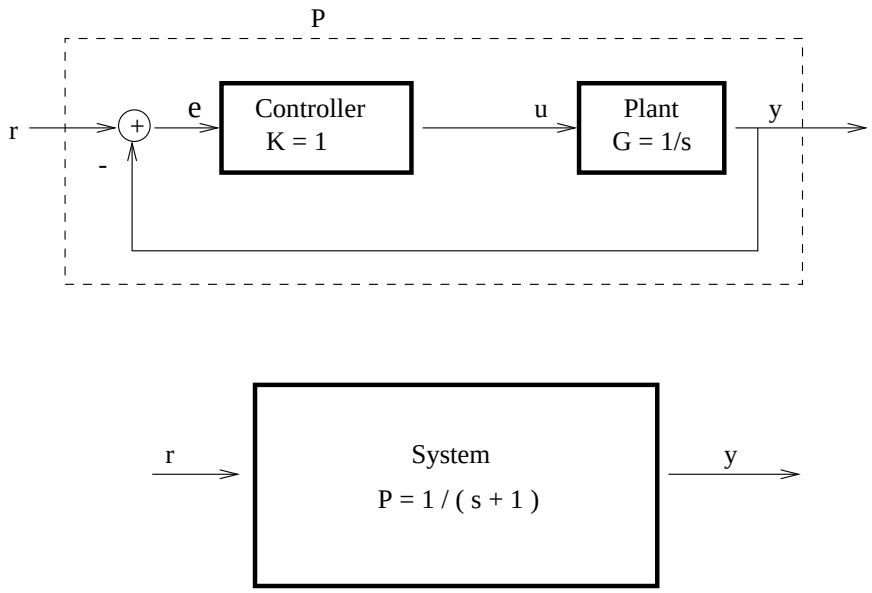


Figure 3.4: Example System

where s is the Laplace variable (frequency). We combine these systems into the *open-loop function*, $L(s)$ which is the system with the feedback path removed. In the time domain, $\ell(t)$, would be computed with a messy convolution integral; in the frequency domain, however, $L(s)$ is computed by multiplication.

$$L(s) = K(s)G(s) = \frac{1}{s} \tag{3.37}$$

Mason's Gain formula allows us to compute the overall transfer function of the system (with the feedback path added) [Phillips and Harbor, 1996]:

$$P(s) = \frac{L(s)}{1 + L(s)} = \frac{1}{s+1} \tag{3.38}$$

The transfer function is a convenient representation because it facilitates easy computation of the system output, $Y(s)$, via multiplication with the system input, $R(s)$:

$$Y(s) = P(s)R(s) \tag{3.39}$$

The attractiveness of the transfer function's frequency domain approach arises from the easy multiplication used to manipulate systems and compute outputs.

We use the inverse Laplace transform to obtain the time-domain response for $P(s)$.

$$P(s) = \frac{1}{s+1} \quad \overset{\text{InverseLaplace}}{\rightarrow} \quad y(t) = e^{-t}y(t_0) + \int_0^t e^{-\tau} r(t-\tau) d\tau \tag{3.40}$$

where the right hand side is recognized as the standard solution to the differential equation; this solution is composed of the zero-input response and the convolution integral for the zero-state response.

The equivalent state space representation is given as:

$$\dot{y}(t) = Ay(t) + Br(t) = -y(t) + r(t) \tag{3.41}$$

Stability of these systems can be determined in a number of ways. Because the system is LTI, the eigenvalues of the state matrix indicate stability (see Theorem 1). Here we have only one eigenvalue ($\lambda = -1$) which has a negative real portion; thus the system is stable [Rugh, 1996].

However, we often do not possess the state space representation and it may be difficult to compute this from the transfer function. Instead, one can examine the *poles* of the transfer function which are the zeros in the denominator of the transfer function. If the zeros all lie in the left-half complex plane (real part is negative), then the system will be stable. Here we see that the zero of $s + 1$ is $s = -1$ and hence the system is stable [Phillips and Harbor, 1996].

A typical control problem involves devising a controller that maximizes the performance (minimum tracking error) while providing a stable feedback system. The two main approaches are numerical and graphical. Numerical techniques such as H_2 and H_∞ optimal control rely upon computers to find a controller to optimize a particular matrix norm of the system [Skogestad and Postlethwaite, 1996]. The art of the numeric methods concerns devising the appropriate matrix norm to capture both the stability and performance criteria. The graphical techniques, often called *loop-shaping*, depict the system visually [Phillips and Harbor, 1996; Skogestad and Postlethwaite, 1996]. The stability and performance properties of the system can be seen by a well-trained control engineer who adjusts the controller to better match a desired graphical feature. Unlike the numerical methods, loop-shaping is restricted to SISO (Single-Input, Single-Output) systems because higher order MIMO (Multi-Input, Multi-Output) systems cannot be portrayed graphically [Phillips and Harbor, 1996].

These two techniques form the basis for much of the voluminous history of control theory. However, the tools of robust control have recently emerged to address weaknesses in these control design techniques. In the next section, we outline an important limitation with traditional control and examine how robust control overcomes this difficulty.

3.7 Robust Stability

The control engineer designs controllers for physical systems. These systems often possess dynamics that are difficult to measure accurately such as friction, viscous drag, unknown torques and other dynamics. Furthermore, the dynamics of the system often change over time; the change can be gradual such as when devices wear or new systems break in, or the change can be abrupt as in catastrophic failure of a subcomponent or the replacement of an old part with a new one. As a consequence, the control engineer never completely knows the precise dynamics of the system.

Modern control techniques rely upon mathematical models. It is necessary to characterize the system mathematically before a controller can be designed. A *mathematical model* of the physical system is constructed, the controller is designed for the model, and then the controller is implemented on the physical system. If there are substantial differences between the model and the physical system, then the controller may operate with compromised performance and possibly be unstable. The situation is exacerbated because we typically require that the model be LTI; thus, the known, nonlinear dynamics of the system cannot be included in the model. This problem of differences between the model and the physical system is called the robustness problem. A robust controller is one that operates well on the physical system despite the differences between the design model and the physical system.

Traditional design methods (loop-shaping and numeric methods) are inadequate to deal with the robustness problem. Because it is graphical, loop-shaping can actually overcome model differences. Appropriate fudge-factors can be incorporated into controller design making loop-shaping robust to a certain extent. However, the graphical techniques are confined to SISO (single-input, single-output) systems; this is a major limitation as most control designs are MIMO (multiple-input, multiple-output). Numeric optimization methods are inadequate because they design tightly around the model; numeric techniques exploit the dynamics of the model in order to achieve the best control performance. Because these exploited dynamics may not be present in the physical system, the performance and stability of the controller can be arbitrarily poor when the controller is implemented on the physical system.

Robust control overcomes this obstacle by incorporating *uncertainty* into the mathematical model. Numerical optimization techniques are then applied to the model, but they are confined so as to not violate the uncertainty regions. When compared to the performance of numeric op-

timization techniques, robust designs typically do not perform as well on the model (because the uncertainty keeps them from exploiting all the model dynamics). However, while the numeric optimization techniques often perform very poorly on the physical plant, the performance of the robust controller on the physical plant is similar to its performance on the model.

The key to robust designs is characterizing the uncertainty and adding it to the model in the appropriate ways. Consider Figure 3.5 in which uncertainty has been added to the system. The uncertainty block is a transfer function (or equivalently a state space representation) that captures the unmodeled dynamics of the system. Note, even though we have depicted only one uncertainty block, models often contain multiple uncertainty blocks to capture unmodeled dynamics in different parts of the system.

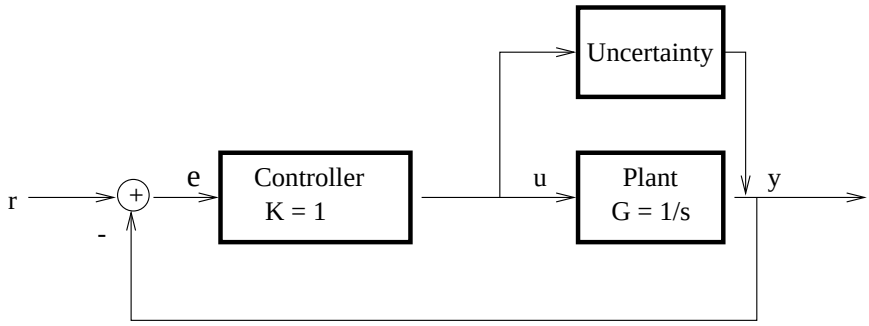


Figure 3.5: Control System with Uncertainty

A full example of a robust control design is given in Chapter 6 with the distillation process example to motivate the need for robust control designs and illustrate the advantages of robust control over numeric optimization techniques. More detail on robust control is found in [Skogestad and Postlethwaite, 1996; Zhou and Doyle, 1998; Doyle et al., 1992].

After the appropriate uncertainty blocks have been added, the system is condensed so that all the uncertainty is grouped together in one block, and all the LTI components are grouped together in a second block. Simple operations on combining systems enable us to form this two sub-system formulation. Such an arrangement is depicted in Figure 3.6 where M represents the known LTI subsystem, and Δ represents the uncertain portion of the system. This arrangement belongs to a general class of system configurations known as linear fractional transforms (LFTs). LFTs are a framework structured to facilitate system analysis.

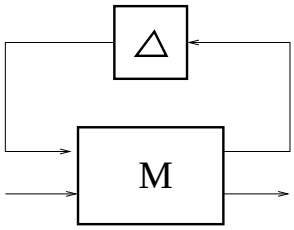


Figure 3.6: M-Δ System Arrangement (as LFT)

Once the system has been posed as an LFT with LTI in one block and uncertainty in the other block, then we can apply a number of tools to determine the stability of such a system. Two of these tools are μ-analysis and IQC-analysis. We present the basics of μ-analysis in the next section and develop the IQC method in Section 3.9.

3.8 μ-analysis

μ-analysis is a tool used to tackle robust control problems in which the uncertainty is complex and structured [Balas et al., 1996; Packard and Doyle, 1993; Young and Dahleh, 1995]. The primary parts of μ-analysis are presented here. More detailed descriptions are available in [Balas et al., 1996; Packard and Doyle, 1993; Young and Dahleh, 1995; Skogestad and Postlethwaite, 1996; Zhou and Doyle, 1998; Doyle et al., 1992]. A traditional introduction to μ-analysis consists of two parts. In the first part, we define a matrix function, μ, and discuss ways to compute it; this first part is an exercise in linear algebra. In the second part of the standard presentation of μ-analysis, we show how the linear algebra result can be used to ascertain the stability of dynamic systems. We first begin by defining μ in the context of a matrix function.

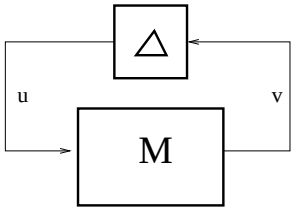


Figure 3.7: M-Δ System Arrangement

Consider the two blocks interconnected in Figure 3.7 where M and Δ are complex matrices. Again, we will later relate these blocks to dynamic systems but for now we consider them soley as interconnected matrices for the purpose of exploring their mathematical properties. We place the following restrictions on M and Δ. Let M be an nxn complex block: $M \in C^{n\text{x}n}$. We define Δ to be a diagonal complex block parameterized by a set of integers $(d_1, d_2, \ldots, d_k, f_1, f_2, ..., f_\ell)$. Δ consists of two parts. The first part of Δ is a series of k identity matrices multiplied by a constant. The dimension of each identity matrix is d_ixd_i and the constant for each is γ_{d_i}. These identity matrices are placed along the upper left diagonal of Δ. The second part of Δ is a series of ℓ complex blocks each of dimension f_jxf_j. These full complex blocks, denoted by δ_{f_j}, are placed along the lower diagonal of Δ. Formally, Δ is the set of all diagonal matrices such that

$$\Delta = \{diag[\gamma_1 I_{d_1}, ..., \gamma_k, I_{d_k}, \delta_{f_1}, ..., \delta_{f_\ell}]\}, \tag{3.42}$$

$$\gamma_{d_i} \in C, \quad \delta_{f_i} \in C^{f_i \text{x} f_i}, \tag{3.43}$$

$$\sum_{i=1}^{k} d_k \quad + \quad \sum_{i=1}^{\ell} f_i \quad = \quad n. \tag{3.44}$$

As an example, consider one particular element of Δ with $n = 6$ parameterized by $(d_1 = 3, d_2 = 1, f_1 = 2)$. This element of Δ has two identity matrices (we will use $\gamma_1 = 5 + i$ for the first constant and $\gamma_2 = 2$ for the second constant); the dimension of the first identity matrix is d_1xd_1 = 3x3 and the second has dimension d_2xd_2 = 1x1. These two identity matrices are place along the upper left diagonal of Δ. Then we have one complex block of dimension f_1xf_1 = 2x2.

$$\begin{bmatrix} 5+i & 0 & 0 & 0 & 0 & 0 \\ 0 & 5+i & 0 & 0 & 0 & 0 \\ 0 & 0 & 5+i & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.1-0.1i & 5i \\ 0 & 0 & 0 & 0 & 46.8 & -7+3i \end{bmatrix} \tag{3.45}$$

It is important to note that the above example is a particular element of Δ; recall Δ is actually defined as the set of all matrices meeting the conditions of the parameter set $(d_1, d_2, \ldots, d_k, f_1, f_2, ..., f_\ell)$.

Also note that Δ is "block diagonal" meaning that there are sub-blocks within Δ that occupy the diagonal of Δ.

Define $\bar{\sigma}(\Delta)$ to be the largest singular value of Δ; this is the largest amplification possible by the matrix. Finally, we arrive at the fundamental definition and theorem for μ-analysis.

Definition 9 *The **complex structured singular value**, $\mu_\Delta(M)$, is the size of the smallest Δ which makes $I - M\Delta$ singular. It is defined as:*

$$\mu_\Delta(M) = \frac{1}{\min_\Delta\{\bar{\sigma}(\Delta) |\ \det(I - M\Delta) = 0\}} \tag{3.46}$$

$$\mu_\Delta(M) = 0, \text{ if no } \Delta \text{ makes } I - M\Delta \text{ singular.} \tag{3.47}$$

Before we conclude the linear algebra part and begin the system stability analysis part, we briefly discuss the computation of μ. For the vast majority of "interesting" M matrices an exact computation of μ is not possible. Instead we compute lower and upper bounds. A well known property of μ is given by the following theorem [Packard and Doyle, 1993; Skogestad and Postlethwaite, 1996]:

Theorem 6 *For the feedback system in Figure 3.7,*

$$\rho(M) \leq \mu_\Delta(M) \leq \bar{\sigma}(M) \tag{3.48}$$

where $\rho(M)$ is the spectral radius (the maximum eigenvalue), and $\sigma(\bar{M})$ is the largest singular value. Despite the difficulties in computing μ, the computation of both $\rho(M)$ and $\bar{\sigma}(M)$ is comparatively simple. However, these two bounds are not generally very tight and thus are not particularly useful in this form. The trick is to exploit the diagonal structure of Δ in order to tighten these bounds. We employ a scaling matrix D to replace M with DMD^{-1}. For different classes of matrices D, this multiplication changes both $\rho(M)$ and $\bar{\sigma}(M)$, but does not affect $\mu_\Delta(M)$. See [Packard and Doyle, 1993] for the detailed restrictions on D. From the class of permissable D matrices, we choose a specific matrix to minimize the upper bound of $\bar{\sigma}$ and from a different class we select a different scaling matrix to maximize the lower bound of ρ. Note, the matrices for maximizing the lower bound and minimizing the upper bound are different and there are different restrictions on the class of each matrix. Fortunately, finding the optimal D matrices for either bound is a linear matrix inequality (LMI) problem that is easily solved in polynomial time [Gahihet et al., 1995]. With these scaled bounds, we are typically able to approximate μ quite tightly.

At this point, we have presented a mathematical definition of μ in terms of a matrix function and we have shown how to approximate μ using upper and lower bounds. Now we turn to the second part of μ involving the application of the matrix function to the stability analysis of systems. We apply a variant of the small gain theorem to prove that this system is stable given the restrictions on M and Δ. Then, we show that this arrangement can be formed for any LTI system with a specific type of uncertainty.

The definition of μ (Defintion 9) yields an immediate stability result. A variant of the small gain theorem employs μ to characterize the closed-loop gain of the system. The importance of this application of μ is clearly and concisely explained by Doyle [Packard and Doyle, 1993]; thus, I quote his explanation here:

> It is instructive to consider a "feedback" interpretation of $\mu_\Delta(M)$ at this point. Let $M \in C^{nxn}$ be given, and consider the loop shown in Figure 3.7. This picture is meant to represent the loop equations $u = Mv$, $v = \Delta u$. As long as $I - M\Delta$ is non-singular, the only solutions u, v to the loop equations are $u = v = 0$. However, if $I - M\Delta$ is singular, then there are infinitely many solutions to the equations, and norms $\|u\|$, $\|v\|$ of the solutions can be arbitrarily large. Motivated by the connections with stability

of systems, we will call this constant matrix feedback system "unstable". Likewise, the term "stable" will describe the situation when the only solutions are identically zero. In this context then, $\mu_\Delta(M)$ provides a measure of the smallest structured Δ that causes "instability" of the constant matrix feedback loop. The norm of this "destabilizing" Δ is exactly $1/\mu_\Delta(M)$.

There are actually many different "flavors" of stability theorems involving μ; each variation measures a slightly different system property and arrives at a different stability result. The result of each theorem is a μ stability test that involves computing μ for a specific part of the perturbed system. We summarize one of the more prominent μ-stability theorems [Packard and Doyle, 1993]:

Theorem 7 ***μ-analysis stability theorem***
Consider the linear system given by:

$$\begin{bmatrix} x_{k+1} \\ err_{k+1} \\ u_{k+1} \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} x_k \\ d_k \\ v_k \end{bmatrix} \tag{3.49}$$

$$v_k = \Delta u_k \tag{3.50}$$

$$\Delta = \begin{bmatrix} \Delta_1 & \\ & \Delta_2 \end{bmatrix} \tag{3.51}$$

which is depicted in Figure 3.8.

The following are equivalent:

1. *The dynamic system is stable,*
2. $\mu_{\Delta_S}(M) < 1$,
3. $\max_{\theta\in[0,2\pi]} \mu_{\Delta_P}(\mathcal{L}(e^{j\theta} I_n, M)) < 1$.

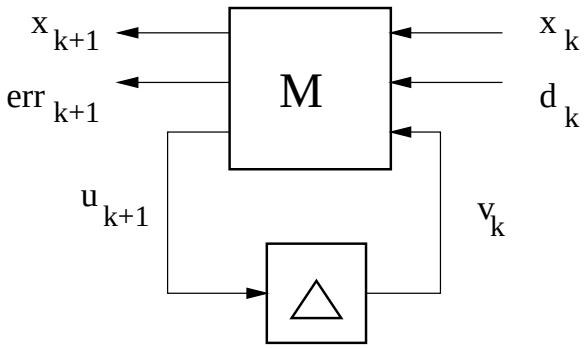


Figure 3.8: μ-analysis System Arrangement

The complete analysis and proof for the μ-analysis stability theorem are available in [Packard and Doyle, 1993]. The direct result of Theorem 7 are two μ-analysis stability tests. Because they are so central, we restate them here:

$$\mu_{\Delta_S}(M) < 1, \tag{3.52}$$

$$\max_{\theta\in[0,2\pi]} \mu_{\Delta_P}(\mathcal{L}(e^{j\theta} I_n, M)) < 1. \tag{3.53}$$

Equation 3.52, commonly known as the *state space μ test*, measures the structured singular value for a specific subcomponent of M [Packard and Doyle, 1993]. If μ is less than unity, then the system

represented by M is robustly stable. Equation 3.53, referred to as the *frequency domain μ test*, measures a slightly different aspect of the system. Again, if the $\mu < 1$ condition is satisfied, then robust stability is achieved. The $\mathcal{L}$ notation refers to an LFT formed from the nominal control system and the uncertainty blocks. More details can be found in [Packard and Doyle, 1993].

As discussed previously, μ cannot be computed directly. Instead, upper and lower bounds for μ are used as an approximation. We are particularly concerned with the upper bounds as these provide limitation on the size of the perturbation. The following three upper bounds are used for actual computations:

$$\inf_{D_s} \bar{\sigma}(D_s^{1/2} M D_s^{-1/2}) < 1, \tag{3.54}$$

$$\inf_{D_p} \max_{\theta \in [0,2\pi]} \bar{\sigma}[D_p^{1/2}(\mathcal{L}(e^{j\theta} I_n, M) D_p^{-1/2}] < 1, \tag{3.55}$$

$$\max_{\theta \in [0,2\pi]} \inf_{D_p} \bar{\sigma}[D_p^{1/2}(\theta) \mathcal{L}(e^{j\theta} I_n, M) D_p^{-1/2}(\theta)] < 1, \tag{3.56}$$

All of the above computations use the maximum singular value, $\bar{\sigma}$, as the upper bound approximation to μ. Recall, that the singular value can be modified via scaling matrices, D, in order to tighten the upper bound. This allows us to approximate μ quite tightly. Equation 3.54 computes the upper bound for the state space μ test of Equation 3.52. For the frequency domain μ test of Equation 3.53, there are actually two different upper bound approximations. Equation 3.55 employs constant D scaling matrices while Equation 3.56 bases its calculations on scaling matrices that vary as a function of frequency.

The distinction between Equation 3.55 and Equation 3.56 is critical because each test has restrictions on the type of structured uncertainty included in Δ. Specifically, the frequency varying upper bound test of Equation 3.56 requires that the uncertainty captured by Δ be LTI while the stronger μ tests of Equation 3.54 and Equation 3.55 allow uncertainty that is both time-varying and "bounded" nonlinear. The Matlab μ-analysis toolbox implements the frequency varying μ test, and thus, is applicable only to systems with LTI uncertainty. At the moment, this is the only available μ test implemented in software. The stronger versions of μ tests are not implemented in available software.

For the work in this dissertation, we use the μ tests to ascertain the stability of systems containing a neuro-controller. The neural network controller has both time-varying and nonlinear dynamics. As a result, the stability *guarantees* of the Matlab μ test do not apply to our neuro-control system. Playing the role of Devil's Advocate, we can construct systems with non-LTI dynamics that pass the frequency-varying test of Matlab's μ-analysis toolbox, but are actually quite unstable.

While the exact mathematical guarantees of μ-analysis stability do not hold for our neuro-control systems, the μ-analysis toolbox does still play an important role for our work. For the neuro-control systems considered in this dissertation, the μ-analysis test works well as an indicator of system stability. Despite the fact that our non-LTI uncertainty violates the strict mathematical assumptions in the μ stability test, in practice, the μ stability test serves as a very good indicator of our system stability. Thus, we use Matlab's μ-analysis toolbox as an approximation of system stability. In the next section, we introduce a different stability tool, IQC. The stability test with IQC *does* apply to our non-LTI uncertainty, and thus, we do have a formal mathematical guarantee of stability. We also include the μ-analysis stability test because this is the primary commercial stability software currently on the market, it is well-known among researchers in the field, it does work accurately in practice for our systems, and it is likely that a stronger, non-frequency varying, commercial, product will be added to the Matlab toolbox in the future.

3.9 IQC Stability

Integral quadratic constraints (IQC) are a tool for verifying the stability of systems with uncertainty. In the previous section, we applied the *tool* of μ-analysis to determine the stability of such a system. IQC is a different tool which arrives at the same stability results. In this section, we present the basic

idea behind IQCs. The interested reader is directed to [Megretski and Rantzer, 1997a; Megretski and Rantzer, 1997b; Megretski et al., 1999] for a thorough treatment of IQCs.

Consider, once again, the feedback interpretation of a system as shown in Figure 3.9. The upper block, M, is the known LTI components; the lower block, Δ is the uncertainty.

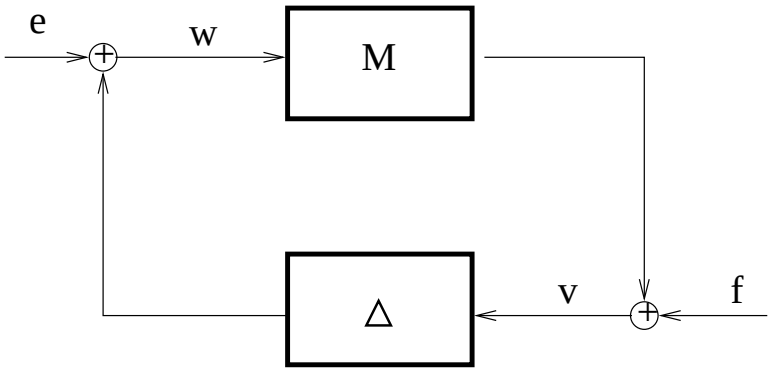


Figure 3.9: Feedback System

Definition 10 *An **integral quadratic constraint** (IQC) is an inequality describing the relationship between two signals, w and v. The relationship is characterized by the matrix Π as:*

$$\int_{-\infty}^{\infty} \left| \begin{array}{c} \hat{v}(j\omega) \\ \hat{w}(j\omega) \end{array} \right|^* \Pi(j\omega) \left| \begin{array}{c} \hat{v}(j\omega) \\ \hat{w}(j\omega) \end{array} \right| d\omega \geq 0 \tag{3.57}$$

where $\hat{v}$ and $\hat{w}$ are the Fourier Transforms of $v(t)$ and $w(t)$. We now summarize the main stability theorem of IQC [Megretski and Rantzer, 1997a].

Theorem 8 *Consider the interconnection system represented in Figure 3.9 and given by the equations*

$$v = Mw + f \tag{3.58}$$
$$w = \Delta(v) + e \tag{3.59}$$

Assume that:

- *The interconnection of M and Δ is well-posed. (i.e., the map from $(v, w) \rightarrow (e, f)$ has a causal inverse)*
- *The IQC defined by Π is satisfied.*
- *There exists an $\epsilon > 0$ such that*

$$\left| \begin{array}{c} M(j\omega) \\ I \end{array} \right|^* \Pi(j\omega) \left| \begin{array}{c} M(j\omega) \\ I \end{array} \right| \leq -\epsilon I \tag{3.60}$$

Then the feedback interconnection of M and Δ is stable.

A few comments are in order. First, the utility of this method relies upon finding the correct IQC, Π, which captures the uncertainty of the system. In general, finding IQCs is a difficult task that is beyond the grasp of most engineers who wish to apply this method. Fortunately, a library of IQCs for common uncertainties is available [Megretski et al., 1999]; more complex IQCs can be built by combining the basic IQCs.

Second, the computation involved to meet the requirements of the theorem is not difficult. The theorem requirements also transform into an LMI (linear matrix inequality). The LMI is a convex optimization problem for which there exist fast, commercially available, polynomial time algorithms [Gahihet et al., 1995].

Third, the IQC does not redefine stability. Once again the IQC is a tool to arrive at the same stability conclusions as other techniques. The inventors of IQC readily demonstrate that IQC is a generalization of both Liapunov's direct method and the small gain theorem; a slightly more complex analysis shows the connection to μ-analysis. The interested reader can find these details in [Megretski and Rantzer, 1997a; Megretski and Rantzer, 1997b].

Chapter 4

Static and Dynamic Stability Analysis

In the previous chapter, we constructed an increasingly sophisticated notion of stability by progressing from simple definitional stability, through Liapunov stability, to input/output stability and then finally system stability with μ-analysis and IQC-analysis. This chapter builds upon this previous framework to develop our main theoretical results. First we present a method to determine the stability status of a control system with a static neural network, a network with all weights held constant. We prove the that this method identifies all unstable neuro-controllers. Secondly, this chapter presents an analytic technique for ensuring the stability of the neuro-controller while the weights are changing during the training process. We refer to this as dynamic stability. Again, we prove that this technique guarantees the system's stability while the neural network is training.

It is critical to note that dynamic stability is not achieved by applying the static stability test to the system after each network weight change. Dynamic stability is fundamentally different than "point-wise" static stability. For example, suppose that we have a network with weight vector W_1. We apply the static stability techniques of this chapter to prove that the neuro-controller implemented by W_1 provides a stable system. We then train the network on one sample and arrive at weight vector W_2. Again we can demonstrate that the static system given by W_2 is stable. However, this does not prove that the system, while in transition from W_1 to W_2, is stable. We require the additional techniques of dynamic stability analysis in order to formulate a reinforcement learning algorithm that guarantees stability throughout the learning process. We include the static stability case for two important reasons. The static stability analysis is necessary for the development of the dynamic stability theorem. Additionally, the existing research literature on neural network stability addresses *only* the static stability case; here, we make the distinction explicit in order to illustrate the additional power of a neuro-controller that solves the dynamic stability problem.

This chapter actually presents two different variations of static and dynamic stability, one variation using μ-analysis and the other variation using IQC-analysis. While these two tools arrive at roughly equivalent decisions regarding system stability, the specific implementation details are different. We include both stability tools, μ-analysis and IQC-analysis, for several reasons. The μ-analysis stability test is beneficial because it is well-known among researchers in robust control and there is a commercial implementation available through Matlab. As discussed in Section 3.8, there are actually many different stability tests based upon the structured singular value, μ. Some of these tests allow non-LTI uncertainty functions while other variants of the μ test are restricted to LTI uncertainty functions; because the μ-analysis stability test implemented in Matlab requires LTI uncertainty while our neuro-controller possesses non-LTI uncertainty, we *invalidate* the mathematical guarantees of stability when we use the commercial Matlab product with our neuro-control system. However, for the type of systems we encounter, the Matlab μ-analysis stability test does work well in practice. Thus, it serves as a good approximate indicator of system stability. It is likely that other μ stability tests not requiring LTI uncertainty may become commercially available in the near future. Therefore, we include a μ-analysis version of the stability "proofs" here and we use

μ-analysis as a stability tool in our case studies. We include IQC-analysis because it is applicable to non-LTI uncertainty. With IQC-analysis, we do achieve the strict mathematical guarantees of system stability. The drawback of IQC-analysis is that the software is not commercially available; there is an unsupported beta version available from MIT [Megretski and Rantzer, 1997a; Megretski and Rantzer, 1997b; Megretski et al., 1999]. Our experiences with experiments in the case studies indicate that IQC-analysis is computationally more time-consuming than μ-analysis.

In summary, the μ-analysis version of the Static and Dynamic Stability Theorems presented in this chapter are valid; however, our application of the theorems is flawed because we use the Matlab μ-analysis toolbox which does not permit non-LTI uncertainty. For the IQC-analysis, both the theorems and our application of the theorems are valid for the non-LTI uncertainty.

The remainder of this chapter is organized as follows. In Section 4.1, we discuss the general approach and framework for proving the stability of systems with neural network controllers. This chapter then divides into two trends; one is based on using μ-analysis as the underlying tool for proving stability while the other trend utilizes IQC-analysis. Section 4.2 discusses how to convert the neural network into a format compatible with μ-analysis. Then Section 4.3 and Section 4.4 introduce and prove the Static Stability Theorem and the Dynamic Stability Theorem based upon the μ-analysis tool. The other trend concerning IQC-analysis has three corresponding sections. Section 4.5 shows how to convert the neural network into the IQC format. Section 4.6 and Section 4.7 present and prove the Static and Dynamic Stability Theorems based upon IQC-analysis. Finally, the two trends converge as Section 4.8 sketches the stable learning algorithm based upon the Dynamic Stability Theorem.

4.1 An Overview of the Neural Stability Analysis

Static and dynamic stability proofs are developed by constructing and analyzing a mathematical model of the plant. As discussed in Sections 3.6 and 3.7, the design engineer typically constructs an LTI model of the plant before building a controller. The plant model is an LTI (linear, time-invariant) system because a vast body of control research and software tools are available to aid in the construction of stable and high-performing controllers for LTI systems. Conversely, the design of controllers for non-LTI systems is greatly complicated by the prohibitively complex dynamics and, therefore, the lack of analytical tools. If the engineer desires stability assurances, the choice of a non-LTI controller is non-existent for all but the most trivial systems.

Adding the neural network to the system introduces nonlinearity. The nonlinearity is a desirable feature because the neural network derives much of its computational power from the nonlinearity. However, the introduction of nonlinearities prohibits us from applying the theory and techniques based on LTI systems. Even if we restrict ourselves to linear networks (a major sacrifice in the computational ability of a neural network), we still encounter difficulties as the network weights change during learning; this system is time-varying, and still violates the LTI assumption.

Thus, we arrive at the dilemma that has impeded progress in neuro-control to date. We desire the nonlinear component of neural networks for their ability to implement better controllers, but we require the system be LTI so that we can employ our analysis tools concerning stability. We also need to allow the network weights to change during learning – a further violation of the LTI principle. In this chapter, we first present a technique to overcome the *static stability problem*; we permit the nonlinearity of a neural network while still guaranteeing stability. We then present a second technique to solve the *dynamic stability problem* so that neural networks with time-varying weights are guaranteed to be stable.

In this dissertation, we use the techniques of robust control to propose a solution to both of these problems. As detailed in Section 3.7, robust control enables the design of controllers for systems that contain unknown nonlinearities and time-varying dynamics. This is achieved by identifying and characterizing the non-LTI components and then covering them with an uncertainty function. The robust control design techniques build controllers that remain stable for not only the LTI system, but also any unknown non-LTI components of the system existing within the uncertainty region.

We create two "versions" of the neuro-control system. The first version contains the neural

network with all of its nonlinear and time-varying features. We refer to this as the *applied version* of the neuro-controller because it is this version that is actually implemented as our control scheme. The second version is a purely LTI model with the non-LTI parts of the neural network converted to uncertainty blocks. This version, the *testable version*, is used only for the robustness tests to determine the stability status of the applied version. If we choose the uncertainty functions correctly, then stability guarantees of the testable version imply stability guarantees for the applied version of the neuro-control system. The next section addresses the problem of how to cover the neural network's nonlinearity with an appropriate uncertainty function; that is, we discuss how to derive the testable version from the applied version of the neuro-controller. The analysis of the next section is specific to the μ-analysis tools. Section 4.5 provides the same conversion for the IQC-analysis stability tool.

4.2 Uncertainty for Neural Networks: μ-analysis

The neural network possesses two violations of the LTI principle. The first violation is the nonlinear activation function of the network's hidden layer. The second violation is the time-varying weight changes that the network incurs during training. If we are to include a neural network as part of the control system, then we must "convert" the non-LTI parts of the neuro-controller. In this section, we address the first violation. We temporarily ignore the second violation of time-varying dynamics by stipulating that the networks' weights remain fixed. Specifically, this section discusses the modifications we make on the applied version of the neuro-controller in order to convert it to the testable version of the neuro-controller in which the nonlinear hidden units are covered with uncertainty functions. Again, the analysis of this section assumes that μ-analysis is the intended stability tool. Refer to Section 4.5 to see a parallel conversion for the IQC-analysis stability tool.

At this point, we interject a few comments about architectural decisions regarding the neural network. There are a multitude of neural networks from which we can select; here we have chosen a commonly used architecture with two feed forward layers, a nonlinear hyperbolic tangent function in the activation layer, and a linear (no activation function) output layer. We have made further stipulations regarding the number of hidden units (h) and the number of layers (2). We need to be precise in sorting out our motivations for selecting a particular architecture. Specifically, we need to indicate which architecture choices are a result of the requirements of the stability theory, the desire to choose common and simple architectures, and the need to select an architecture that works well in practice for a particular problem. For the moment, we simply present the neural network architecture. We discuss why and how we arrived at this architecture in Chapter 5.

Let us begin with the conversion of the nonlinear dynamics of the network's hidden layer into an uncertainty function. Consider a neural network with input vector $e = (e_1, ..., e_n)$ and output vector $\hat{u} = (\hat{u}_1, ..., \hat{u}_m)$. The symbols e and $\hat{u}$ are chosen intentionally to be consistent with the network used as a controller in other chapters (the network input is the tracking error $e = r - y$, and the network output is the appended control signal $\hat{u}$). The network has h hidden units, input weight matrix $W_{h \times n}$, and output weight matrix $V_{m \times h}$ where the bias terms are included as fixed inputs. The hidden unit activation function is the commonly used hyperbolic tangent function. The neural network computes its output by:

$$\begin{array}{rcl} \phi_j & = & \sum_{i=1}^{n} W_{i,j} \quad e_i, \\ \hat{u}_k & = & \sum_{j=1}^{h} V_{k,j} \quad \mathsf{tanh}(\phi_j). \end{array} \tag{4.1}$$

We can write this in vector notation as

$$\begin{array}{rcl} \Phi & = & We, \\ T & = & \mathsf{tanh}(\Phi), \\ \hat{u} & = & VT. \end{array} \tag{4.2}$$

These are the equations of the applied version of the neuro-controller. This is the neuro-controller that will actually be incorporated into the physical control system. With moderate rearrangement, we can rewrite the vector notation expression in (4.2) as

$$\begin{array}{rcl} \Phi & = & We, \\ \gamma_j & = & \begin{cases} \frac{\tanh(\phi_j)}{\phi_j} & \text{if } \phi_j \neq 0 \\ 1 & \text{if } \phi_j = 0 \end{cases}, \\ \Gamma & = & \mathsf{diag}\{\gamma_j\}, \\ \hat{u} & = & V\Gamma\Phi. \end{array} \tag{4.3}$$

The function, γ, computes the output of the hidden unit divided by the input of the hidden unit; this is the *gain* of the hyperbolic tangent hidden unit. Figure 4.1 is a plot of both the *tanh* function and the gain of the *tanh* function. The gain of *tanh* is bounded in the unit interval: $\gamma \in [0, 1]$. Equivalently, we say that *tanh* is a sector bounded function belonging to the sector [0,1]. This sector boundedness is demonstrated in Figure 4.2a.

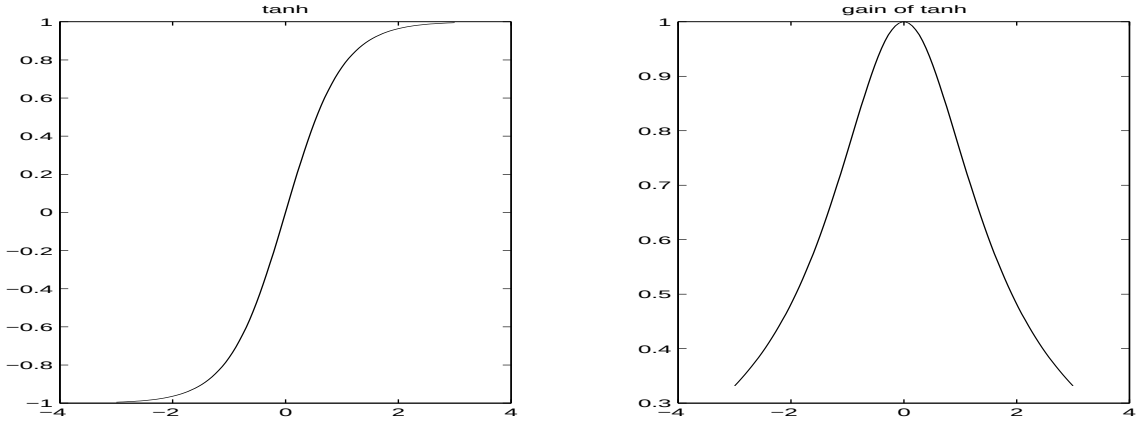
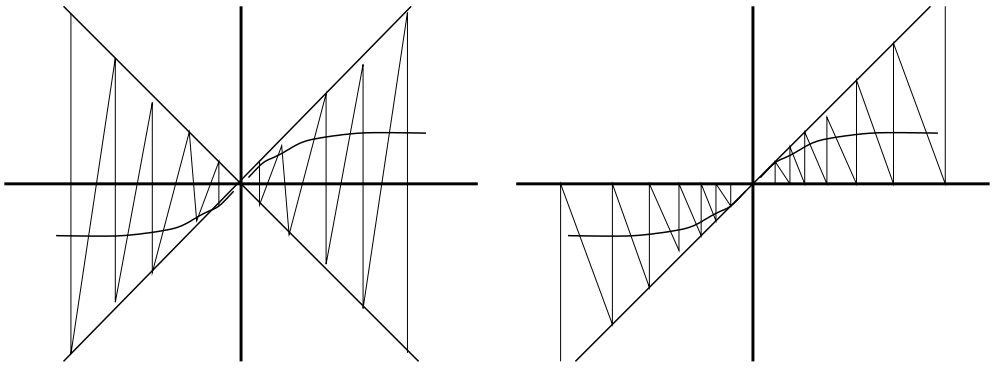


Figure 4.1: Functions: tanh and tanh gain



a. tanh in [0,1] b. improved sector

Figure 4.2: Sector Bounds on tanh

Equation 4.3 offers two critical insights. First, it is an exact reformulation of the neural network computation. We have not changed the functionality of the neural network by restating the computation in this equation form; this is still the applied version of the neuro-controller. Second,

Equation 4.3 cleanly separates the nonlinearity of the neural network hidden layer from the remaining linear operations of the network. This equation is a multiplication of linear matrices (weights) and one nonlinear matrix, Γ. Our goal, then, is to replace the matrix Γ with an uncertainty function to arrive at the testable version of the neuro-controller.

Γ is a diagonal matrix with the γ_js along its diagonal where γ_j is the nonlinear gain of the j^{th} hidden unit. We "cover" Γ by finding an uncertainty function for each γ and constructing the corresponding diagonalized matrix. To cover γ we employ an affine function:

$$\gamma \leq \bar{\gamma} + \tilde{\gamma}\delta \tag{4.4}$$

where $\bar{\gamma}$ and $\tilde{\gamma}$ are constants, and $\delta \mapsto [-1, 1]$ is the unit uncertainty function. In a strict mathematical sense, δ is simply a function that returns a real value on the range $[-1, 1]$. From an engineering perspective, δ is the system information that is unknown. We normalize the unknown to have a magnitude of unity by choosing the appropriate constants $\bar{\gamma}$ and $\tilde{\gamma}$. From a stability perspective, we can envision someone playing the role of Devil's Advocate by selecting values for δ that make our system as unstable as possible. δ is the range, or size, of values for which our system must remain stable.

As a first attempt, we bound, or cover, γ with an uncertainty region of ± 1 as shown in Figure 4.2a ($\bar{\gamma} = 0, \tilde{\gamma} = 1$). Note however, this region is double the size of the [0,1] sector; it is overly conservative. We can form a tighter inequality by choosing $\bar{\gamma} = 0.5$ and $\tilde{\gamma} = 0.5$ to arrive at $\gamma = 0.5 + 0.5\delta$. Thus, we cover γ with a known fixed part, $\frac{1}{2}$, and an unknown, variable part $[-\frac{1}{2}, \frac{1}{2}]$. The fixed part is the median value of the uncertainty, and the unknown part is one half the range. This improved sector is shown in Figure 4.2b. We now express γ as:

$$\gamma_j \leq \bar{\gamma} + \tilde{\gamma}\delta_j, \tag{4.5}$$

$$\delta_j \mapsto [-1, 1], \tag{4.6}$$

$$\bar{\gamma} = (\gamma_{MAX} + \gamma_{MIN})/2 = \frac{1}{2}, \tag{4.7}$$

$$\tilde{\gamma} = (\gamma_{MAX} - \gamma_{MIN})/2 = \frac{1}{2}. \tag{4.8}$$

Finally, we arrive at an expression for our testable version of the neural network in matrix notation:

$$\begin{array}{rcl} \bar{\Gamma} & = & \mathsf{diag}\{\bar{\gamma}\}, \\ \tilde{\Gamma} & = & \mathsf{diag}\{\tilde{\gamma}\delta_j\}, \\ \hat{u}_{\text{testable}} & = & V(\bar{\Gamma} + \tilde{\Gamma})\Phi \\ & = & V\bar{\Gamma}\Phi + V\tilde{\Gamma}\Phi. \end{array} \tag{4.9}$$

Equation 4.9 is a reformulation of the neural network; this is the testable version of the neuro-controller. It is no longer an exact representation of the network because we have approximated the nonlinear hidden unit with an uncertainty region. But the formulation given by Equation 4.9 "covers" the dynamics of the original network given in Equation 4.1. That is, any value from the applied version of the neural network function, $\hat{u}$, can be achieved in the testable version, $\hat{u}_{\text{testable}}$, by an appropriate choice of values for the uncertainty functions, δ_j's. Notice that all the $\bar{\gamma}$s and $\tilde{\gamma}$s are the same for each hidden unit, but the δ_j's are unique for each hidden unit. Also the coverage of $\bar{\Delta} + \tilde{\Delta}$ exceeds the actual function area of Δ.

At this point, we have recast the nonlinear applied version of the neural network as an LTI system plus an uncertainty component. To iterate, this testable version of the neural network is used for the stability analysis while the applied version is actually employed in the physical system as part of the controller. The next step is to apply the stability analysis tools to determine the stability status of the neuro-control system.

4.3 Static Stability Theorem: μ-analysis

Now that we have recast the nonlinear, applied, neural network into a testable version composed of linear, time-invariant components and an uncertainty function, we state our Static Stability Theorem. This version of the Static Stability Theorem is predicated upon using μ-analysis as the underlying tool to prove stability. Section 4.6 presents a nearly identical version of the same theorem using IQC-analysis.

Theorem 9 ***(Static Stability Theorem: μ-analysis version)***
Given the following:

1. *We have an LTI control system with a (nonlinear) neural network as a component. We refer to this as the applied version of the neuro-controller.*
2. *The nominal LTI closed-loop system (without the neural network) is internally stable.*
3. *The neural network is static: the hidden layer and output layer weights are held constant.*
4. *We can recast the neural network into an LTI block plus an uncertainty function. We refer to this as the testable version of the neuro-controller.*
5. *The induced norm of the testable version of the neural network is greater than or equal to the induced norm of the applied version of the neural network.*
6. *A μ-analysis robust stability tool (that permits non-LTI uncertainty) indicates that the testable version of the neuro-control system is stable.*

Under these conditions, the applied version with the full nonlinear neural network is stable.

Proof:

We separate the proof of the Static Stability Theorem into two parts. The first part primarily concerns assumptions 4 and 5. Specifically, if we recast the applied version of the neural network as an LTI system and an uncertainty function (the testable version), then the induced norm of the applied version is bounded by the induced norm of the testable version. We demonstrate that the specific uncertainty function outlined in Section 4.2 meets the induced norm assumption. The second part of the proof restates the conclusion of the μ-analysis Stability Theorem (Theorem 7 from Section 3.8).

Part I

Recall from Section 4.2, we can state the computation of a two-layer network with *tanh* activation functions in the hidden layer as:

$$\begin{array}{rcl} \hat{u} & = & V\Gamma\Phi \\ \Phi & = & We, \\ \Gamma & = & \mathsf{diag}\{\gamma_j\}, \\ \gamma_j & = & \begin{cases} \frac{\mathsf{tanh}(\phi_j)}{\phi_j} & \phi_j \neq 0 \\ 1 & \phi_j = 0 \end{cases} . \end{array} \tag{4.10}$$

This is the applied version of the neuro-controller that will be implemented in the real control system. We have manipulated the standard formulation to produce the above expression where all the nonlinearity is captured in the matrix Γ. Furthermore, the nonlinearity in Γ is posed as a hidden layer gain. We then derived the testable version of this formulation as:

$$\hat{u}_{testable} = V\bar{\Gamma}\Phi + V\tilde{\Gamma}\Phi \tag{4.11}$$

where the uncertainty function, $\delta \mapsto [-1, 1]$, is included in the diagonal matrix $\tilde{\Gamma}$. Given these two formulations of the neural network, we need to show that the induced norm of the testable version is greater than or equal to the induced norm of the applied version.

At this point, it is helpful to view the neural networks as explicit vector functions. That is, $\hat{u}$ and $\hat{u}_{testable}$ are vector functions and should be written as $\hat{u}(e)$ and $\hat{u}_{testable}(e)$ respectively.
The induced norm of a vector function, f, is given by

$$\|f\| = \max_{e \neq 0} \frac{\|f(e)\|}{\|e\|}. \tag{4.12}$$

The induced norm of a function is the maximum gain of that function. The induced norm is usually rewritten in a more familiar, yet equivalent, formulation as [Skogestad and Postlethwaite, 1996]:

$$\|f\| = \max_{\|e\|=1} \|f(e)\|. \tag{4.13}$$

We now state the induced norm of the applied version of the neural network as:

$$\|\hat{u}(e)\| = \max_{\|e\|=1} \|V\Gamma We\|, \tag{4.14}$$

and the induced norm of the testable version as

$$\|\hat{u}_{testable}(e)\| = \max_{\|e\|=1} \|V(\bar{\Gamma} + \tilde{\Gamma})We\|. \tag{4.15}$$

The difference in these two norms arises from the distinction between Γ for the applied version and $(\bar{\Gamma} + \tilde{\Gamma})$ for the testable version. These Γ's are all diagonal matrices; thus we first consider a comparison on a per element basis. In the applied version of Γ, each diagonal element is given by γ_j. Recall that γ_j is the *gain* of a nonlinear *tanh* hidden unit. Also recall that the gain is bounded by the sector $[0, 1]$. For the testable version, the diagonal elements of $(\bar{\Gamma} + \tilde{\Gamma})$ are given by $(\bar{\gamma}_j + \tilde{\gamma}_j \delta_j)$ where $\bar{\gamma}_j = \frac{1}{2}$, $\tilde{\gamma}_j = \frac{1}{2}$, and δ_j is a function whose range is $[-1, 1]$.

It is clear that each element of the testable version *covers* each element of the applied version. That is, for some particular element of the applied version, γ_j, we can select a value for $\delta_j \in [-1, 1]$ such that $\|(\bar{\gamma}_j + \tilde{\gamma}_j \delta)\| \geq \|\gamma_j\|$. The same argument extends for the other elements along the diagonals because we can choose each δ_j independently. Then, we say that the matrix function $(\bar{\Gamma} + \tilde{\Gamma})$ covers the matrix Γ. From inspection of Equation 4.14 and Equation 4.15, we arrive at:

$$\|\hat{u}(e)\| \leq \|\hat{u}_{testable}(e)\|. \tag{4.16}$$

This satisfies Part I of the proof.

Part II

In Chapter 3, we presented the formal framework for robust stability. Specifically, Section 3.8 outlines robust control theory and states the μ-analysis Stability Theorem. Part II of this proof is a straight-forward application of this theorem.

Recall that Theorem 7 (the μ-analysis Stability Theorem) has the the following conditions:

1. The known parts of the system can be grouped into a single functional box labelled M.
2. The system is internally stable. (M is a stable system).
3. The nonlinear and time-varying components are replaced by uncertainty functions.
4. The uncertainty functions have induced norms which bound the induced norms of the unknown (and possibly non-LTI) dynamics they replace.
5. These uncertainty functions are usually normalized (with constants of normalization being absorbed into the M block) and grouped together in a Δ box.

Then μ-analysis performs computations on the LFT formed by the M-Δ interconnection. Based upon the result of this computation, we can infer the stability of the original non-LTI system.

We meet these conditions of the μ-analysis Stability Theorem through the assumptions listed in the Static Stability Theorem and through the result of Part I of the proof. Thus, we apply the result of the μ-analysis Stability Theorem to conclude that the applied version of our neuro-controller is stable.

Q.E.D.

4.4 Dynamic Stability Theorem: μ-analysis version

We extend the application of robust stability theory from static stability analysis to handle dynamic stability analysis. Dynamic stability differ from static stability in that we permit the weights of the neural network to change as a result of training; this is a further violation of the LTI model. The key concept for dynamic stability analysis is to consider neural network weight updates as perturbations to the network weights. We add additional uncertainty to the network weights and then apply the μ-analysis tools to ascertain the stability status of the system. Again, we emphasize that the theorem in this section assumes the use μ-analysis as the stability tool. There is a parallel version of the Dynamic Stability Theorem for the IQC-analysis tool in Section 4.7.

To begin, we again consider an LTI system with a neural network as a non-LTI component. Recall that the network is a two-layer, feed forward network with *tanh* activation functions in the hidden layer. The network has two sets of weight matrices: $W_{h\text{x}n}$ is the input-side weight matrix and $V_{m\text{x}h}$ is the output-side weight matrix (n inputs, h hidden units, and m outputs with the bias terms included as fixed inputs). Let us form the matrix B as a diagonal matrix in which the weights of W and V are distributed along the diagonal

$$B = \begin{bmatrix} W_{1,1} & & & & & & & \\ & W_{1,2} & & & & & & \\ & & \ldots & & & & & \\ & & & W_{h,n} & & & & \\ & & & & V_{1,1} & & & \\ & & & & & \ldots & & \\ & & & & & & V_{m,h} \end{bmatrix}. \tag{4.17}$$

Matrix B is of size zxz where $z = hn + mh$. Now we form a diagonal matrix B_P (the P subscript denotes *perturbation*) also of size zxz by:

$$B_P = \begin{bmatrix} B_{p_1} & & & \\ & B_{p_2} & & \\ & & \ldots & \\ & & & B_{p_z} \end{bmatrix}, \tag{4.18}$$

where each $B_{p_i} \in \Re$. Finally, we form a diagonal matrix Δ_L (the L subscript denotes *learning*) again of size zxz:

$$\Delta_L = \begin{bmatrix} \delta_{l_1} & & & \\ & \delta_{l_2} & & \\ & & \ldots & \\ & & & \delta_{l_z} \end{bmatrix}, \tag{4.19}$$

where $\delta_{l_i} \mapsto [-1, 1]$ is the unit uncertainty function as described in the previous section on static stability. These square matrices, B, B_P, and Δ_L, are all diagonalized so that when they are multiplied together, the corresponding entries of each matrix will multiply together.

At this point it is instructive to discuss the purpose of B_P and Δ_L. Even though their semantic interpretations have little bearing on the Dynamic Stability Theorem, it is important to understand why we are adding uncertainty to the system in the form of B_P. We regard B_P as a set of individual perturbation *sizes*, one per weight in W and V, that will allow us to add uncertainty to the individual

network weights. Essentially, B_{p_i} is the amount of permissible uncertainty for the corresponding neural network weight in B. This uncertainty directly translates into how much of a perturbation we tolerate for each individual network weight. We will manipulate the perturbation sizes in B_P until the μ-analysis stability tool verifies that the overall control system is stable for all these perturbations. In the dynamic stability proof which follows, we demonstrate that given specific stipulations regarding B_P we can arrive at a stability guarantee for a neural network with dynamic weight changes. In other words, B_P is the part of the uncertainty function that covers the time-varying LTI violations of a learning neural network.

Returning to the discussion of B_P, we form the interconnection system in Figure 4.3. This arrangement is referred to as "multiplicative uncertainty" because the uncertainty in B_P is a factor of the known LTI system in B. Equivalently, Figure 4.3 has a transfer function of $B(I + B_P\Delta_L)$. Thus each element of B_P is a multiplication factor for a corresponding weight in B.

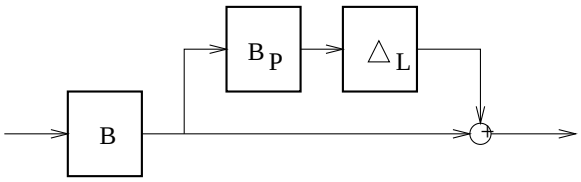


Figure 4.3: Multiplicative Uncertainty Function for Network Weights

The Δ_L matrix selects some portion of that perturbation to apply in either the positive or negative direction. Each δ_{l_i} in Δ_L can be chosen individually; because there are z different network weights, we have z degrees of freedom in selecting how to "weight" the perturbation sizes in B_P. Notice also that $\delta_{l_i} = 1$ finds the largest positive perturbation for the corresponding weight in B and $\delta_{l_i} = -1$ designates the smallest (most negative) perturbation. For example, consider the first neural network weight on the diagonal in B, $W_{1,1}$. The corresponding perturbation for $W_{1,1}$ in B_P is B_{p_1}, the upper left element of B_P. This will be multiplied by δ_{l_1} in Δ_L. If $\delta_{l_1} = 1$, then the *maximum perturbed value* for $W_{1,1}$ equals $W_{1,1}(1 + B_{p_1})$. Similarly the *minimum perturbed value* is $W_{1,1}(1 - B_{p_1})$. In this way, the Δ_L matrix has the effect of selecting a range of possible perturbed values for each weight in the neural network.

Definition 11 *The **perturbed range**, R_{W_i}, of a network weight is the continuous range of values for the network weight specified by the maximum, Max_{W_i}, and minimum, Min_{W_i}, perturbed values of the network weight.*

$$R_{W_i} = [Min_{W_i}, Max_{W_i}] \tag{4.20}$$
$$Min_{W_i} = W_i(1 - B_{p_i}) \tag{4.21}$$
$$Max_{W_i} = W_i(1 + B_{p_i}) \tag{4.22}$$

We are now ready to state the Dynamic Stability Theorem.

Theorem 10 ***(Dynamic Stability Theorem: μ-analysis version)***
Given the following:

1. *We have an LTI control system with a nonlinear, time-varying neural network as a component. We refer to this as the applied version of the system.*
2. *The nominal LTI closed-loop system (without the neural network) is internally stable.*
3. *We recast the neural network into an LTI block plus an uncertainty block to cover the nonlinear hidden layer. This process is fully described in the Static Stability Theorem. We refer to this as the testable version.*

4. *To the testable version, we introduce multiplicative weight uncertainty in the form of a diagonalized matrix as described by B_P in this section. This results in a maximum, Max_{W_i}, and minimum, Min_{W_i}, perturbed value for each weight in the neural network. These values specify a perturbed range, R_{W_i}, for each neural network weight. This introduces additional uncertainty to the system. With both uncertainty functions (one for the nonlinear hidden layer and one for the time-varying weights), we now have the full testable version of the system.*

5. *When training the neural network in the applied version of the system, the accumulated changes to each neural network weight do not cause the weight to exceed its perturbed range, R_{W_i}.*

6. *A μ-analysis stability tool (that permits non-LTI uncertainty) indicates that the testable version system is stable.*

Under these conditions, the applied version of the system with the learning, nonlinear neural network is stable.

Proof:
The proof for the Dynamic Stability Theorem immediately follows from that of the Static Stability Theorem. As is the case with the Static Stability Theorem, we separate the proof of the Dynamic Stability Theorem into two parts. In the first part, we establish that the uncertainty function covers the non-LTI dynamics of the nonlinear *tanh* hidden layer and the time-varying weights. In the second part, we show that meeting the conditions stated in the Dynamic Stability Theorem imply that the overall applied version of the system (with learning neural network) is stable.

Part I

We again need to establish the conditions for replacing the non-LTI components in the system with uncertainty blocks. That is, we need to ensure that the induced norm of the uncertainty block is no smaller than the induced norm of the non-LTI blocks by showing:

$$\|\hat{u}(e)\| \leq \|\hat{u}_{testable}(e)\| \tag{4.23}$$

where $\|\hat{u}(e)\|$ is the induced norm of the applied version (nonlinear and time-varying) and $\|\hat{u}_{testable}(e)\|$ is the induced norm of the testable version.

This result is slightly more mathematically complex than the static stability case because we have two uncertainty blocks in the dynamic stability case. When we reconfigure the system into the $M - \Delta$ feedback arrangement for the static stability case, the only source of uncertainty in Δ arises from the matrix of hidden layer gains, Γ. In the dynamic stability case, the Δ in the $M - \Delta$ arrangement contains both the hidden layer gains, Γ, and also the time-varying neural network weights, W and V. Because the Δ matrix is block diagonal, we can satisfy the overall induced norm condition on Δ by satisfying the induced norm condition on each individual part of Δ. We need to show that the induced norm of the nonlinear part is less than the induced norm of uncertainty function we use to cover it, and we need to show that the induced norm of the time-varying part is less than the induced norm of the uncertainty function we use to cover this latter part.

Regarding the nonlinear hidden layer, we need only show $(\bar{\Gamma} + \tilde{\Gamma})$ covers Γ. This fact was established in the proof of the Static Stability Theorem.

Regarding the time-varying weight changes, we must demonstrate that the uncertainty function of the testable version covers the weight changes of the applied version of the neuro-controller. Condition 5 of the Dynamic Stability Theorem supposes this very condition. The uncertainty function forms a region of permissible weight changes, R_{W_i}; as stated in Condition 5, we need only ensure that the neural network training algorithm does not exceed any of these ranges.

Because the uncertainty functions in the testable version of the neuro-controller cover both the nonlinear dynamics of the hidden layer and the time-varying dynamics of the weight changes, we can choose δ's such the induced norm of the testable version is at least as large as the induced norm of the applied version. We have met the condition in Equation 4.23 and Part I of the proof is complete.

Part II

As we did for the Static Stability Theorem, Part II of the proof for the Dynamic Stability Theorem is simply a restatement of the μ-analysis Stability Theorem (Theorem 7 in Section 3.8).

The presuppositions outlined in the Dynamic Stability Theorem and Part I of the proof meet the requirements for the μ-analysis Stability Theorem. It is then trivial to conclude that the applied version of the neuro-controller (with dynamic weight updates) is a stable system.

Q.E.D.

Again, we emphasize that the above theorems on static and dynamic stability are presented in terms of the μ-analysis stability tool. The mathematical correctness of these theorems requires that the μ-analysis stability tool accommodate non-LTI uncertainty. In practice, the only currently available μ-analysis stability tool presupposes LTI uncertainty. Thus, our application of the theorems will be technically flawed because we are using the wrong variant of the μ-analysis stability tool. However, μ-analysis is the most common software tool for research in robust stability and, for our particular uses, the LTI-uncertainty μ-analysis tool provides a very good approximation to the stability status of the neuro-controller. In the next three sections, we introduce a parallel development of static and dynamic stability in terms of the IQC-analysis tool. The current implementation of IQC software does support non-LTI uncertainty. Even though IQC-analysis is not widely used or accepted, it does provide the strict mathematical guarantees of stability not available in the current implementation of μ-analysis.

4.5 Uncertainty for Neural Networks: IQC-analysis

In Section 4.2, we showed how to convert the nonlinear portion of the neural network into an LTI block plus an uncertainty function. The analysis in this previous section assumes that μ-analysis is the tool used for stability checking. Here, in this section, we present a similar transformation from the applied version of the neuro-controller to the testable version, except that we assume IQC-analysis will be the underlying stability analysis tool. Specifically, we show how to cover the nonlinear gain in the *tanh* hidden units with an appropriate IQC function and the time-varying weight changes with another IQC function. The reader is directed to Section 3.9 for details on IQC-analysis and a list of references.

Let us first start with the nonlinear *tanh* hidden units and assume that the neural network weights are held constant. Recall that we have transformed the computation of the neuro-controller to the following equations:

$$\begin{array}{rcl} \hat{u} & = & V\Gamma\Phi, \\ \Phi & = & We, \\ \Gamma & = & \mathsf{diag}\{\gamma_j\}, \\ \gamma_j & = & \begin{cases} \frac{\tanh(\phi_j)}{\phi_j} & \text{if } \phi_j \neq 0 \\ 1 & \text{if } \phi_j = 0 \end{cases} \end{array} \tag{4.24}$$

where Γ is a diagonal matrix function that captures all the nonlinearity in the neural network.

Just as we did for the μ-analysis case, we need to find a function that covers the non-LTI dynamics in the above neural network. In this IQC-analysis case, the covering function must be of the IQC form. As discussed previously, custom building IQCs is a highly specialized task requiring extensive training in nonlinear system analysis. While this level of analysis is beyond the scope of this dissertation, the interested reader can find some of the details and further references in [Megretski and Rantzer, 1997a; Megretski and Rantzer, 1997b; Megretski et al., 1999]. Fortunately, IQCs for many common non-LTI functions have already been built and arranged in an accessible library. We will use two of these pre-existing IQCs to cover the non-LTI features of the neuro-controller.

First, we must find an appropriate IQC to cover the nonlinearity in the neural network hidden layer. From Equations 4.10, we see that all the nonlinearity is captured in a diagonal matrix, Γ. This matrix is composed of individual hidden unit gains, γ, distributed along the diagonal. In IQC terms, this nonlinearity is referred to as a *bounded odd slope nonlinearity*. There is an Integral Quadratic Constraint already configured to handle such a condition. The IQC nonlinearity, ψ, is characterized by an odd condition and a bounded slope [Megretski et al., 1999]:

$$\psi(-x) = -\psi(x), \tag{4.25}$$

$$\alpha(x_1 - x_2)^2 \leq (\psi(x_1) - \psi(x_2))(x_1 - x_2) \leq \beta(x_1 - x_2)^2. \tag{4.26}$$

Let us examine how this IQC function, ψ, "covers" the nonlinearity of the neural network hidden layer. First, Equation 4.25 requires ψ to be an odd function. Next, we examine how Equation 4.26 indicates that ψ must meet a slope-boundedness requirement. Without loss of generality, first consider the case in which $x_1 > x_2$; then Equation 4.26 reduces to

$$\alpha(x_1 - x_2) \leq (\psi(x_1) - \psi(x_2)) \leq \beta(x_1 - x_2). \tag{4.27}$$

Looking at the left-hand inequality first, we see that

$$\alpha(x_1 - x_2) \leq (\psi(x_1) - \psi(x_2)), \tag{4.28}$$

or

$$\alpha \leq \frac{\psi(x_1) - \psi(x_2)}{x_1 - x_2}. \tag{4.29}$$

Notice that the r.h.s of Equation 4.29 is the "slope" of the function. If this inequality holds for all x_1 and x_2, then the slope (or derivative) of ψ is no smaller than α. Similarly, we take the right-hand inequality of Equation 4.27 to see that the slope of ψ is no larger than β. A parallel analysis holds for the case of $x_2 > x_1$. In summary, this IQC has a bound limitted slope in the range of $[\alpha, \beta]$. For our specific use, we choose $\alpha = 0$ and $\beta = 1$ to accommodate each hidden unit gain, γ_j. Because the hidden unit gain is a nonlinear function bounded by $[0, 1]$, we know that the bounded odd slope nonlinearity is an appropriate IQC to cover the nonlinear hidden units of the neural network.

The *tanh* hidden unit function satisfies these two IQC constraints. Clearly, *tanh* is an odd function as:

$$tanh(-x) = -tanh(x). \tag{4.30}$$

It is also evident that *tanh* meets the bounded slope condition as:

$$0 \leq (tanh(x_1) - tanh(x_2))(x_1 - x_2) \leq (x_1 - x_2)^2, \tag{4.31}$$

which reduces to (assuming $x_1 \geq x_2$ without loss of generality)

$$0 \leq (tanh(x_1) - tanh(x_2)) \leq (x_1 - x_2). \tag{4.32}$$

This too is satisfied because *tanh* is a monotonically increasing function.

We now need only construct an appropriately dimensioned diagonal matrix of these bounded odd slope nonlinearity IQCs and incorporate them into the system in place of the Γ matrix. In this way we form the testable version of the neuro-controller that will be used in the Static Stability Theorem in the following section.

Before we jump into the Static Stability Theorem, we also address the IQC used to cover the other non-LTI feature of our neuro-controller. In addition to the nonlinear hidden units, we must also cover the time-varying weights that are adjusted during training. Again, we will forego the complication of designing our own IQC and, instead, select one from the preconstructed library of IQCs. The *slowly time-varying real scalar IQC* is defined by relationship [Megretski and Rantzer, 1997a]:

$$w(t) = \psi(t)v(t), \tag{4.33}$$

$$\psi(t) \leq \beta, \tag{4.34}$$

$$\dot{\psi}(t) \leq \alpha, \tag{4.35}$$

where ψ is the non-LTI function (our neuro-controller). The key features are that ψ is a scalar function of time, that ψ is finite, and the rate of change of ψ is bounded by some constant, α (a different α than used in the bounded slope odd IQC). We can view each weight in our neuro-controller as a scalar factor. Because the weights alter during neural network training, these "scalar factors" change as a function of time. If we can adjust the neural network learning rate so that there is an upper bound on the rate of change, then this IQC is covers the time-varying nonlinearity in our learning neural network.

4.6 Static Stability Theorem: IQC-Analysis

We are now again able to construct two versions of the neuro-control system. The applied version contains the full, nonlinear neural network. For this static stability section, we temporarily assume the network weights are held constant. We also are able to form the testable version by replacing the nonlinear hidden unit gains with the bounded odd slope nonlinearity IQC. We now restate the Static Stability Theorem in terms of IQC-analysis.

Theorem 11 ***(Static Stability Theorem: IQC-analysis version)***
Suppose the following:

1. *We have an LTI control system with a (nonlinear) neural network as a component. We refer to this as the applied version of the neuro-controller.*
2. *The nominal LTI closed-loop system (without the neural network) is internally stable.*
3. *The neural network is static: the hidden layer and output layer weights are held constant.*
4. *We can recast the neural network into an LTI block plus an IQC function. We refer to this as the testable version of the neuro-controller.*
5. *The IQC covers the nonlinearity of the neural network hidden unit.*
6. *The IQC-analysis robust stability tool finds a feasible solution to the IQC thus indicating that the testable version of the neuro-control system is stable.*

Under these conditions, the applied version with the full nonlinear neural network is stable.

Proof:

The proof for this theorem is straight forward. Because we have selected the appropriate IQC to address condition 5, we need only apply IQC theory (see Section 3.9) to conclude that our applied version of the neuro-controller is stable.

Q.E.D.

4.7 Dynamic Stability Theorem: IQC-Analysis

We are now ready to state the Dynamic Stability Theorem in terms of IQC-analysis.

Theorem 12 ***(Dynamic Stability Theorem: IQC-analysis version)***
Suppose the following:

1. *We have an LTI control system with a nonlinear, time-varying neural network as a component. We refer to this as the applied version of the system.*
2. *The nominal LTI closed-loop system (without the neural network) is internally stable.*

3. *We recast the neural network into an LTI block plus an IQC block to cover the nonlinear hidden layer. This process is fully described in Section 4.5. We refer to this as the testable version.*

4. *To the testable version, we introduce an additional IQC block to cover the time-varying weights in the neural network. This process is also described in Section 4.5.*

5. *When training the neural network in the applied version of the system, the rate of change of the neuro-controller's* vector function *is bounded by a constant.*

6. *The IQC-analysis stability tool indicates that the testable version system is stable by finding a feasible solution satisfying the IQCs.*

Under these conditions, the applied version of the system with the learning, nonlinear neural network is stable.

Proof:
The proof for the Dynamic Stability Theorem also immediately follows from that of the Static Stability Theorem. The conditions listed above satisfy the preconditions of IQC stability theory. We need only apply the IQC theorems to conclude that our applied version of the neuro-controller is stable.

Q.E.D.

In the next section, we sketch the basic *stable* learning algorithm. This is the concrete realization of the Dynamic Stability Theorem. The stable learning algorithm allows us to implement a learning agent with a neuro-controller and to safely integrate this agent into a control environment in which instability cannot be tolerated. The following section is meant only to introduce the high-level concept of a stable learning algorithm. Many of the details that we have overlooked will be addressed in Chapter 5.

4.8 Stable Learning Algorithm

The Dynamic Stability Theorem of the previous section naturally specifies an algorithmic implementation. Many of the previous theoretical results in neuro-control stability have assumptions in their theorems which are violated when applying the theory to practical problems. One of the key features of the Dynamic Stability Theorem is its applicability to real-life control problems without violating any of the assumptions in the proof. We combine the aspects of the Static and Dynamic Stability Theorems with a learning algorithm to arrive at the following *stable, learning algorithm.*

The stable learning algorithm alternates a stability testing phase with a learning phase. The purpose of the stability testing phase is to find the largest set of neural network weight perturbations that still retain system stability. These perturbations form a "safety region" for each weight in the network; we can move each individual weight within its safety region and still guarantee system stability. In the second phase, the learning phase, we train the network until either we are satisfied with the control performance or until one of the network weights exceeds its safety region. Then the algorithm repeats with additional series of alternating stability testing phases and learning phases. These steps are explicitly described in the following procedure:
Stable Learning Algorithm

1. We check the stability of the nominal system (without the neuro-controller). Recall that BIBO stability presupposes internal stability of the nominal system.

2. If the nominal system is stable in Step 1, then we add the neuro-controller, replace the non-LTI neural controller with an LTI uncertainty block, and then perform a static stability check with either the μ-analysis or IQC-analysis stability tools. This ensures that the initial weight values of the neuro-controller implement a stable system. Initially, we choose the network output weights to be small so that the neuro-controller has little effect on the control signal of the system. Thus, if the nominal system is stable, then the "initialized" neuro-controller is typically stable as well.

3. The next step is the stability testing phase. We compute the maximum network weight uncertainty that retains system stability. This is done with the following subroutine:

   Stability Testing Phase

   (a) For each individual weight in the neural network, we select an uncertainty factor. These uncertainty factors are the diagonal entries in the B_P matrix.

   (b) We then combine all the uncertainty into the $M - \Delta$ LFT arrangement and apply either the μ-analysis tool or the IQC-analysis tool.

   (c) If μ (or IQC) indicates that we have a stable system, we will increase each individual weight uncertainty factor. We multiply all the weights by the same factor to keep all the ratios constant. This issue is discussed in detail in Section 6.2.4 and Section 6.2.5.

   (d) Similarly, if μ (or IQC) indicates that we have an unstable system, we decrease each individual weight uncertainty by multiplying each weights by the same factor to keep all the ratios fixed.

   (e) We repeat sub-steps 3c and 3d until we have the largest set of individual weight perturbations in B_P that still just barely retain system stability. This is the maximum amount of perturbation each weight can experience while still retaining a stable control system.

4. We then use these uncertainty factors to compute a permissible perturbation range, R_{W_i}, for each individual network weight. The perturbation range is the "safety range" for each individual weight; all perturbations to a weight that keep the weight within this range are guaranteed not to induce instability.

5. We then enter the learning phase. Notice at this point we have not specified a particular learning algorithm. We could employ any learning algorithm that updates the neuro-controller weights as long as we do not violate the allowable perturbation range.
   Learning Phase

   (a) Train on one sample input.

   (b) Compute the desired weight updates.

   (c) If the weight updates do not exceed any network weight's perturbation range, update the weights and repeat with the next training example.

   (d) If the weight updates do exceed a perturbation range, stop learning with the last set of allowable network weights.

The above description is a high-level sketch of the stable reinforcement learning algorithm. At this point, we conclude the theoretical contribution of the dissertation, having established a learning algorithm which is guaranteed to be stable in a control environment.

Chapter 5

Learning Agent Architecture

Our secondary goal is to demonstrate the applicability of the Dynamic Stability Theorem to real control situations. Before we can apply the theory to several case studies, we must first construct a detailed neuro-control agent to bridge the gap between theory and practice. This agent must accommodate both the stipulations of the stability theory and also the requirements of practical control situations. In this chapter, we detail the development of this stable neuro-control agent.

First, we motivate our choice of the reinforcement learning algorithm by comparing it to alternative algorithms in Section 5.1. We then address the high-level architectural issues of the learning agent. Essentially, the high-level architecture must facilitate the dual role of the learning agent; the learning agent must act both like a reinforcement learner and a controller. Each role has specific functional requirements that sometimes conflict with the other role. We discuss how to resolve the dual nature of the learning agent in Section 5.2. We also consider the low-level architecture of the system; we select specific neural networks for implementing different parts of the agent in Section 5.3. In Section 5.4, we resolve neuro-dynamic difficulties unique to our control situation. Finally, Section 5.5 summarizes all the component architectures and presents the detailed learning algorithms.

5.1 Reinforcement Learning as the Algorithm

A myriad of learning algorithms have been developed for use in neural networks, machine learning, statistical learning, planning, and other branches of artificial intelligence. We define the *learning algorithm* to be the abstract procedure that accumulates the agent's experience and uses this experience to make decisions within an environment. The *architecture* is the physical structure that implements the algorithm. Naturally, the two concepts are codependent; thus, we should be careful to not blindly design for one of these concepts without considering the other. However, for the purposes of clear exposition, we first present the choice of a learning algorithm in this section and then defer the discussion of the specific learning architecture until the subsequent two sections.

A widely accepted taxonomy of learning algorithms classifies algorithms based on their information exchange with the environment [Haykin, 1994]. There are three broad categories of learning algorithms based upon the type of information they receive from the environment: supervised learning, reinforcement learning, and unsupervised learning listed from the most feedback information available to the algorithm to the least.

Supervised learning assumes that the environment provides a teacher. The supervised learner collects information from the environment and then produces an output. The teacher provides feedback in the form of the "correct" output. The supervised learner changes its internal parameters to produce the correct decision next time it observes the same state. The standard back propagation algorithm is an example of supervised learning. In the control situation, we typically do not know a priori what the optimal control action is for any given state. The optimal control action is dependent on the objective of minimizing tracking errors over time and on the complicated dynamics of the

plants. Consequently, we cannot use supervised learning as the primary learning algorithm [1].

The unsupervised learner observes system states and produces outputs. However, an unsupervised agent receives no feedback from the environment. Instead, it adjusts parameter vectors to capture statistical tendencies in the frequency and distribution of the observed states. Kohonen has proposed a number of clustering algorithms that are all unsupervised in nature [Kohonen, 1997]. We could employ unsupervised learning in this situation; however, we would not be using all the information available to us. While we cannot know the optimal control action, we can measure its performance by accumulating the tracking error over time. The unsupervised algorithm does not use this information. Furthermore, the result of an unsupervised algorithm, statistical information about the system, is really not the desired result. We desire a controller to produce control actions per each observed state.

In between the extremes of supervised and unsupervised algorithms is reinforcement learning. Upon observing the system state and producing an output, the reinforcement learner receives an evaluation signal from the environment indicating the utility of its output. Through trial and error, the reinforcement learner is able to discover better outputs to maximize the evaluation signal. The reinforcement learner adjusts its internal parameters to improve future outputs for the same observed state. Thus, reinforcement learning is an ideal choice, because the information needs of the algorithm exactly match the information available from the environment.

Other aspects of reinforcement learning also match well with our control environment. Primary among these is the ability of a reinforcement learner to optimize over time [Sutton and Barto, 1998]. Specifically, we desire to minimize the mean squared tracking error over time. For most control problems, each control action affects not only the immediate tracking error of the next time step, but also the tracking error at time steps into the future. Thus, the goal of each control action is to minimize the *sum* of future tracking errors. The statistical sampling nature of reinforcement learning permits the optimization of control decisions without having to keep track of lengthy tracking error sums [Sutton and Barto, 1998; Kaelbling et al., 1996].

Closely associated with optimization-over-time is the ability of a reinforcement learner to naturally handle delays, or time-constants, in the system. Suppose that the system is a discrete-time system with a delay of two time steps. Any action made at time step k will not affect the output at time $k+1$, but instead, first affects the output at time $k+3$. Delays are a common feature in many of the control systems of interest. Often, the delays in these systems are of unknown duration and can be highly variable for MIMO systems (there will be different delay dynamics from each input to each output). Naturally, this greatly complicates the design procedure. Again, because of the statistical sampling nature of reinforcement learning, the learning agent "discovers" these delays and learns a control function that is appropriate for the delays. In reinforcement learning, the problem of delayed rewards is referred to as the *temporal credit assignment problem* [Sutton and Barto, 1998]; reinforcement learning provides an elegant solution.

Reinforcement learning is structured for a trial-and-error approach to learning. Most reinforcement learning algorithms have an exploration factor that can be tuned to vary the degree to which the agent tries new actions (high exploration) versus exploits accumulated knowledge (low exploration). Typically, the exploration factor is set high initially and then "annealed" as the learning progresses. This tendency for exploration allows the reinforcement learner to discover new and possibly better control actions. Through repeated attempts at novel control actions, the reinforcement learning algorithm accumulates information used to produce improved control. It is this feature of reinforcement learning that allows adaptive control algorithms to outperform traditional control techniques on complex plants. However, it is also this feature that forms the most significant drawback of reinforcement learning for this application; intermediate control functions acquired during the learning process often perform poorly and are frequently unstable.

In Section 5.5, we explicitly list the entire stable learning algorithm including the detailed reinforcement learning portion. More detailed reviews of learning algorithms can be found in [Hertz et al., 1991; Hassoun, 1995; Rumelhart et al., 1986b]. For more detail on reinforcement learning in

[1] We will use a form of back propagation to adjust a part of the neuro-controller, but the nature of the algorithm is based on reinforcement learning principles

particular, consult [Sutton and Barto, 1998; Kaelbling et al., 1996]. In Section 5.5, we explicitly list the entire stable learning algorithm including the detailed reinforcement learning portion.

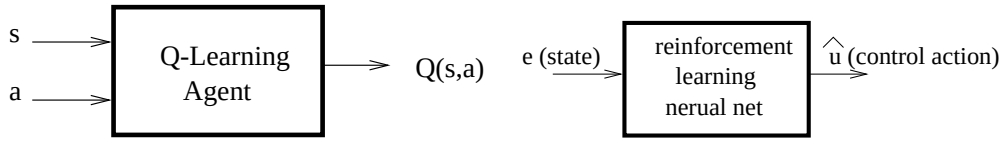
5.2 High-Level Architecture: The Dual Role of the Learning Agent

Among our top concerns in selecting a specific high-level architecture for the learning agent is the need to balance the algorithmic requirements of reinforcement learning with the functional aspects of a control task. By high-level architecture, we mean the design features of the learning agent which accommodate the unique controller environment.

Essentially, the agent must act like a reinforcement learner by storing and manipulating value functions, by performing policy evaluation, and by performing policy improvement. In contrast, the agent must also act like a controller which requires an additional set of functions from the agent. Primary among these requirements, the agent must provide the correct control action for a given state; that is, it must implement a policy. Furthermore, we stipulate that the control actions be available in real-time. This prohibits the storage of the policy in a format that is not readily accessible. The stability proofs for our neuro-controller also place restrictions on the role of the agent as a controller.

The dichotomy of the agent as a reinforcement learner and the agent as a controller is depicted in Figure 5.1. Shown in Figure 5.1a is the canonical representation of a specific type of reinforcement learner: a Q-learning agent [Watkins, 1989]. A state/action pair are input to the Q-learner to produce the value function representative of the input pair. This value function is commonly referred to as a *Q-value*. The Q-learning agent specifically implements the reinforcement learner by storing value functions. Via these value functions, the Q-learner can perform policy evaluation and policy improvement. [Sutton and Barto, 1998; Kaelbling et al., 1996] provide detailed information on Q-learners and value functions.

Contrast this with the diagram in Figure 5.1b showing the agent as a controller. Input to this agent is the current state of the system; the output is the agent's control signal. This agent implements a policy.



a. Reinforcement Learner b. Controller

Figure 5.1: Reinforcement Learning and Control Agents

Many neuro-controllers designed by the reinforcement learning community implement the Q-learner of Figure 5.1a. This specific high-level architecture suits the requirements of the agent-as-reinforcement-learner, but does not fulfill the duties of the agent-as-controller. A subtle but important point of discussion is that the Q-learner does not store a policy. There is no explicit function in which the state is input and an action output. The question naturally arises, if this agent is incapable of storing an explicit policy, then why is this type of learning architecture so common (and successful) in control problems found in the reinforcement learning research literature? The answer to this dilemma is that while the Q-learner is unable to compute an explicit policy, the agent can implement an implicit policy. Because the value function is stored in the Q-learner, it is able to perform a search over the set of actions to find the best action for a particular state; this "best action" is the one with the largest value function. Thus, implicitly, the optimal policy can be computed via a search procedure [Sutton and Barto, 1998].

While the implicit policy is satisfactory for many control tasks, it still falls short of our requirements in two important ways. First, the search procedure is potentially quite lengthy and complex.

We stipulate that the control actions of the agent must be available in real-time. In some dynamic systems, the implicit policy computation time may exceed the time window in which we must produce a control action. The second, and more important, failure of the Q-learner concerns the robust stability analysis of the previous chapter. We require an explicit policy function to determine whether it meets the stability requirements; the policy must be available as an explicit mathematical function. The implicit policy search is not amenable to the stability analysis.

Because of these difficulties, we cannot utilize the common Q-learning, high-level, architecture for our agent. We require an agent with a split personality – an agent that explicitly implements both architectures in Figure 5.1. We fall back upon early efforts in reinforcement learning architecture to utilize the actor-critic design. Here, we highlight the features of this dual architecture; more information can be obtained for actor-critic architectures in [Sutton and Barto, 1998; Barto et al., 1983; Witten, 1977]. The actor-critic architecture has two different networks, one to implement the reinforcement learner (critic), and one to implement the controller (actor). This arrangement, depicted in Figure 5.2, is copied from Sutton and Barto's text on reinforcement learning [Sutton and Barto, 1998].

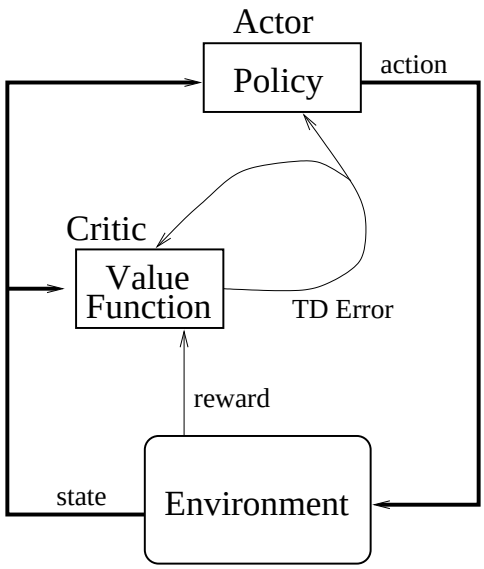


Figure 5.2: Actor-Critic Agent

The actor network can be thought of as the control agent, because it implements a policy. The actor network is part of the dynamic system as it interacts directly with the system by providing control signals for the plant. The critic network implements the reinforcement learning part of the agent as it provides policy evaluation and can be used to perform policy improvement. This learning agent architecture has the advantage of implementing both a reinforcement learner and a controller. Because the policy is computed explicitly in the actor network, we can meet the real-time demands of the control system. Also, we can select an actor network architecture that is amenable to the robust stability constraints. Note that the critic network is not actively connected to the control system; it is a "meta-system" that guides the changes to the actor network.

The drawback of the two-network architecture is a more complex training algorithm and extended training time [Sutton and Barto, 1998]. The primary reason why the Q-learning architecture is frequently used today is the simplified training algorithm. The Q-learner is not only easier to code, but more significantly, the neuro-dynamics of the learning agent are greatly simplified. Because we must use the actor-critic design for our high-level architecture, we are faced with additional neuro-dynamic problems that will be detailed in the next section. At this point we have crafted a suitable high-level architecture for our learning agent. Next we turn our attention to the low-level design of the system. Namely, we must select a specific neural network for both the actor and critic

components.

5.3 Low-Level Architecture: Neural Networks

Splitting the learning agent into two components, actor and critic, introduces a number of technical problems. We solve these problems with judicious design choices of specific neural networks for both the actor and critic networks.

We begin by selecting an architecture for the actor network (the controller). For the actor, we select the two-layer, feed forward network with *tanh* hidden units and linear output units. This architecture explicitly implements a policy as a mathematical function and is thus amenable to the stability analysis detailed in Chapter 4. Because this architecture is a *continuous network* rather than a discrete network, the actor will be able to provide better resolution and more closely learn the desired control policy [2]. Also, the continuous network is better suited to the stability analysis, because a discrete network would require a piece-meal approach to the stability analysis rather than the one-shot analysis possible with the continuous network. Another advantage of this network arises because it is a global network rather than a local network; thus, the network would likely be able to learn the control policy faster [3].

It is likely that other neural architectures would also work well for the actor net provided they met the conditions of the stability theorems. Here, we intend only to demonstrate that the stability theory is applicable to real control problems by using some neural architecture; it is not our goal to demonstrate that the two-layer, feed forward network is the best choice. The two-layer, feed forward, neural network is a common and widely studied neural architectures and also this architecture satisfies the requirements of the stability theorems; thus, we utilize this architecture for our neuro-controller.

Next we turn our attention to the critic network (the reinforcement learner). Recall that the critic accepts a state and action as inputs and produces the value function for the state/action pair. The state is the tracking error, e, and the action is the control signal, $\hat{u}$. The key realization is that *the critic network is not a direct part of the control system feedback loop and thus is not limited by the stability analysis requirements.* For the case studies of the next chapter, we originally implemented several different architectures for the critic network and found that a simple table look-up mechanism is the architecture that worked best in practice [4].

A CMAC (Cerebellar Model Articulation Controller) network is a more sophisticated variant of table lookup methods [Sutton, 1996; Miller et al., 1990] which features an improved ability to generalize on learning examples; often CMAC networks require more training time than table look-up methods to reach their final weight values. We find that CMAC also worked well, but the table look-up provided nearly the same control performance and required less training time than the pure table look-up architecture.

The reasons for finally arriving at the table look-up scheme for the critic network are quite complex; the table look-up architecture was selected because it overcame several neuro-dynamic problems that arose due to the multiple feedback loops present in the neuro-controller. The reader who is not interested in these details may skip the next section and move to the discussion of a reinforcement learning algorithm in Section 5.5 without loss of continuity. However, the analysis and solution to these neuro-dynamic problems is an important aspect in the design and implementation of the neuro-controller. Thus, we include these details in the subsequent section.

---

[2] A discrete network has a discrete function mapping from the input space to the output space while the continuous network has a continuous mapping. See [Haykin, 1994; Royas, 1996; Hertz et al., 1991] for more details

[3] A global network has "activation" inputs that typically extend over the a large part of the input space whereas a local network is only activated by inputs in a small, localized region of the input space. See [Haykin, 1994; Royas, 1996; Hertz et al., 1991] for more details.

[4] Some researchers in Artificial Intelligence do not classify a table look-up architecture as a true neural network

5.4 Neuro-Dynamic Problems

In this section, we first discuss how the complex, feedback loops of the actor-critic controller introduce two neuro-dynamic problems, and then we show why the a table look-up architecture for the critic network solved these problems.

The first circular relationship in our agent stems from the interaction between the actor network and the plant. The actor network receives the tracking error e from the plant to produce a control signal, $\hat{u}$. This control signal is fed back to the plant and thus affects the future state of the plant. The actor network is trained on the input/output pairs it experiences. Because the actor net is tied to a plant, the state of the plant dictates the training examples for the actor network. Our first circular relationship arises between the actor network which directs the state of the plant, and the plant which dictates the training inputs for the actor network.

The circular relationship becomes more complex with the addition of the critic neural network. The critic network receives its inputs (tracking error, e, and control signal, $\hat{u}$) from the plant and actor net respectively. Thus, the plant and actor net determine which training examples the critic will experience. In turn, the critic network forms the gradient information that is used to train the actor network; the critic network causes changes in the actor network.

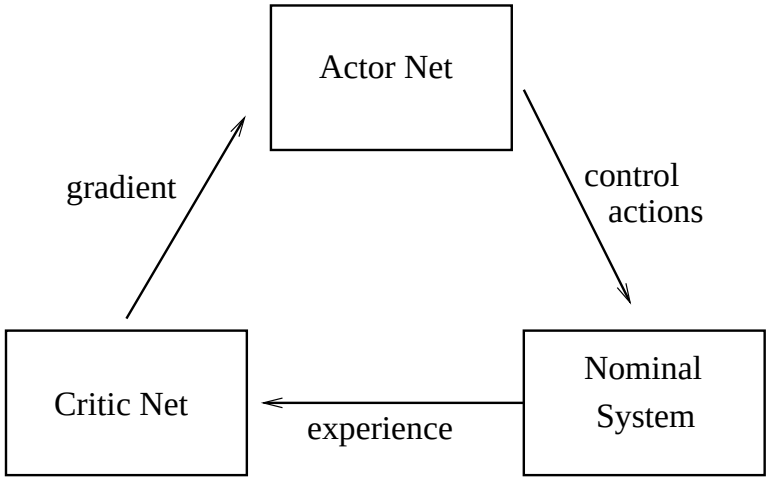


Figure 5.3: Circular Causality in the Actor-Critic Architecture

In summary, the plant affects both the actor network and critic network by dictating which training samples the neural networks experience. The actor network provides an input to the plant that determines its future states. The critic network provides a training gradient to the actor network to determine changes in the actor network's policy. As a result of this complex interaction, the training examples for the neural networks exhibit a *chaotic* sampling distribution; the training samples will be similar for extended periods of time, then suddenly, the training samples will shift widely into a different "operating regime".

Intuitively, the sampling policy is similar to the the Lorenz Attractor [Gleick, 1987] which looks like two circular rings adjoined at one intersection point. In this attractor, the state of the dynamic system is confined to one of two nearly circular orbits for long periods of time; then the system state will suddenly jump into the other orbit. It is the nature of chaotic mathematics that this jump between operating regimes cannot be predicted. In the same way, the plant state (and hence training examples) is confined to a small region of the plant state space for large periods of time. Then the plant state quickly changes to a different confined region of the state space.

The chaotic sampling causes problems in convergence of the weights for the two neural networks. Because training inputs are concentrated in one small part of the state space for a long period of time, the neural network tends to "memorize" this small operating regime at the expense of remembering

training examples from other parts of the state space. We introduce two new terms, value migration and value leakage to describe this neuro-dynamic phenomena. *Value migration* occurs when a neural network weights change so as to leave some parts of the state space under represented while other parts of the state space are heavily concentrated with neural network resources. We define *value leakage* as the phenomena in which the function learned by a neural network is incorrectly changed as a result of a non-uniform sampling distribution.

Value migration occurs when neural network learning resources are incorrectly distributed within the state space of training examples. Here, we examine this general phenomena as it specifically applies to the critic network. Recall the critic network produces an output (the value function) in response to a state/action input pair, $(e, \hat{u})$. The $(e, \hat{u})$ inputs can be viewed as k-dimensional input vectors, and each input sample as a point in $\Re^k$ space [5]. The critic network is a function: $\Re^k \mapsto \Re$. While $\Re^k$ is infinitely large, in practice the training inputs tend to be concentrated in a smaller subspace of $\Re^k$ determined by the physical limitations of the plant and the control signals of the actor network. The critic network does not need to map a function from the entire domain of $\Re^k$ to $\Re$, but instead only needs to map from a restricted region of $\Re^k$ to $\Re$.

The neural network that implements the critic network uses *computational resources* to perform this mapping. Depending on the specific neural architecture, these computational resources are hidden *tanh* units, rbf units, tables in a table look-up scheme, splines, or other related neural network components (see [Hertz et al., 1991; Haykin, 1994] for a detailed description of neural networks as geometrical space mappings). Because there are only a finite number of computational resources available to a neural network, it is critical that the network locate them within the input space appropriately so as to provide the best possible approximation to the function being learned. For example, it does not make sense to locate hidden sigmoid units within a region of the state space where there will be no training samples; this is a waste of the network's computational resources. Because the correct locations for training resources is not usually known a priori, part of the training routine for a neural network is to place the network's computational resources in the correct locations in the state space; that is, the network's resources should be located among the dense areas of training samples in the input state space.

Typical neural network training algorithms make small changes to the computational resources (for example, adjusting the input-side weights for a two-layer, feed forward network) to locate the computational resources closer to input training vectors. Usually, these algorithms require a uniform sampling distribution so that training samples are drawn from all parts of the state space. This prevents network resources from migrating toward any one over-sampled region of the input space. This is precisely the problem that occurs with our chaotic sampling in the on-line, neuro-control framework, because we do not have control over the distribution of training samples.

To overcome value migration, we must either ensure a uniform sampling distribution or we must fix the computational resources in parts of the state space a priori; the latter option is the only viable option for our neuro-control scheme. Through inspection of the tracking error, e, and by knowing bounds on the actor network's output signal, $\hat{u}$, we can estimate the subregion of the state space, $\Re^k$, in which the training samples will be drawn. Our table look-up network, evenly distributes computational resources (table entry boundaries) among this subregion of the input state space. We do not change the table boundaries as a result of training experience. The disadvantage is that we cannot fine-tune the computational resources to better fit this subregion of training samples; but the advantage we can is that our table look-up network is immune to value migration.

The second neurodynamic problem, value leakage, occurs when the network's output function is adversely affected by an uneven sampling distribution. Technically, value migration is a subset of value leakage, but here, we use value leakage in reference to additional neurodynamic problems. Consider a neural network whose computational resources are statically located within the input state space (the network resources will not change as a result of training). Additionally, let these computational resources either be global or be local with continuous activation functions. Most neural network training algorithms update the output function (output weights, table entries) in proportion to the activation of the computational resources.

[5] The actual numerical value for k depends on the dimension of e and $\hat{u}$.

Resources that are global, will be fully activated by a large number of inputs because each unit's region of activation extends globally across the input state space. The output values corresponding to these resources will change as a result of input samples that fall within the region of activation. Resources that are local and continuous are fully activated only by those input samples that fall within the local region of activation. However, these same resources will be slightly activated for input samples that occur near to the local region of activation. Thus, these local, continuous resources receive a full training update for a small number of input samples, but receive slight training updates for other training inputs adjacent to the activation region.

These type of neural network computational resources, global or continuous and local, require training samples that are uniformly distributed among the input state space. With our chaotic training samples, the network receives a large number of locally concentrated training samples at a time. This disrupts the output values of either global resources or nearby continuously local resources. As a result, the neural network's weights are unable to converge to the appropriate values.

To overcome this problem, we select a neural network whose computational resources are local and discrete. The table look-up method meets both of these requirements. These and other neurodynamic problems are an area of active research. A few groups have begun to answer parts of the overall problem; the interested reader is directed to [Kretchmar and Anderson, 1997; Kretchmar and Anderson, 1999; Anderson, 1993; Sutton, 1996; Sutton and Barto, 1998] for more information.

5.5 Neural Network Architecture and Learning Algorithm Details

In this section, we present the details of the neural networks and the robust, stable, reinforcement learning algorithm. Figure 5.4 depicts the actor and critic networks. A summary of the details for each component is listed here:

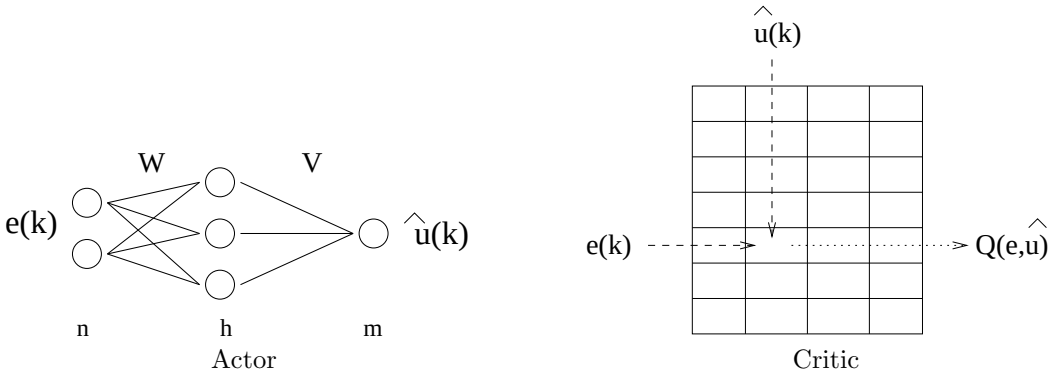


Figure 5.4: Network Architectures

Actor Net

- Feed-forward, two-layer, neural network,
- Parameterized by input and output weights, W and V,
- n (# inputs) determined by the control task. For most tasks, this includes the tracking error and possibly additional plant state variables. Also included is an extra variable held constant at 1 for the bias input.

- m (# outputs) determined by the control task. This is the number of control signals needed for the plant input.
- h (# hidden units) A free variable we can choose to be smaller for faster learning or larger for more expressive control functionality.
- *tanh* hidden unit activation functions,
- linear output unit activation functions,
- $e(k)$ is the input signal at time k. The signal is composed of the tracking error and additional plant and controller internal state variables. Also includes the bias input set to 1.
- $\hat{u}(k)$ is the output signal at time k. Computed by the actor net via feed forward computation:

$$\begin{array}{rcl} \phi_j & = & \sum_{i=1}^{n} W_{i,j} \quad e_i, \\ \hat{u}_k & = & \sum_{j=1}^{h} V_{k,j} \quad \mathsf{tanh}(\phi_j). \end{array}$$

- Trained via back propagation (gradient descent). Training example provided by critic net.

Critic Net

- Table look-up,
- Parameterized by table, Q,
- $n-1+m$ inputs determined by the control task. The input to the critic network includes the actor net input, $e(k)$ (without bias term) and the actor net output, $\hat{u}(k)$ signals. The actor net input has $n-1$ signals (without bias term) and the actor net output has m signals for a total $n-1+m$ input signals to the critic network.
- A single output, the value function $Q(e, \hat{u})$.
- Trained via SARSA, a variant of reinforcement learning [Sutton, 1996].

The learning algorithm is composed of two primary phases: a stability phase in which we use μ or IQC to compute the largest set of perturbations that can be added to the actor net weights while still keeping the overall system stable, and a learning phase in which we use reinforcement learning to train both the actor and critic neural networks. We start with the high-level description of the algorithm and then present the details of each of the two phases. Figure 5.5 lists the steps in the high-level description, while Figure 5.6 and Figure 5.7 detail the steps in the stability and learning phases, respectively.

We now describe each step of the **Stability Phase** algorithm as given in Figure 5.6:

- Step 1:
  The inputs to this routine are the control problem, P, and the current actor network weights in W and V.
- Step 2:
  We must select initial values for the perturbations in the B_P matrix. We have z degrees of freedom in selecting the perturbations. Recall that z is the number of weights in the actor network, $z = nh + hm$. We opt to initialize the perturbations so that they are all some fixed constant times their corresponding weight in B. There are other ways to assign the initial values for perturbations. In Sections 6.2.3 and 6.2.4 we discuss why we use this particular method for initializing the perturbations.

Initialize:
W,V = arbitrarily small random numbers
$Q = 0$ (set table look-up entries to 0)
Repeat
/* **Stability Phase** */
/* computes maximum weight perturbations */
B_P = stability_phase(P,W,V);

/* **Learning Phase** */
/* trains network weights */
W,V,Q = learning_phase(P,W,V,Q,B_P,c)

Until (desired tracking performance level achieved OR
no further tracking performance obtained)

Figure 5.5: Stable Reinforcement Learning Algorithm

- Step 3:
  B_{base} stores the initial set of perturbations. We will be assigning the actual perturbations in B_P by multiplying B_{base} by some other constants. $minf$ and $maxf$ are two such constants. We will compute $B_P = minf * B_{base}$ and $B_P = maxf * B_{base}$ as two possible perturbation matrices to use when computing μ. In general, we would like $minf$ to be the largest constant for which μ indicates stability and $maxf$ to be the smallest constant for which μ indicates instability. Eventually, $minf$ acts like a lower bound for stability, and $maxf$ acts like an upper bound. However, in this initial step, we seed them both, $minf$ and $maxf$, with the value of 1.

- Step 4:
  We arrange the system into an LFT and compute μ. For all future μ computations, we use this same LFT arrangement except that we substitute in new values for B_P, the perturbation matrix.

- Step 5:
  If the initial value for $maxf = 1$ produces a stable system for μ, then we will double the value of $maxf$ until $B_P = maxf * B_{base}$ produces an unstable μ value. Thus, $maxf$ lies in the set of constants producing unstable systems.

- Step 6:
  Similarly, if the initial value for $minf = 1$ produces an unstable system for μ, then we will halve the value of $minf$ until $B_P = minf * B_{base}$ produces a stable μ value. Thus $minf$ is in the range of constants that produce stable systems.

- Step 7:
  At this point, $minf$ is a constant that produces stable systems, and $maxf$ is a constant that produces unstable systems. Our objective is to find the constant in between $minf$ and $maxf$ for which the system is still just barely stable. This "cross-over point" represents the maximum perturbation sizes the system can handle and still be stable. In this step of the algorithm, we use binary search to halve the distance between $minf$ and $maxf$ on each loop pass. We stop when we are arbitrarily close to the cross-over point (5% in our algorithm).

- Step 8:
  $minf$ is now very close to the cross-over point into instability. Yet $B_P = minf * B_{base}$ still produces a stable system. We return this B_P matrix as a result of the stability phase of the algorithm.

We now describe each step of the **Learning Phase** algorithm as listed in Figure 5.7:

1. Inputs:
   - P: The control system (used for μ or IQC calculations),
   - W,V: The current neuro-controller weights which form B.
2. Initialize the individual neural network weight perturbations in B_P. Set each perturbation, B_{p_i}, proportional to its corresponding weight in B. (The rationale for keeping the perturbations proportional is discussed at length in Section 6.2.3 and Section 6.2.4 in the next chapter).

$$B_{p_i} = \frac{B_i}{\sum B}$$

3. Set: $B_{base} = B_P$, $minf = 1$, $maxf = 1$
4. Arrange the overall system, P, and the LTI uncertainty (with B_P) into the $M - \Delta$ LFT. Compute μ (or IQC).
5. If μ (or IQC) indicates that the system is stable, then
   While (system is stable) do
   Begin
   $maxf = maxf * 2$
   $B_P = B_{base} * maxf$
   recompute μ (or IQC)
   End
6. Else if μ (or IQC) indicates that the system is not stable, then
   While (system is not stable) do
   Begin
   $minf = minf \div 2$
   $B_P = B_{base} * minf$
   recompute μ (or IQC)
   End
7. Reduce the range between $minf$ and $maxf$ by:
   While ($\frac{maxf - minf}{minf} < 0.05$)
   Begin
   $test = minf + (maxf - minf)/2$
   compute μ for $B_P = B_{base} * test$
   if stable, then $minf = test$, else $maxf = test$
   End
8. Return $B_{base} * minf$

Figure 5.6: Stability Phase

1. Inputs:

   - P: The system (used for μ or IQC calculations),
   - W,V: The current neuro-controller weights.
   - Q : The current table look-up values.
   - B_P: Set of actor net perturbations (computed in stability phase).
   - c : A criteria for halting the training.

2. Initialize:

   - e = current state of system (tracking error and possibly other variables).
   - $\hat{u}$ = current actor net control action.

3. Take control action $u = u_c + \hat{u}$ and observe new state (tracking error) e'.

4. Choose next control action: $\hat{u}' = \epsilon-$greedy(e).

$$\left.\begin{array}{rcl} \Phi & = & \tanh(We') \\ \hat{u}' & = & \Phi V \end{array}\right\} \quad \text{with probability } 1-\epsilon$$
$$\hat{u}' = \Phi V + \text{ random from } 0.1(\hat{u}_{MAX} - \hat{u}_{MIN}) \quad \text{with probability } \epsilon$$

5. Train critic network:

$$Q(e,\hat{u}) = Q(e,\hat{u}) + \alpha(\gamma(r - y + Q(e',\hat{u}')) - Q(e,\hat{u}))$$

6. Compute desired actor net output: $\hat{u}^* = \text{gradient\_search}(Q(e,*))$

7. Train actor network:

$$\begin{array}{rcl} V & = & V + \beta_1 \Phi(\hat{u}^* - \hat{u}) \\ W & = & W + \beta_2 eV(1-\Phi^2)(\hat{u}^* - \hat{u}) \end{array}$$

   If W and V exceed perturbation ranges R, then retain previous values of W and V and exit learning phase.

8. Update state information: $e = e'$, $\hat{u} = \hat{u}'$

9. If perturbation criteria c is met, then exit learning phase. Otherwise, goto Step 3.

Figure 5.7: Learning Phase

- Step 1:
  The inputs to this routine are the control system (P), the actor network weights (W and V), and the critic network weights (Q). The allowable actor network perturbations, B_P, are also input. These perturbations are computed in the stability phase; we use them to determine the allowable perturbation ranges, R, for each weight in the actor net. The final input is c, the halting criteria. The learning phase of the algorithm will continue until training causes one of the actor network weights to exceed its perturbation range. If this never happens, then the algorithm would proceed indefinitely. To prevent an infinite loop situation, we provide additional halting criteria. In our control tasks, this takes the form of a fixed number of training trials; after we train for the maximum number of samples, we exit the learning phase.

- Step 2:
  We compute the current tracking error e by subtracting the plant output y from the reference signal r. We then add any additional signals required by the actor network inputs to form the e vector. This "state vector" e is then fed to the actor network to produce the control vector $\hat{u}$.

- Step 3:
  We first combine the actor net control vector $\hat{u}$ with the nominal controller vector u_c to produce the overall control vector u. We feed the control vector to the plant and observe the new plant output y'. We then compute the new tracking error e'.

- Step 4:
  We use a well-known algorithm called $\epsilon-$greedy to compute the next actor network control action $\hat{u}'$. The $\epsilon-$greedy algorithm uses the actor network output with probability $1-\epsilon$ or adds a small random perturbation to the actor network output with probability ϵ. This provides a natural *exploration* capability to allow our actor network to search for better actions and not converge to a suboptimal control policy too quickly. See [Sutton, 1996] for details on the $\epsilon-$greedy algorithm.

- Step 5:
  The critic network table stores the value function for our system. The table is indexed by state/action pairs: $Q(e, \hat{u})$. Each entry in the table refers to the value of a particular state/action pairing. Recall that *value* refers to the sum of the future tracking errors over time. For example, if our system is currently at state e and we selected control action $\hat{u}$, then we should expect our future sum of tracking errors to be $Q(e, \hat{u})$. If our system then moves to the next state/action $(e', \hat{u}')$, we should expect our sum of future tracking errors to be $Q(e', \hat{u}')$. If we add the current tracking error, $r - y$, to $Q(e', \hat{u}')$ we should expect this to be equal to $Q(e, \hat{u})$. Any difference between these two quantities is called our *temporal difference error*. In reinforcement learning, we compute the temporal difference error and then perform gradient descent to update the table entries so as to minimize the temporal difference error. The learning rate is given by α. One additional feature is the discount factor of γ [6]. For control tasks that do not typically end after a finite number of steps, the sum of future control errors will grow to infinity and all the values in the critic network table look-up will grow to infinity. For these tasks, we use a *discount factor* identified by the constant γ to keep the Q-values in the critic network finite. We use 0.95 and 0.90 for our discount factors in different tasks. See [Sutton, 1996] for detailed information on discount factors in infinite horizon, temporal difference learning algorithms.

- Step 6:
  In the next step, we use the back propagation algorithm to train the actor network. Since back propagation is a supervised learning algorithm, we need a training exemplar, $\hat{u}^*$, for the

[6] The γ used here is not to be confused with our earlier use of γ when converting the nonlinear *tanh* hidden layer to an uncertainty block. Reinforcement learning algorithms have traditionally used the symbol γ as a constant discount factor.

actor network. We use the information in the critic network to compute the training exemplar. For example, the system is currently in a state specified by the tracking error e. Using e as an input, the actor network produced a control signal as output, $\hat{u}$. This might not be the best control signal; there may be a better control signal, $\hat{u}^*$ which minimizes the sum of future tracking errors. Because the critic network stores the value functions (sum of future tracking errors), we can use the critic to find the "optimal" control action $\hat{u}^*$. First we define a local neighborhood around the actor network's current output, $\hat{U}_{ln}$. We do not want to search globally for $\hat{u}^*$ because we want our actor net to make small incremental adjustments to its control function output. We map out a grid of control actions within the neighborhood $\hat{U}_{ln}$ to find the control action with the smallest value function according to the critic network. We use this value as the training exemplar for the actor net.

$$\hat{U}_{ln} = \text{small local neighborhood of } \hat{u}$$
$$\hat{u}^* = \min_{\hat{u} \in \hat{U}_{ln}} Q(e_{ln}, \hat{u})$$

- Step 7:
  This step is the standard back propagation algorithm for a two-layer, feed forward neural network with *tanh* hidden unit activation functions and linear output unit activation functions. See [Hassoun, 1995; Rumelhart et al., 1986a; Haykin, 1994; Hertz et al., 1991] for more information on the back propagation algorithm. Importantly, in this step we test to see if the weight updates to the actor network would exceed the perturbation ranges specified by B_P. If the perturbation range would be exceeded by the update, then we do not perform the weight update, exit the learning phase, and return the current network weight values W, V, and Q.

- Step 8:
  We update the state information.

- Step 9:
  At this point, we test to see if we have met the additional halting criteria specified by input c. If we have not, then we repeat the algorithm for a new training sample. Again, examples of halting criteria include a stopping after a maximum number of training iterations or a condition to halt when the network has ceased to improve control performance. If we do not include the additional halting criteria, then the learning phase might continue indefinitely as it is possible that the neural network weights may never exceed the perturbation ranges.

Chapter 6

Case Studies

In the previous two chapters, we develop both the theory and the practical architecture for a robust reinforcement learning neuro-control agent. Specifically, Chapter 4 introduces a framework in which a neural network is recast as an LTI system; we present conditions on this network such that the resulting control system is stable with fixed network weights (static stability) and varying network weights (dynamic stability). Chapter 5 concerns the details of constructing an agent suitable for implementing the static and dynamic stability theory. Also in Chapter 5, we discuss practical design decisions required to fit the learning agent into the unique environment of the control agent.

In this current chapter, we apply the learning agent to four example control tasks. The purpose of this chapter is to serve as a case study for the application of the static/dynamic stability theory to real-life control problems. We intend to demonstrate that the theory is easily amenable to practical control application. The purpose of this chapter is *not* to provide an empirical study on the performance of this method, nor is it to compare this learning agent's control performance with other control designs. However, we will show that the control performance of our learning agent is at least comparable to other control methods in order to demonstrate that the statically and dynamically stable learning agent is able to perform well in practice.

The four example tasks we have selected for our case study each afford a different demonstrative purpose. The first two control tasks are relatively simple from a control standpoint; both are dynamic systems of simple enough complexity that the reader can easily visualize the control dynamics. The first task is a simple first-order positioning control system. The second task adds second-order dynamics which are more characteristic of standard "physically realizeable" control problems.

The third case study involves a challenging distillation column control task which is selected from a robust control textbook [Skogestad and Postlethwaite, 1996]. The distillation column task is MIMO (multi-input, multi-output) with highly uncoupled dynamics that make this task a highly difficult control problem. The final case study concerns an HVAC (Heating, Ventilation, and Air-Conditioning) control system. We apply our robust control agent to a model of a heating coil and discuss the suitability of applying our robust control agent to simulation models.

6.1 Case Study: Task 1, A First-Order Positioning System

Task 1 is a simple non-mechanical positioning task. A single input called the reference signal, r moves on the interval $[-1, 1]$ at random points in time. The plant output, y, must track r as closely as possible. The system is depicted in Figure 6.1

The plant is a first order system and thus has one internal state variable x. It is the plant output, y, that must track r. A control signal u is provided by the controller(s) to position y closer to r. A block diagram of the system is shown in Figure 6.2.

Although the original task is posed in the continuous time, we convert it to a discrete-time system for compatibility with the digital learning agent. The dynamics of the discrete-time system are given by:

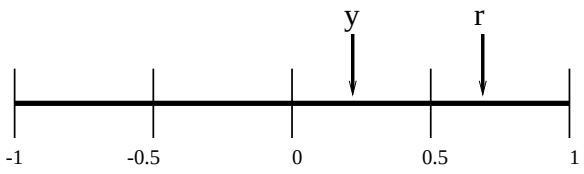


Figure 6.1: Task 1: First Order System

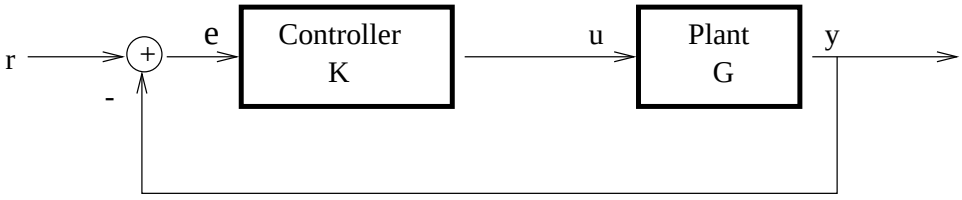


Figure 6.2: Task 1: Nominal Control System

$$x(k+1) = x(k) + u(k) \tag{6.1}$$
$$y(k) = x(k) \tag{6.2}$$

where k is the discrete time step representing 0.01 seconds of elapsed time. We implement a simple proportional controller (the control output is proportional to the size of the current error) with $K_p = 0.1$.

$$e(k) = r(k) - y(k) \tag{6.3}$$
$$u(k) = 0.1e(k) \tag{6.4}$$

Notice that the system is first order with none of the physically interpretable properties such as friction, momentum and spring forces.

6.1.1 Learning Agent Parameters

Recall from Chapter 5 that the learning agent has a dual network architecture. The critic network is responsible for learning the value function (mapping state variable and control action to control performance) and the actor network is responsible for learning the control policy (mapping state variables to control actions).

The critic network is a table look-up with input vector $[e, \hat{u}]$ and the single value function output, $Q(e, \hat{u})$. The table has 25 partitions separating each input forming a 25x25 matrix. The actor network is a two-layer, feed forward neural network. The two inputs are $(e, 1)$ where the 1 is the fixed-input bias term. There are three *tanh* hidden units, and one network output $\hat{u}$. The entire network is then added to the control system. This arrangement is depicted in block diagram form in Figure 6.3.

For training, the reference input r is changed to a new value on the interval $[-1, 1]$ stochastically with an average period of 20 time steps (every half second of simulated time). We train for 2000 time steps at learning rates of $\alpha = 0.5$ and $\beta = 0.1$ for the critic and actor networks respectively. Then we train for an additional 2000 steps with learning rates of $\alpha = 0.1$ and $\beta = 0.01$. Recall from Section 5.5 that α is the learning rate of the critic network and β is the learning rate for the actor network. In Section 5.4 we discussed the reason for the different learning rates: in order to ensure that both networks converge during learning, the critic network must learn faster than the actor network.

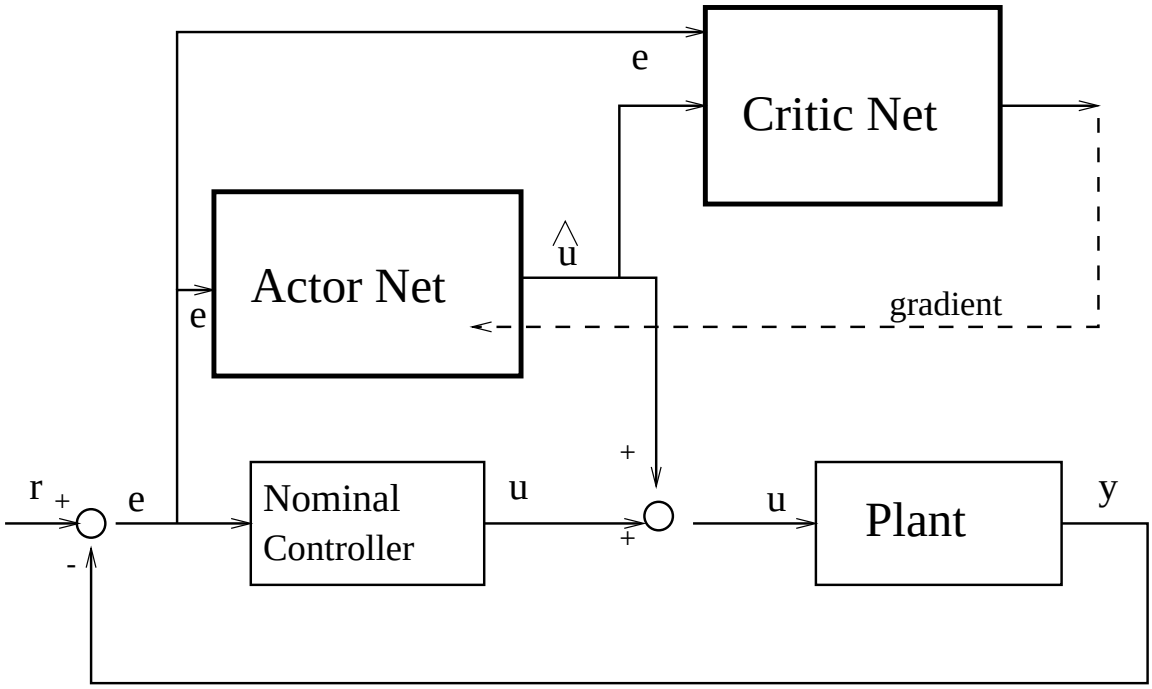


Figure 6.3: Task 1: Control System with Learning Agent

In the above description, we make several decisions regarding design issues such as network sizes, network types, and learning rates. We must be explicit about the motivations behind each design decision. For the actor and critic networks, the number of inputs and outputs are predetermined by the task; Task 1 has one state variable and one control variable that necessitate the input/output sizes of the two networks. We use trial and error to test several different numbers of hidden units for the actor network; three hidden units seem to provide adequate functional expression for learning the actor net's policy. We caution that this trial and error approach is not an exhaustive or thorough empirical investigation into which number of hidden units provides the best results in terms of learning speed and overall control performance. The trial and error testing simply allows us to quickly find network parameters (the number of hidden units in this case) which seem to work well in practice. We apply similar trial and error testing to arrive at the learning rates and the number of required training iterations. Regarding the type of network, we choose a table look-up network for the critic net in order to avoid neuro-dynamic problems; these are discussed in great detail in Chapter 5. The actor net is a two-layer feed forward net with *tanh* hidden unit activation functions as per the requirements also stated in Chapter 5: we need a continuous function to implement the policy in order to satisfy the theoretical requirements of static and dynamic stability analysis.

6.1.2 Static Stability Analysis

In this section, we will assess the stability of neuro-control system using both μ-analysis and IQC-analysis. μ-analysis is an optional software toolbox offered with the Matlab commercial software package. Simulink, also part of the Matlab toolbox collection, allows control engineers to depict control systems with block diagrams. We construct several of these Simulink diagrams in this chapter. Figure 6.4 depicts the Simulink diagram for the nominal control system in Task 1. We refer to this as the nominal system because there is no neuro-controller added to the system. The plant is represented by a rectangular block that implements a discrete-time state space system. The simple proportional controller is implemented by a triangular gain block. Another gain block provides the negative feedback path. The reference input is drawn from the left and the system output exits to

the right.

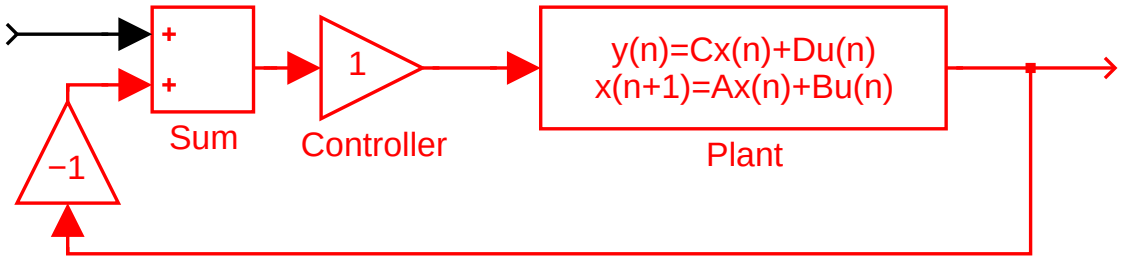


Figure 6.4: Task 1: Nominal System

Next, we add the neural network controller to the diagram. Figure 6.5 shows the complete version of the neuro-controller including the *tanh* function. This diagram is suitable for conducting simulation studies in Matlab. However, this diagram cannot be used for stability analysis, because the neural network, with the nonlinear *tanh* function, is not represented as an LTI system. Constant gain matrices are used to implement the input side weights, W, and output side weights, V. For the static stability analysis in this section, we start with an actor net that is already fully trained. The static stability test will verify whether this particular neuro-controller implements a stable control system. In the next section on dynamic stability, we demonstrate how we ensure stability while the actor net is training.

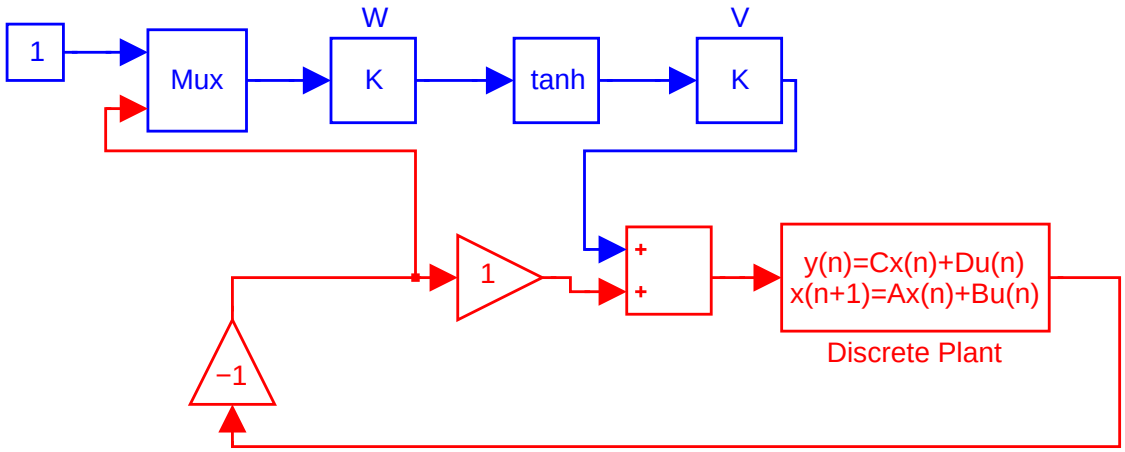


Figure 6.5: Task 1: With Neuro-Controller

Notice that the neural network (in blue) is in parallel with the existing proportional controller; the neuro-controller adds to the proportional controller signal. The other key feature of this diagram is the absence of the critic network; only the actor net is depicted here. Recall that the actor net is a direct part of the control system while the critic net, more of a meta-system, does not directly affect the feedback/control loop of the system. The critic network only influences the direction of learning for the actor network. Since the critic network plays no role in the stability analysis, there is no reason to include the critic network in any Simulink diagrams.

Figure 6.6 shows the the LTI version of the same system. Recall from Section 4.2 that we replace the nonlinear *tanh* function with γ composed of a known fixed part and an unknown uncertainty part. The upper path represents the fixed part (0.5) while the lower path implements the unknown

uncertainty (± 0.5). The uncertainty is represented in Simulink by the circular input-output blocks.

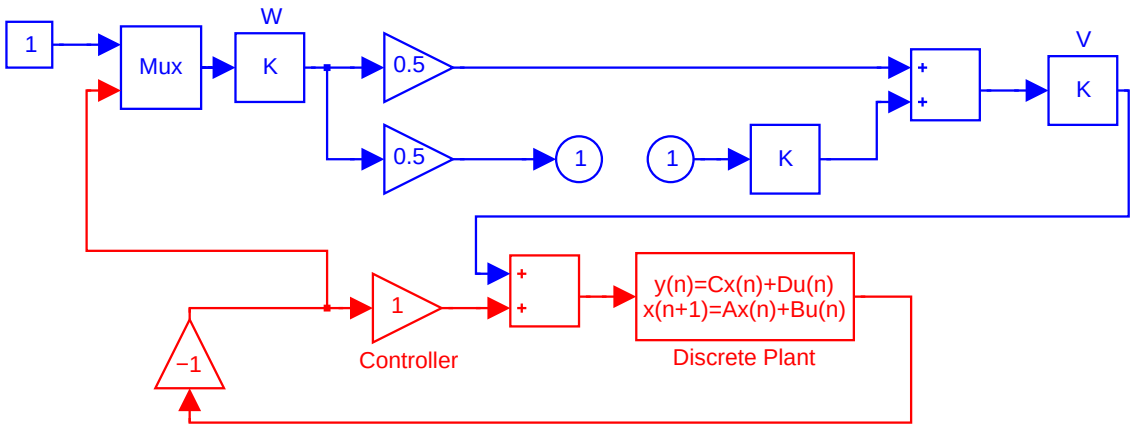


Figure 6.6: Task 1: With Neuro-Controller as LTI

Again, we emphasize that there are *two* versions of the neuro-controller. In the first version, shown in Figure 6.5, the neural network includes all its nonlinearities. This is the actual neural network that will be used as a controller in the system. The second version of the system, shown in Figure 6.6, contains the neural network converted into the LTI framework; we have replaced the nonlinear *tanh* hidden layer with LTI uncertainty. This version of the neural network will never be implemented as a controller; the sole purpose of this version is to ensure stability. Because this version is LTI, we can use the μ-analysis tools to compute the stability margin of the LTI system. Again, because the LTI system *overestimates* the gain of the nonlinearity in the non-LTI system, a stability guarantee on the LTI version also implies a stability guarantee on the non-LTI system.

The next step is to have Matlab automatically formulate the LFT, and then to apply μ-analysis. The Matlab code for this step is shown in Appendix A. The Simulink diagram of Figure 6.6 is given as input to the Matlab μ-analysis commands. Recall from Section 3.8 that $\mu_\Delta(M)$ computes the reciprocal of the smallest perturbation that will cause the system to be unstable. The only uncertainty (perturbation) in the system originates in the neural network hidden layers. Recall we have also normalized the uncertainty to have a maximum norm of 1. After normalization, if $\mu(M)$ is computed to be less than unity at all frequencies, then our system is guaranteed to be stable. In fact, we would like μ to be significantly less than unity, because this indicates that our system is very stable (the closer to 1, the closer to instability) and the smaller μ value gives us extra "room" in which to adjust the network weights. We will be adding additional uncertainty in the dynamic stability section and we would like to have extra "stability room" for adjusting network weights during learning. Figure 6.7 shows the results of the μ computation plotted by frequency. Recall that μ-analysis operates by computing the μ value on a frequency by frequency basis. As seen in Figure 6.7, μ attains a maximum of approximately 0.128 which is significantly less than 1; our system with neuro-controller is very stable.

Recall that IQC-analysis (integral quadratic constraints) is an additional tool for stability analysis [Megretski and Rantzer, 1997a; Megretski and Rantzer, 1997b; Megretski et al., 1999]. IQC is a non-commercial Matlab toolbox which arrives at equivalent stability guarantees using different methods. For IQC-analysis, we make some slight changes to the "LTI-version" of the Simulink diagram. Figure 6.8 depicts the Simulink diagram ready to perform IQC stability analysis on Task 1. The nonlinearity of the neural network is simplified by the single IQC block labeled *odd slope nonlinearity*. IQC provides a number of blocks for different types of uncertainties. The *performance* block is another IQC block that must be included in all IQC Simulink diagrams.

When we run the IQC commands, the automated software executes a feasibility search for a matrix satisfying the IQC function. If the search is feasible, the system is guaranteed stable; if the

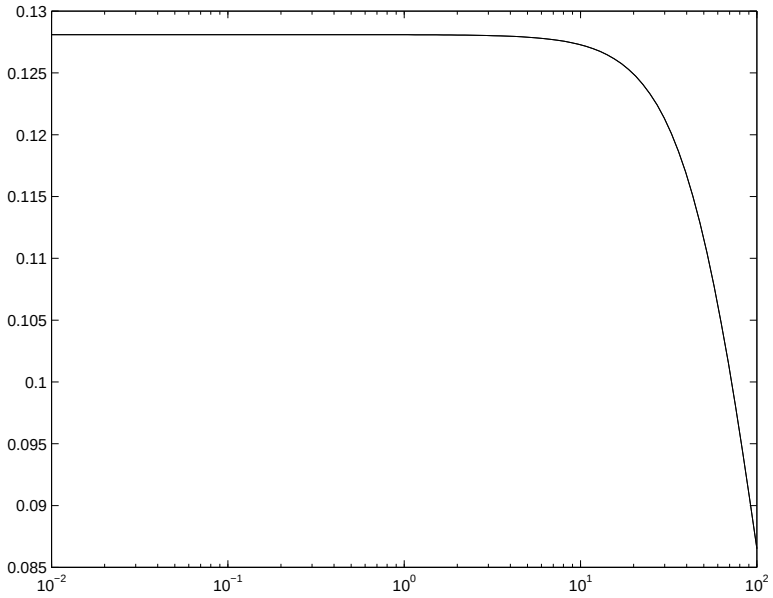


Figure 6.7: Task 1: μ-analysis

search is infeasible, the system is not guaranteed to be stable. IQC does not produce the frequency-by-frequency result of μ-analysis; instead it simply responds with a single feasible/infeasible reply. We apply the IQC commands to the Simulink diagram for Task 1 and find that the feasibility constraints are easily satisfied; the neuro-controller is guaranteed to be stable. This reaffirms our results obtained with μ-analysis. See Appendix A for details on the IQC software commands.

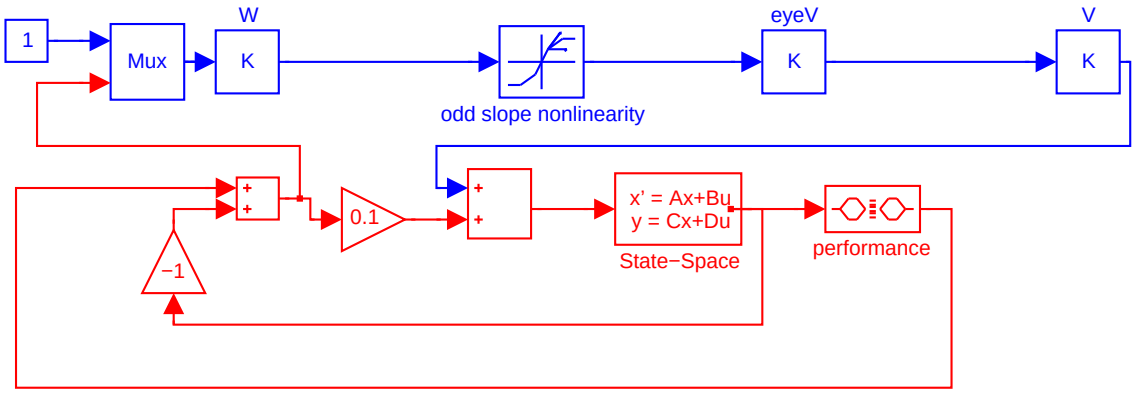


Figure 6.8: Task 1: With Neuro-Controller as LTI (IQC)

At this point, we have assured ourselves that the neuro-controller, after having completely learned its weight values during training, implements a stable control system. Thus we have achieved *static stability*. We have not, however, assured ourselves that the neuro-controller did not temporarily implement an unstable controller while the network weights were being adjusted during learning.

6.1.3 Dynamic Stability Analysis

In this subsection we impose extra limitations on the learning algorithm in order to ensure the network is stable according to dynamic stability analysis. In Chapter 4 we developed a "stable reinforcement learning algorithm"; in Section 5.5 we detail the steps of the algorithm. Recall this algorithm alternates between a stability phase and a learning phase. In the stability phase, we use μ-analysis or IQC-analysis to compute the maximum allowed perturbations for the actor network weights that still provide an overall stable neuro-control system. The learning phase uses these perturbation sizes as room to safely adjust the actor net weights.

To perform the stability phase, we add an additional source of uncertainty to the Simulink diagrams of the previous section. In Figure 6.9 we see the additional uncertainty in the green section. The important matrices are dW and dV. These two matrices are the perturbation matrices. In our previous analysis, we combine all the entries into one matrix called B_P. In this Simulink diagram, we must divide B_P into its two parts: one for the actor net input weights, W, and one for the actor net output weights, V. An increase or decrease in dW implies a corresponding increase or decrease in the uncertainty associated with W. Similarly we can increase or decrease dV to enact uncertainty changes to V.

The matrices WA, WB, VA, and VB are simply there for redimensioning the sizes of W and V; they have no affect on the uncertainty or norm calculations. In the diagram, dW and dV contain all the individual perturbations along the diagonal while W and V are not diagonal matrices. Thus, W_{hxn} and dW_{hnxhn} are not dimensionally compatible. By multiplying with WA and WB we fix this "dimensional incompatibility" without affecting any of the numeric computations. Similarly with V and dV.

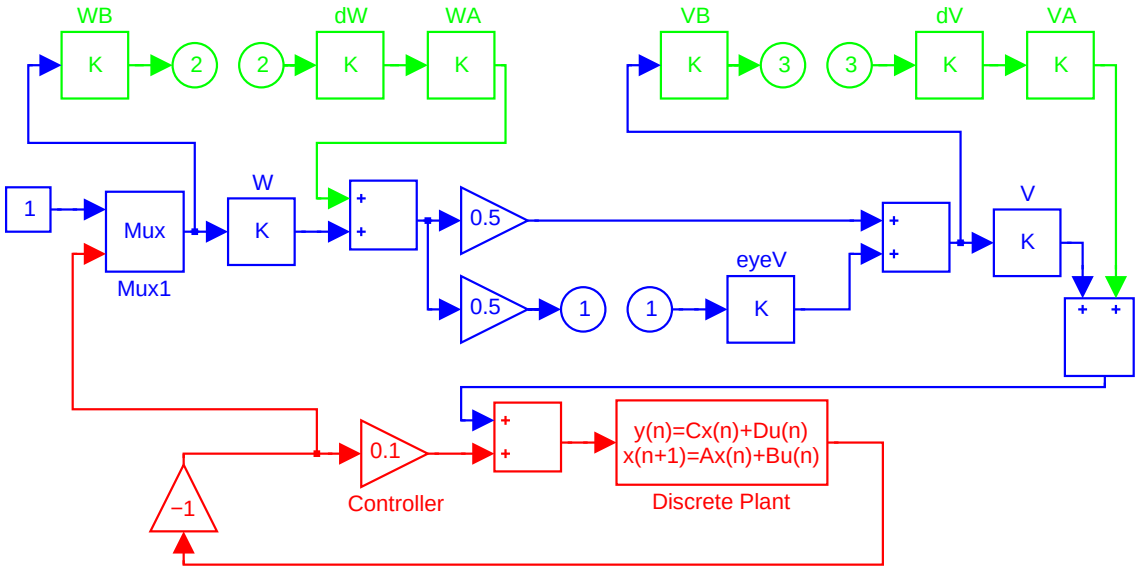


Figure 6.9: Task 1: Simulink Diagram for Dynamic μ-analysis

The stability phase algorithm interacts with the Simulink diagram in Figure 6.9 to find the largest set of uncertainties (the largest perturbations) for which the system is still stable. As a result of this process, we now possess a very critical piece of information. We are now guaranteed that our control system is stable for the current neural network weight values. Furthermore, the system will remain stable if we change the neural network weight values as long as the new weight does not exceed the range R specified by the perturbation matrices, dW and dV (formally called B_P). In other words, we alter network weight values and are certain of a stable control scheme as long as the changes do not exceed R. In the learning phase, we apply the reinforcement learning algorithm until one of the

network weights exceeds the range specified by the additives.

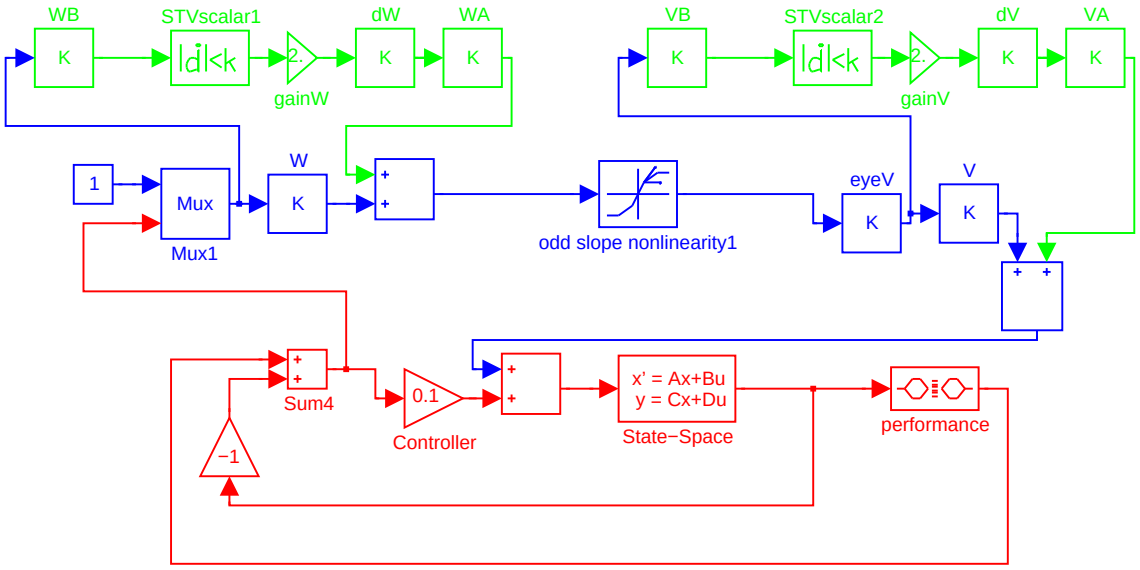


Figure 6.10: Task 1: Simulink Diagram for Dynamic IQC-analysis

We have an additional Simulink diagram for dynamic stability analysis with IQC. We use an STV (Slowly Time-Varying) IQC block to capture the weight change uncertainty. This diagram is shown in Figure 6.10. Other than the different Simulink diagram, the IQC dynamic stability algorithm operates in exactly the same way as μ-analysis. We simply use IQC-analysis to compute the stability of the network for each set of perturbations specified in dW and dV.

6.1.4 Simulation

We fully train the neural network controller as described in Section 6.1.1 and Section 6.1.3. After training is complete, we place the final neural network weight values (W and V) in the constant gain matrices of the Simulink diagram in Figure 6.5. We then simulate the control performance of the system. A time-series plot of the simulated system is shown in Figure 6.11. The top diagram shows the system with only the proportional controller corresponding to the Simulink diagram in Figure 6.4. The bottom diagram shows the same system with both the proportional controller and the neuro-controller as specified in Figure 6.5. The blue line is the reference input r. The green line is the plant output y. The red line is the control signal u.

The system is tested for a 10 second period (1000 discrete time steps with a sampling period of 0.01). We compute the sum of the squared tracking error (SSE) over the 10 second interval. For the proportional only controller, the $SSE = 33.20$. Adding the neuro-controller reduced the SSE to 11.73. Clearly, the reinforcement learning neuro-controller is able to improve the tracking performance dramatically. Note, however, with this simple first-order system it is not difficult to construct a better performing proportional controller. In fact, setting the constant of proportionality to 1 ($K_p = 1$) achieves optimal control (minimal control error). We have purposely chosen a suboptimal controller in this case study so that the neuro-controller has room to learn to improve control performance.

6.2 Detailed Analysis of Task 1

In this section, we provide a more thorough analysis of Task 1. Specifically, we present a detailed analysis of the neuro-dynamics and the trajectory of weight updates. We also provide a discussion

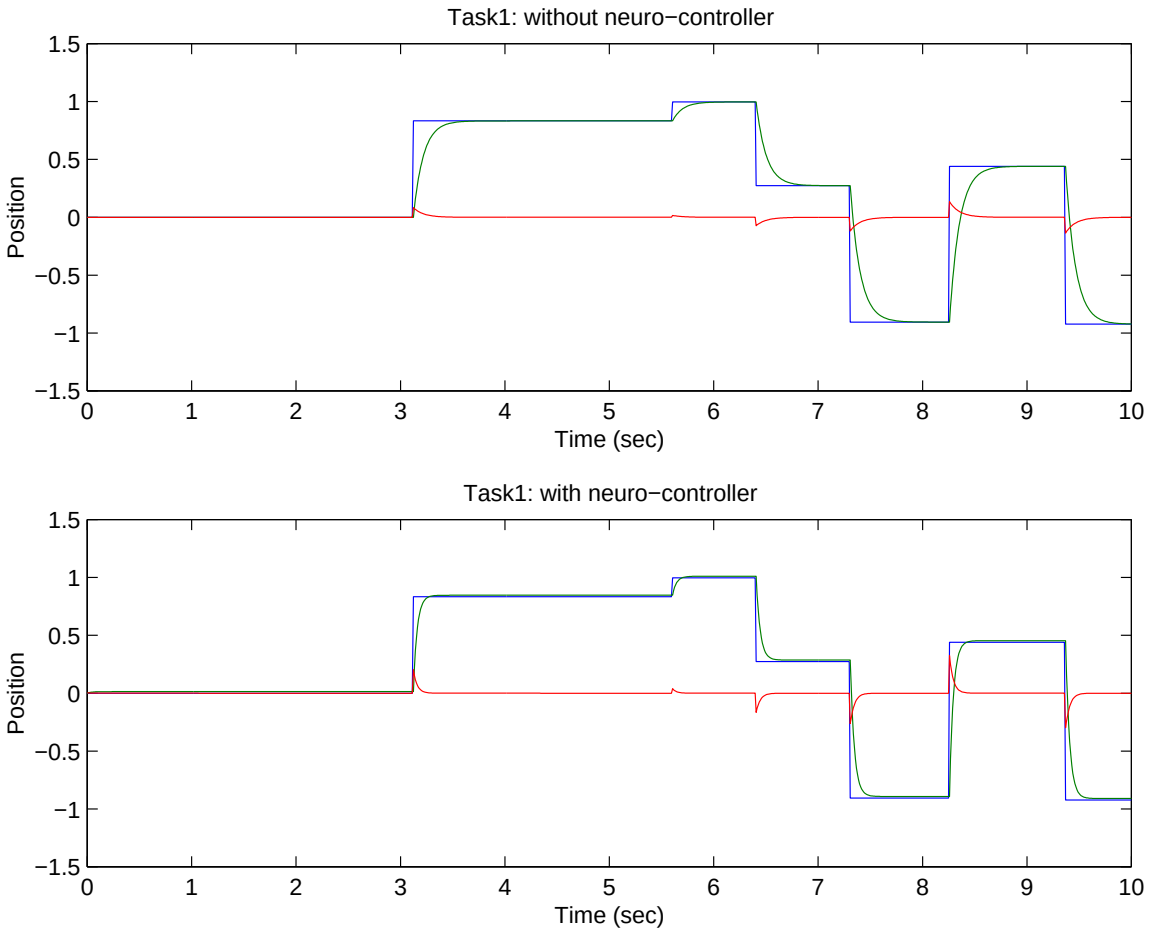


Figure 6.11: Task 1: Simulation Run

of how the "stability" part of the algorithm affects the reinforcement learning part. This analysis is possible because of the simple dynamics involved in Task 1; we cannot perform this analysis with the other three case studies because of the higher order dynamics. This analysis is the primary motivation for including Task 1 in our case study despite the unchallenging control problem this task presents.

6.2.1 Actor/Critic Net Analysis

In order to provide a better understanding of the nature of the actor-critic design, we include the following diagrams. Recall that the purpose of the critic net is to learn the value function (Q-values). The two inputs to the critic net are the system state (which is the current tracking error e) and the actor net's control signal ($\hat{u}$). The critic net forms the Q-values, or value function, for these inputs; the value function is the expected sum of future squared tracking errors. In Figure 6.12 we see the value function learned by the critic net. The tracking error e is on the x-axis while the actor network control action $\hat{u}$ forms the y-axis. For any given point $(e, \hat{u})$ the height (z-axis) of the diagram represents the expected sum of future squared tracking errors.

We can take "slices", or y-z planes, from the diagram by fixing the tracking error on the x-axis. Notice that for a fixed tracking error e, we vary $\hat{u}$ to see a "trough-like" shape in the value function. The low part of the trough indicates the minimum discounted sum squared error for the system. This low point corresponds to the control action that the actor net should ideally implement. We

use the trough gradient to do back propagation for the actor net. The surface gradient in the critic net is used to provide training exemplars for the actor net.

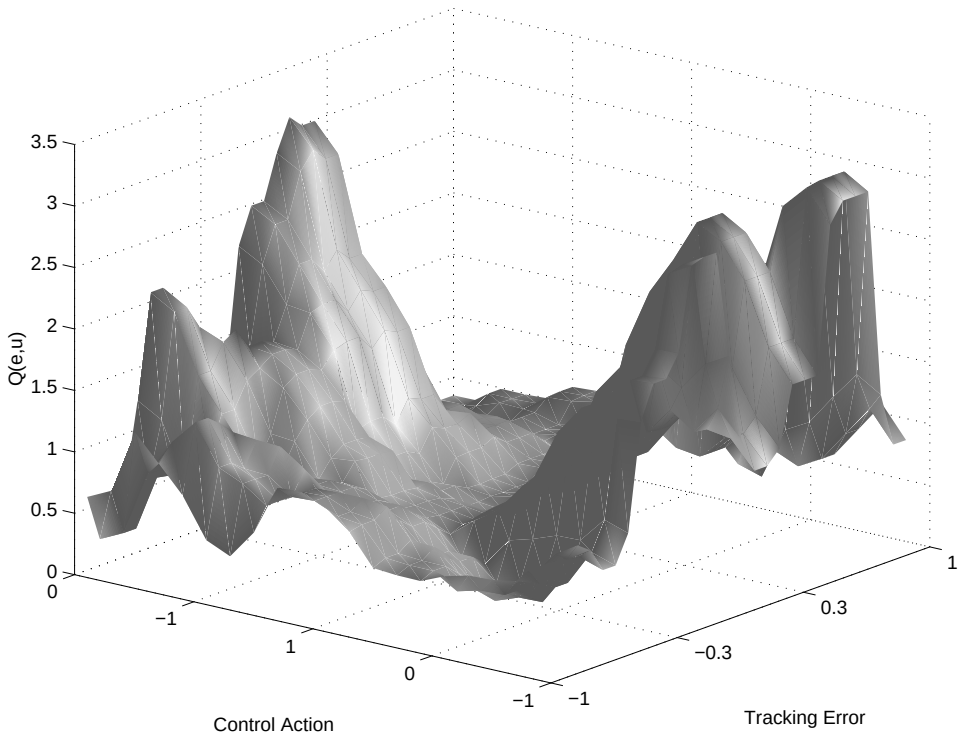


Figure 6.12: Task 1: Critic Net's Value Function

It is important to keep in mind that the critic network is an approximation to the true value function. The critic network improves its approximation through learning by sampling different pairs $(e, \hat{u})$ and computing the resulting sum of future tracking errors. This "approximation" accounts for the bumpy surface in Figure 6.12. After more training, the surface smoothes as it becomes closer to the true value function. This is also why it is important to have a faster learning rate for the critic network than for the actor network. Because the value function learned by the critic network directs the updates of the actor network, we must be able to learn the gradient of the critic network faster than the actor network changes its weight values.

The actor net's purpose is to implement the current policy. Given the input of the system state (e), the actor net produces a continuous-valued action $(\hat{u})$ as output. In Figure 6.13 we see the function learned by the actor net. For negative tracking errors $(e < 0)$ the system has learned to output a strongly negative control signal. For positive tracking errors, the network produces a positive control signal. The effects of this control signal can be seen qualitatively by examining the output of the system in Figure 6.11.

Note, these diagrams are only possible because of the extraordinary simplicity of this control task. Because this case study has first order dynamics, the system has only one internal state variable, the tracking error e. With only one internal state, the output of the actor network can be viewed in a two-dimensional plot while the output of the critic network is captured with a three-dimensional surface plot. This visualization is not possible with the higher-order tasks in the next three case studies.

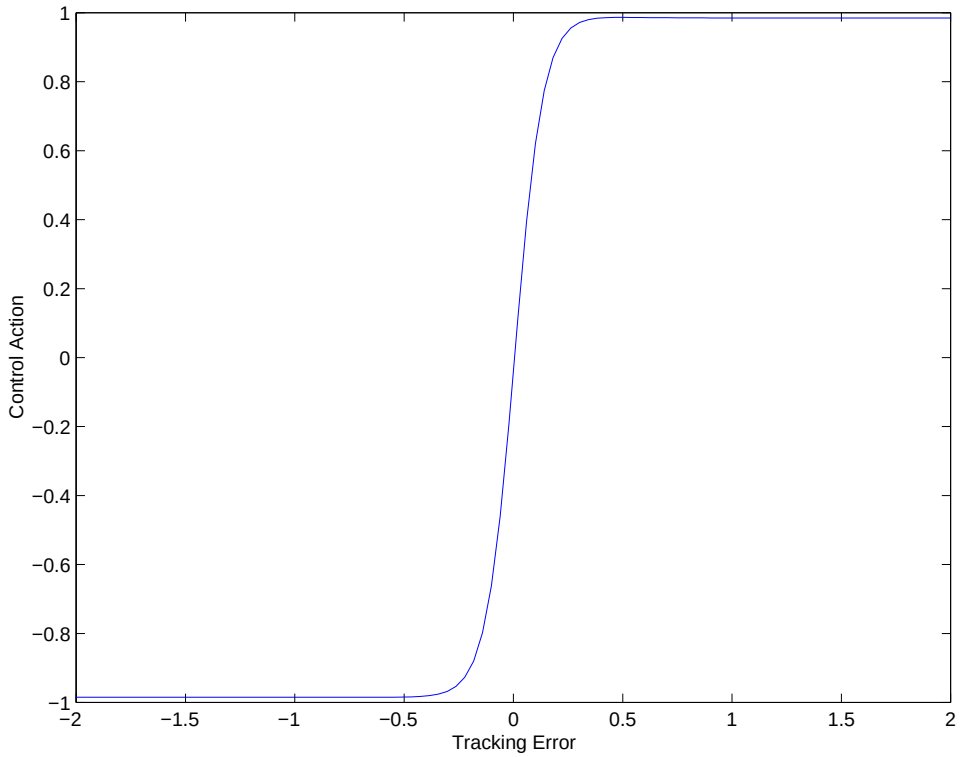


Figure 6.13: Task 1: Actor Net's Control Function

6.2.2 Neural Network Weight Trajectories

From our discussion in previous sections, we know that the learning algorithm is a repetition of stability phases and learning phases. In the stability phases we compute the maximum additives, dW and dV, which still retain system stability. In the learning phases, we adjust the neural network weights until one of the weights exceeds its range specified by its corresponding additive. In this section, we present a visual depiction of the learning phase for an agent solving Task 1.

In order to present the information in a two-dimensional plot, we switch to a minimal actor net. Instead of the three *tanh* hidden units specified earlier in this chapter, we use one hidden unit *for this subsection only*. Thus, the actor network has two inputs (the bias = 1 and the tracking error e), one *tanh* hidden unit, and one output ($\hat{u}$). This network will still be able to learn a relatively good control function. Refer back to Figure 6.13 to convince yourself that only one hidden *tanh* unit is necessary to learn this control function; we simply found, in practice, that three hidden units often resulted in faster learning and slightly better control.

For this *reduced* actor net, we now have smaller weight matrices for the input weights W and the output weights V in the actor net. W is a 2x1 matrix and V is a 1x1 matrix, or scalar. Let W_1 refer to the first component of W, W_2 refer to the second component, and V simply refers to the lone element of the output matrix. The weight, W_1, is the weight associated with the bias input (let the bias be the first input to the network and let the system tracking error, e, be the second input). From a stability standpoint, W_1 is insignificant. Because the bias input is clamped at a constant value of 1, there really is no "magnification" from the input signal to the output. The W_1 weight is not on the input/output signal pathway and thus there is no contribution of W_1 to system stability. Essentially, we do not care how weight W_1 changes as it does not affect stability. However, both W_2 (associated with the input e) and V do affect the stability of the neuro-control system as these weights occupy the input/output signal pathway and thus affect the closed-loop energy gain of the

system.

To visualize the neuro-dynamics of the actor net, we track the trajectories of the individual weights in the actor network as they change during learning. The weights W_2 and V form a two-dimensional picture of how the network changes during the learning process. Figure 6.14 depicts the two-dimensional weight space and the trajectory of these two weights during a typical training episode. The x-axis shows the second input weight W_2 while the y-axis represents the single output weight V. The trajectory begins with the blue colorings, progresses to red, green, magenta, and terminates with the yellow coloring. Each point along the trajectory represents a weight pair (W_2,V) achieved at some point during the learning process.

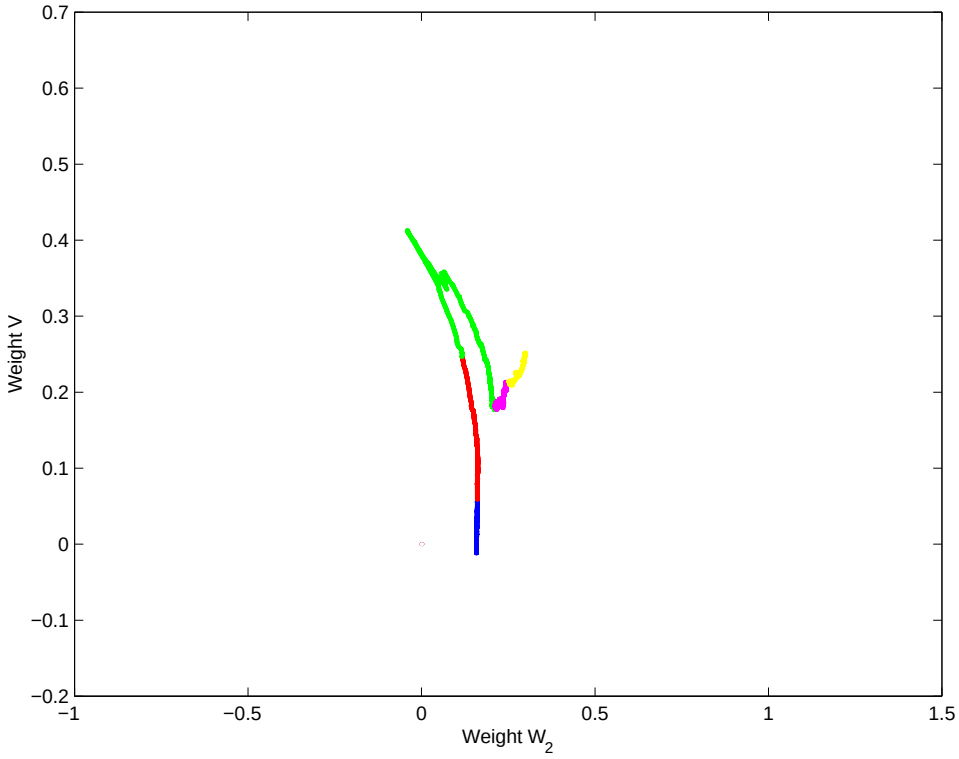


Figure 6.14: Task 1: Weight Update Trajectory

The colors represent different phases of the learning algorithm. First, we start with a stability phase by computing, via μ-analysis or IQC-analysis, the amount of uncertainty which can be added to the weights; the resulting perturbations, dW and dV, indicate how much learning we can perform and still remain stable. The blue part of the trajectory represents the learning that occurred for the first values of dW and dV. The blue portion of the trajectory corresponds to the first learning phase. After the first learning phase, we then perform another stability phase to compute new values for dW and dV. We then enter a second learning phase that proceeds until we attempt a weight update exceeding the allowed range. This second learning phase is the red trajectory. This process of alternating stability and learning phases repeats until we are satisfied that the neural network is fully trained (more comments about this in the next sections). In the diagram of Figure 6.14 we see a total of five learning phases (blue, red, green, magenta, and yellow).

6.2.3 Bounding Boxes

Recall that the terms dW and dV indicate the maximum uncertainty, or perturbation, we can introduce to the neural network weights and still be assured of stability. If W_2 is the current weight

associated with the input e, we can increase or decrease this weight by dW and still have a stable system. $W_2 + dW$ and $W_2 - dW$ form the range, R_{W_2}, of "stable values" for the input actor weight W_2. These are the values of W_2 for which the overall control system is guaranteed to be stable. Similarly $V \pm dV$ form the stable range of output weight values. We depict these ranges as rectangular boxes in our two-dimensional trajectory plot. These boxes are shown in Figure 6.15.

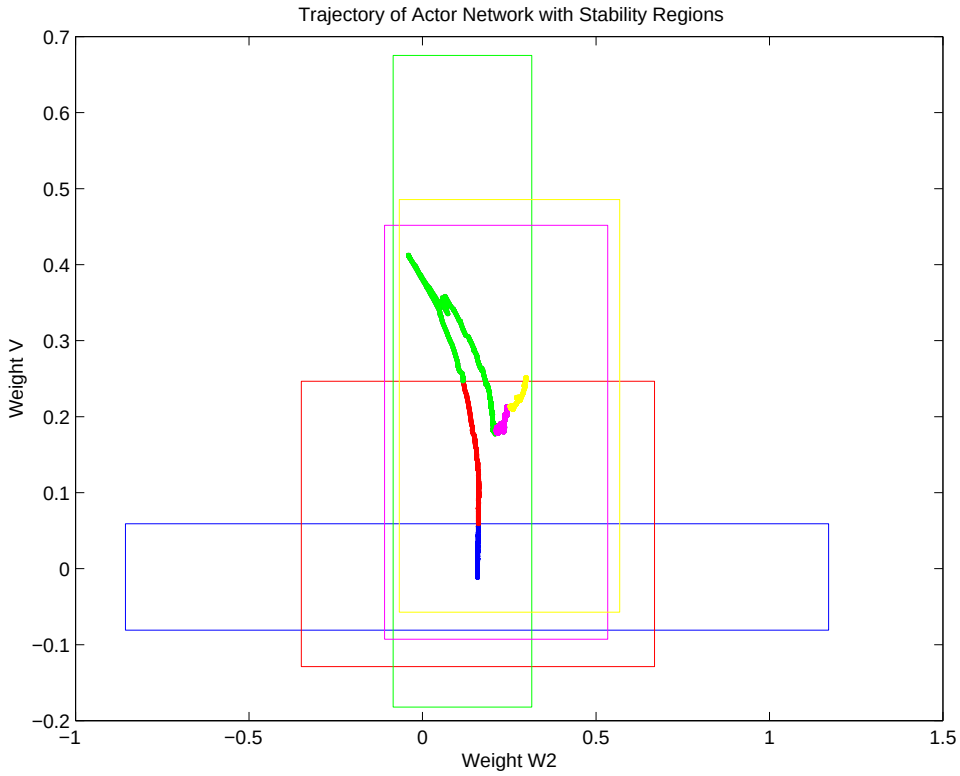


Figure 6.15: Task 1: Trajectory with Bounding Boxes

Again, there are five different bounding boxes (blue, red, green, magenta, and yellow) corresponding to the five different stability/learning phases. As can be seen from the blue trajectory in this diagram, training progresses until the V weight reaches the edge of the blue bounding box. At this point we must cease our current reinforcement learning phase, because any additional weight changes will result in an unstable control system (technically, the system might still be stable but we are no longer *guaranteed* of the system's stability – the stability test is conservative in this respect). At this point, we recompute a new bounding box (red) using a second stability phase; then we proceed with the second learning phase until the weights violate the new bounding box. In this way the stable reinforcement learning algorithm alternates between stability phases (computing bounding boxes) and learning phases (adjusting weights within the bounding boxes).

It is important to note that if the trajectory reaches the edge of a bounding box, we may still be able to continue to adjust the weight in that direction. Hitting a bounding box wall does not imply that we can no longer adjust the neural network weight(s) in that direction. Recall that the edges of the bounding box are computed *with respect to the network weight values at the time of the stability phase*; these initial weight values are the point along the trajectory in the exact center of the bounding box. This central point in the weight space is the value of the neural network weights at the beginning of this particular stability/learning phase. This central weight space point is the value of W_2 and V that are used to compute dW and dV. Given that our current network weight values are that central point, the bounding box is the *limit* of weight changes that the network tolerates without forfeiting the stability guarantee. This is not to be confused with an *absolute* limit on the

size of that network weight. We will return to address this point further in the next subsections.

The green trajectory reveals some insightful dynamics. The green portion of the trajectory stops near the edge of the box (doesn't reach it), and then moves back toward the middle. Keep in mind that this trajectory represents the weight changes in the actor neural network. At the same time as the actor network is learning, the critic network is also learning and adjusting its weights; the critic network is busy forming the value function. It is during this green phase in the training that the critic network has started to mature; the "trough" in the critic network has started to form. Because the gradient of the critic network directs the weight changes for the actor network, the direction of weight changes in the actor network reverses. In the early part of the learning (red and blue trajectories) the critic network gradient indicates that "upper left" is a desirable trajectory for weight changes in the actor network. By the time we encounter our third learning phases in the green trajectory, the gradient in the critic network has changed to indicate that "upper-left" is now an undesirable direction for movement for the actor network. The actor network has "over-shot" its mark. If the actor network has higher learning rates than the critic network, then the actor network would have continued in that same "upper-left" trajectory, because the critic network would not have been able to learn quickly enough to direct the actor net back in the other direction.

Further dynamics are revealed in the last two phases. As can be seen from the magenta and yellow trajectories, the actor network weights are not changing as rapidly as they did in the earlier learning phases. We are reaching the point of optimal tracking performance according to the gradient in the critic network. The point of convergence of the actor network weights is a local optima in the value function of the critic network weights. We halt training at this point because the actor weights have ceased to move and the resulting control function improves performance (minimizes tracking error) over the nominal system.

This two dimensional plot of the trajectory enables us to demonstrate some of the critical dynamics of the stable reinforcement learning algorithm. The plot shows how the weights adjust during a typical reinforcement learning session. More importantly, by super-imposing the bounding boxes, the relationship between a "pure" reinforcement learning algorithm and the dynamic stability proof is demonstrated. We show that the bounding boxes represent the currently known "frontier" of safe neural network weight values – those weights which implement stable control in the actor network. We also use this diagram to show how the critic network affects the learning trajectory of the actor network weights. The discussion to this point provides a reasonable overview of the neuro-dynamic details of the stable reinforcement learning algorithm. However, there are some subtle implementation issues which are critical to the operation of the algorithm. In the remainder of this subsection we address a number of these more subtle issues. Namely, we discuss the details of computing dW and dV, how to decide when to stop training, and how the trajectories and bounding boxes might differ for other control problems.

6.2.4 Computing Bounding Boxes

During the stability phases, we compute the maximum perturbations, dW and dV, which can be added to the actor neural network's input and output weights while still guaranteeing stability for the overall control system. Each individual change to an entry in dW (or in dV) will affect the stability calculations of μ-analysis and IQC-analysis. Given a current set of neural network weights (W, V), we use μ-analysis to find out how much *total* uncertainty the system can handle. We can then distribute this total uncertainty among the various elements of dW and and dV [1]. As a consequence of these multiple degrees of freedom, we now must find a method for selecting the individual entries of the additive matrices dW and dV.

It is critical to note that *how* we select entries for dW and dV has absolutely no impact on the ultimate weights that the actor neural network learns; this point is spelled out explicitly in

[1] In reality, μ-analysis does not work this way. We cannot compute a total amount of uncertainty and then go back to redistribute it among dW and dV. Because μ is largely a boolean test (T = stable, F = not stable), we can only preselect dW and dV and then use μ to test their stability status. A possible direction of future research would be to develop a μ tool that works in this way.

Section 6.2.5 below. Instead, our selection of the perturbation matrices will impact only the efficiency of our algorithm; the impact arises in the number of computationally expensive stability phases that must be executed. By judicious choice of how we distribute uncertainty among dW and dV, we can make the learning algorithm faster computationally, but we cannot change the weight update trajectory formed during learning.

Refer back to the bounding boxes of Figure 6.15. In this figure, we have two weights of interest, W_2 and V. The matrix dW has two entries, dW_1 and dW_2 corresponding to the two entries in W. Similarly, dV has one entry corresponding to the one entry of V. We have already indicated that W_1, which is tied to the bias input term, has no effect on the stability calculations. Thus dW_1 is ignored or set to 0. We now possess two degrees of freedom in selecting uncertainty for neural network weights: dW_2 and dV. These two additive matrix entries are two "dials" that we can turn to adjust the amount of uncertainty associated with their respective neural network weights. If we compute μ for the Simulink diagram and the result is less than unity (indicating stability), then we are permitted to increase one or both of these dials. Conversely, if μ indicates instability, we must decrease the dials – decrease the amount of weight uncertainty. We can also increase one dial, say dW_2, while simultaneously decreasing the other dial, dV, in order to reach the same μ-analysis result. In general, μ, which produces our stability result, is not an explicit function available for introspection; that is, we cannot figure out exactly how much to turn a particular dial in order to reach a desired stability result. Instead, we must simply set the dials (set the levels for dW and dV) and then recompute μ to ascertain the result. It is very much a set-and-test procedure.

Looking back at the trajectory of Figure 6.15, we see how these "dials" come in to play. Recall that dW_2 is the amount of uncertainty associated with the W_2 weight on the x-axis; $W_2 \pm dW_2$ forms the left and right edges of the bounding boxes. Similarly, $V \pm dV_2$ form the upper and lower edges of the bounding boxes. The relative size of dW_2 vs dV determines the "shape" of our bounding boxes. Large dW_2/dV produce wide and short boxes while small dW_2/dV result in tall, narrow boxes. We can turn the dW_2 and dV knobs to reach any desired rectangular shape that we wish. Roughly, though not exactly, the *area* of the rectangle remains somewhat constant due to the μ stability computations; the larger the area of the rectangle, the closer to instability. The purpose of the stability phase is to find the largest area rectangle which is still just barely stable.

One obvious question now remains: how do we set dW_2 in relation to dV? Do we want tall, skinny boxes or short, wide boxes? The answer depends on the future trajectory of the network weights. If the actor net training results in weight changes only to V and not W_2, then we would want tall skinny boxes to maximize the amount of uncertainty associated with V and minimize the amount of uncertainty associated with W_2. However, we would not know this a priori and thus cannot predict how future learning might progress [2].

For the work in this dissertation, we select the following method of relating dW_2 to dV. We set the ratio of dW_2/dV to the ratio of W_2/V and we keep dW_2 and dV in this ratio throughout a stability/learning phase. Once the ratio has been set, we can then increase or decrease the total amount of uncertainty by multiplying dW_2 and dV by the same constant. The larger the constant, the more total weight uncertainty, and the closer to system instability. The stability phase of each learning episode is a mini-search to find the largest such constant that just barely retains system stability guarantees. Keeping dW_2/dV equal to W_2/V has an intuitive appeal, because it fixes the relative amount of learning available to each weight equal to the relative current size of each weight (small amounts of learning allowed for small weights, large amounts of learning allowed for larger weights). We do not claim that this is the optimal way to assign dW_2 and dV; other possibilities are mentioned in the concluding chapter. This method has the other advantage that it is easily scalable to the more complicated tasks with more states and larger neural network weights.

[2] Actually, there are several ways in which we might try to predict future learning. Two of them involve extrapolating from the current learning trajectory, and looking at the current gradient in the critic network. These and other options are discussed in Chapter 7 on future research directions.

6.2.5 Effects on Reinforcement Learning

A drawback of the bounding-box imagery is that one often ascribes more restrictive powers to the bounding box than actually exist. It is critical to note that *the bounding box does not necessarily form an absolute limit on the size of each network weight.* We must distinguish the *local limits* imposed on neural network weights by each bounding box from the *absolute limits* on neural network weights imposed by stability considerations.

For example, suppose we start with an initial point in the weight space (an initial set of network weights), compute the bounding box using μ-analysis, and then plot the bounding box on our two-dimensional trajectory diagram. This bounding box forms a *local limit* on how much weight changes we can tolerate before going unstable. In truth, we may be able to tolerate much larger weight changes, but *from our current perspective* at the initial weight point, we can only ensure stability up to the edge of the bounding box. We then execute one reinforcement learning step to make one small incremental adjustment to the network weights. We thus have a second point in the weight space. These two points, the initial weights and the new weights, form the first two points in our network weight trajectory. After this first learning step is complete, we could recompute a new bounding box by executing a new stability phase (more μ calculations). This second round of stability calculations will result in different values for dW and dV and thus form an entirely new bounding box. It is probable that this new bounding box encloses "safe" areas of the two-dimensional weight space that were not enclosed in the first bounding box. Thus, we have added to the overall region of network weight values that implement stable controllers. This overall safe region is the *absolute limit* on neural network weight values. In practice, we do not execute a stability phase calculation at each learning step, because stability phases are computationally expensive operations. Instead, we continue to learn until we reach the currently known limits of the safe weight range, that is, until we hit the edge of our bounding box. Then we are *forced* to return to the stability phase if we desire to make any further weight updates.

Figure 6.16 illustrates a number of critical points regarding the interaction of the stability phases and the reinforcement learning phases. In the center of Figure 6.16 is a small bounding box drawn in black. There is an initial trajectory (also in black) which starts at the center of the bounding box and moves toward the lower right corner. The bounding box is computed during the stability phase. The initial weight point (W_2, V) in the center of the bounding box is used for the μ-analysis computation. The result of the stability phase is the pair (dW_2, dV) which form the side and top/bottom portions of the bounding box, respectively. The result of the μ computation is the largest amount of uncertainty that the network can tolerate from our current perspective at the initial weight point (W_2, V). We may be able to tolerate more uncertainty, but we cannot ascertain this by μ-analysis performed on the current weight point (W_2, V). In fact, if we select other initial weight points we will generate additional bounding boxes. It is the overall union of all bounding boxes that truly indicates the entire scope of the "safe" network weight values: those network weight points which implement stable control.

In Figure 6.16 we have artificially drawn this global safety range as a red circle. For real control tasks, it is unlikely that this region is the shape of a circle; we have merely drawn it as a circle for simplicity. This region is real in the sense that inside the circle are network weights which implement stable control and outside the circle are network weights which do not implement stable control. We could perform a number of *static stability tests* for (W_2, V) points; those points inside the circle would pass the static stability test while those weight points outside the circle would fail the static stability test.

Although the global stable region exists, it is entirely possible that the network weights may never reach the edge of this red circle during learning; all the bounding boxes might fall well within the interior of the circle. In fact, the trajectory in Figure 6.15 is such an example where all our bounding boxes fall well within the interior of our stability region. We discuss the situation in which learning does cause the trajectory to approach the edge of the global stability region at the end of this subsection.

There is a second region in Figure 6.16 which is also of interest. Show in the blue rectangle is the *performance improving* region. Again, the true performance improving region is not likely to be

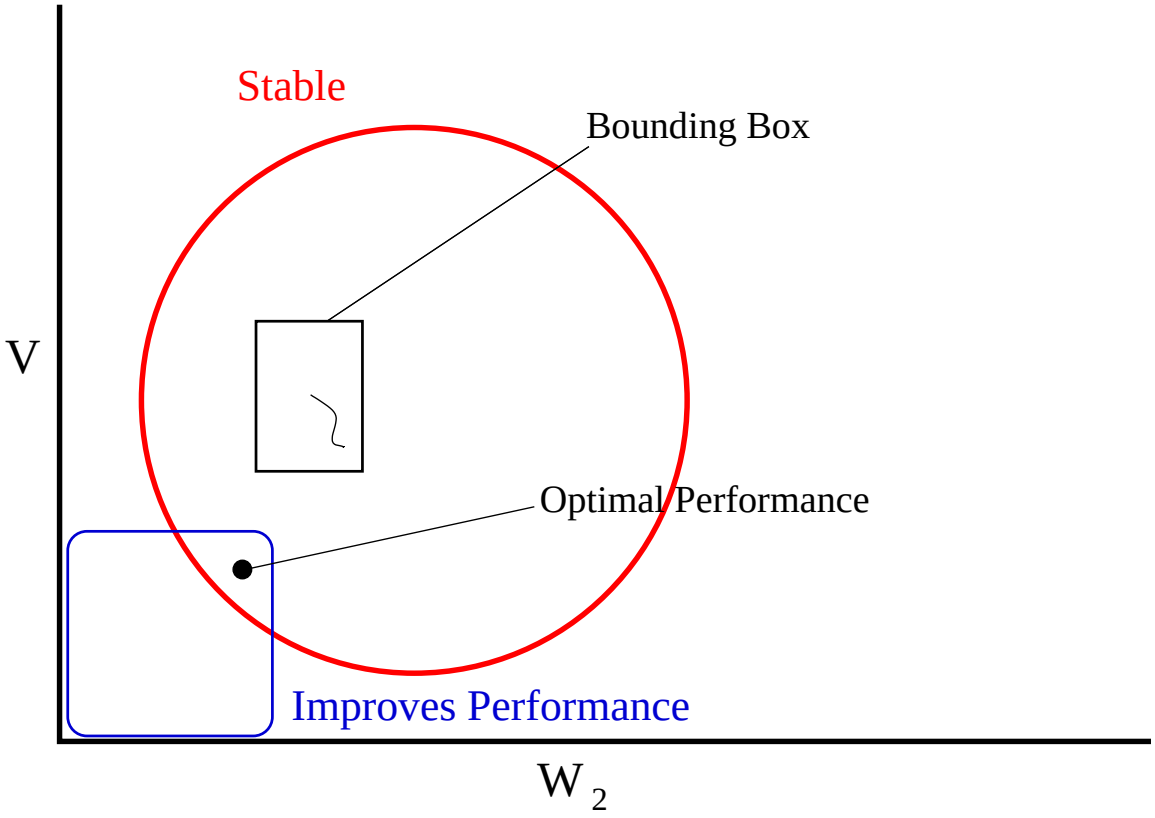


Figure 6.16: Actor Network Weight Space: Stability and Performance Improving Regions

rectangular in shape. In this region are the set of all neural network weights which provide improved control (smaller total tracking error) over the nominal system without the neuro-controller. The interior points of this region form improved neuro-controllers, the exterior points implement worse performing neuro-controllers, and the border points implement the same level of control performance as the nominal system [3]. There exists a point (or points) within this region that provides the best tracking performance possible. This point is the set of actor network weights with *optimal performance*.

There are numerous ways in which these two regions can interact. The size, shape, and overlap of the regions is determined by the control task and by the neural network architecture. One such possible arrangement is for the performance-improving region to be a subset of the stable region; all network weights which improve the tracking performance are also stable. The other possibilities are presented in the following list in which SR is the stable region, PR is the performance-improving region, and OP is the optimal performance point:

1. $SR \subset PR$
2. $PR \subset SR$
3. $SR \cap PR = \emptyset$
4. $SR \cap PR \neq \emptyset, OP \in SR$
5. $SR \cap PR \neq \emptyset, OP \notin SR$

[3] As a technical point, the origin is the point where all the network weights are zero which is, essentially, the nominal system. Thus the origin is always on the border of the performance-improving region.

6. $SR = \emptyset$

7. $PR = \emptyset$

For the third case, we have the unfortunate situation where the performance-improving region and stable region are disjoint, there will be no neural network weights which are both stable and provide improved control [4]. Cases (4) and (5) above are illustrated in Figure 6.16 where the two regions overlap, but are not contained within each other. Here we must distinguish the case where the point of optimal performance is or is not within the stable region. There are also the special cases where the stable set is empty (5); this occurs if there are no possible neural network weight values which will implement a stable controller. The other possibility (6) is that there are no neuro-controllers which will improve the tracking performance over the nominal system; this would have been the case if we had used an optimal controller ($K_p = 1$) for example Task 1.

Finally we come to the main focus of this subsection: bounding boxes do not affect the trajectory of weights encountered during learning. Essentially, the reinforcement learning part of the algorithm is oblivious to the existence of the bounding boxes. The network will sequence through the same set of weight values during learning whether there are bounding boxes or not. The only exception to this rule is when the bounding boxes happen to abut the global stability region (red circle). Only then does the "stability" part of the algorithm affect the "reinforcement learning" part of the algorithm. We discuss this special case immediately below. Consequently, how we chose bounding boxes will not affect what the network is able to learn, because we are not affecting the weight trajectory formed during learning. The choice of how we make bounding boxes only affects how often we must re-compute new bounding boxes.

What happens as learning progresses and takes the weight trajectory close to the edge of the global stability region? The first thing that happens is the control system edges closer and closer to an unstable operating point; there will be less room for uncertainty because more uncertainty will push us over the edge of guaranteed stability. As a result, the perturbations (dW_2, dV) computed during the stability phase will be smaller and smaller. Thus the *area* of the bounding box decreases. In the limit as the weight trajectory approaches the edge of the global stability region, the perturbations and the area of the bounding box go to zero. It is at this point that no further learning can occur. This is the only instance in which the "stable reinforcement learning algorithm" differs from a regular reinforcement learning algorithm. In fact, this is the ideal situation, we would like learning to proceed without interference until learning attempts to push the neuro-controller into a point where the overall system is not guaranteed to be stable.

6.3 Case Study: Task 2, A Second-Order System

The second task, a second order mass/spring/dampener system, provides a more challenging and more realistic system in which to test our neuro-control techniques. Once again, a single reference input r moves stochastically on the interval $[-1, 1]$; the single output of the control system y must track r as closely as possible. However, there are now friction, inertial, and spring forces acting on the system to make the task more difficult than Task 1. Figure 6.17 depicts the different components of the system.

We use the same generic block diagram for Task 2 except that we must keep in mind that the plant now has two internal states (position and velocity) and the controller also now has an internal state.

[4] Technically they cannot be completely disjoint due to the trivial case of the origin which is always "in" both sets.

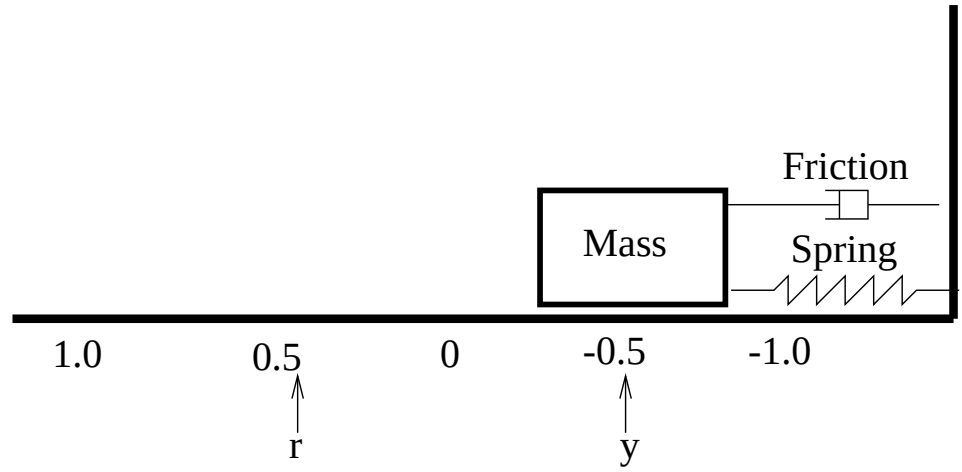


Figure 6.17: Task 2: Mass, Spring, Dampening System

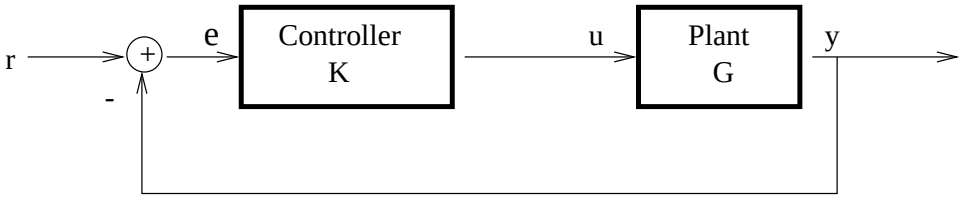


Figure 6.18: Task 2: Nominal Control System

The discrete-time update equations are given by:

$$e(k) = r(k) - y(k) \tag{6.5}$$

$$u(k) = K_p e(k) + \int K_i e(k) \tag{6.6}$$

$$K_p = 0.01 \quad K_i = 0.001 \tag{6.7}$$

$$x(k+1) = \begin{bmatrix} 1 & 0.05 \\ -0.05 & 0.9 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 1.0 \end{bmatrix} u(k) \tag{6.8}$$

$$y(k) = \begin{bmatrix} 1 & 0 \end{bmatrix} x(k) \tag{6.9}$$

Here, the nominal controller is a PI controller with both a *proportional* term and an *integral* term. This controller is implemented with its own internal state variable. The more advanced controller is required in order to provide reasonable nominal control for a system with second-order dynamics as is the case with Task 2. The constant of proportionality, K_p, is 0.01, and the integral constant, K_i, is 0.001. Once again, we have purposely chosen a controller with suboptimal performance so that the neural network has significant margin for improvement.

6.3.1 Learning Agent Parameters

The neural architecture for the learning agent for Task 2 is mostly identical to that used in Task 1. Here we have an actor network with two inputs (the bias term and the current tracking error) and one output (the appended control signal). We retain the three hidden units because, in practice, three *tanh* hidden units seemed to provide the fastest learning and best control performance.

Again, for training, the reference input r is changed to a new value on the interval $[-1, 1]$ stochastically with an average period of 20 time steps (every half second of simulated time). Due

to the more difficult second-order dynamics, we increase the training time to 10,000 time steps at learning rates of $\alpha = 0.5$ and $\beta = 0.1$ for the critic and actor networks respectively. Then we train for an additional 10,000 steps with learning rates of $\alpha = 0.1$ and $\beta = 0.01$.

6.3.2 Simulation

In Figure 6.19, we see the simulation run for the second order task. The top portion of the diagram depicts the nominal control system (with only the PI controller) while the bottom half shows the same system with both the PI controller and the neuro-controller acting together. The blue line is the reference input r and the green line is the position of the system (there is a second state variable, velocity, which is not depicted). Importantly, the K_i and K_p parameters are *suboptimal* so that the neural network has opportunity to improve the control system. As is clearly shown in Figure 6.19, the addition of the neuro-controller clearly does improve system tracking performance. The total squared tracking error for the nominal system is $SSE = 246.6$ while the total squared tracking error for the neuro-controller is $SSE = 76.3$.

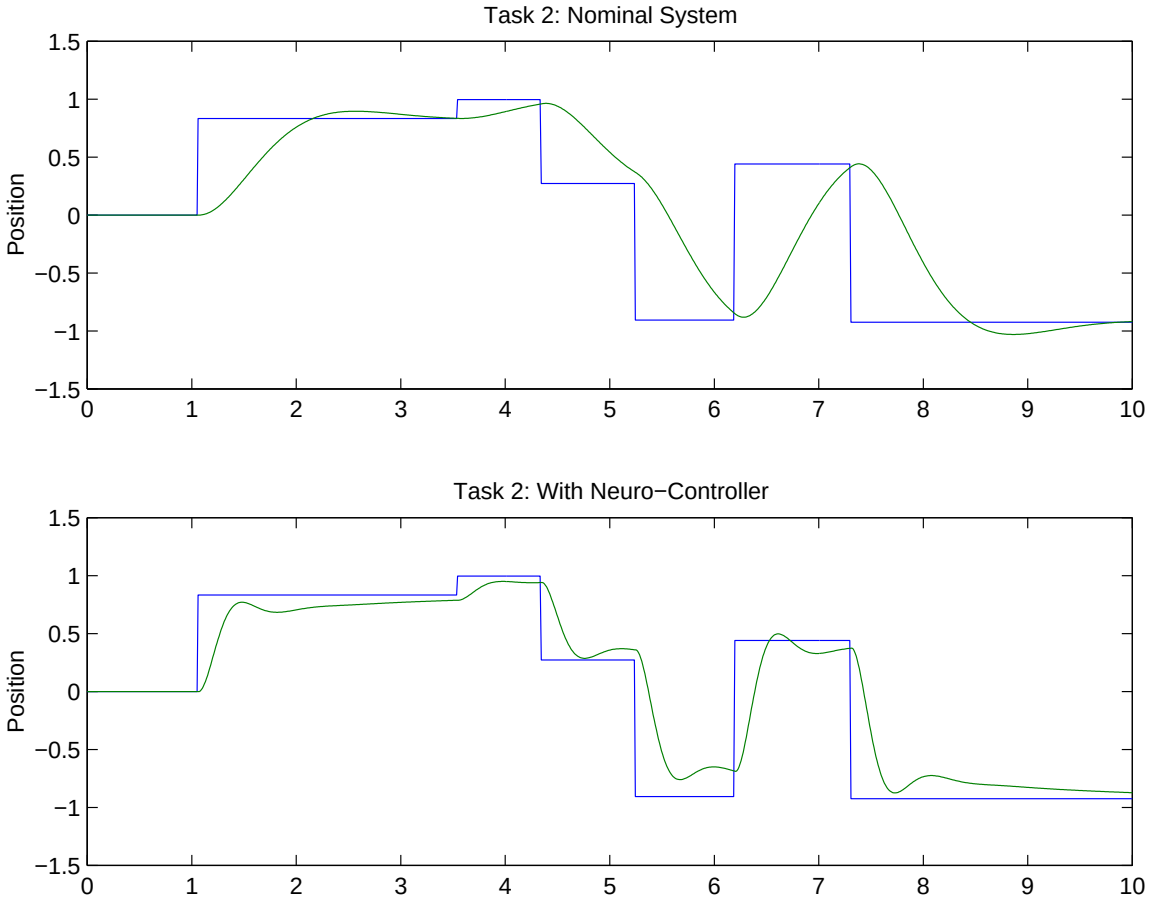


Figure 6.19: Task 2: Simulation Run

6.3.3 Stability Analysis

In the previous section, we demonstrate the ability of the neuro-controller to improve control performance. In this section, we address the stability concerns of the control system. In Figure 6.20

we see the Simulink diagram for dynamic stability computations of Task 2 using μ-analysis. This diagram is necessary for computing the maximum additives, dW and dV, that can be appended to the actor neural network weights while still retaining stability. These additives are computed anew for each pass through the stability phase. Then, during the learning phase, the actor net is trained via reinforcement learning until one of the weight changes exceeds the safety range denoted by the additives. The final weights used to produce the simulation diagram in Figure 6.19 were learned using this μ-analysis Simulink diagram.

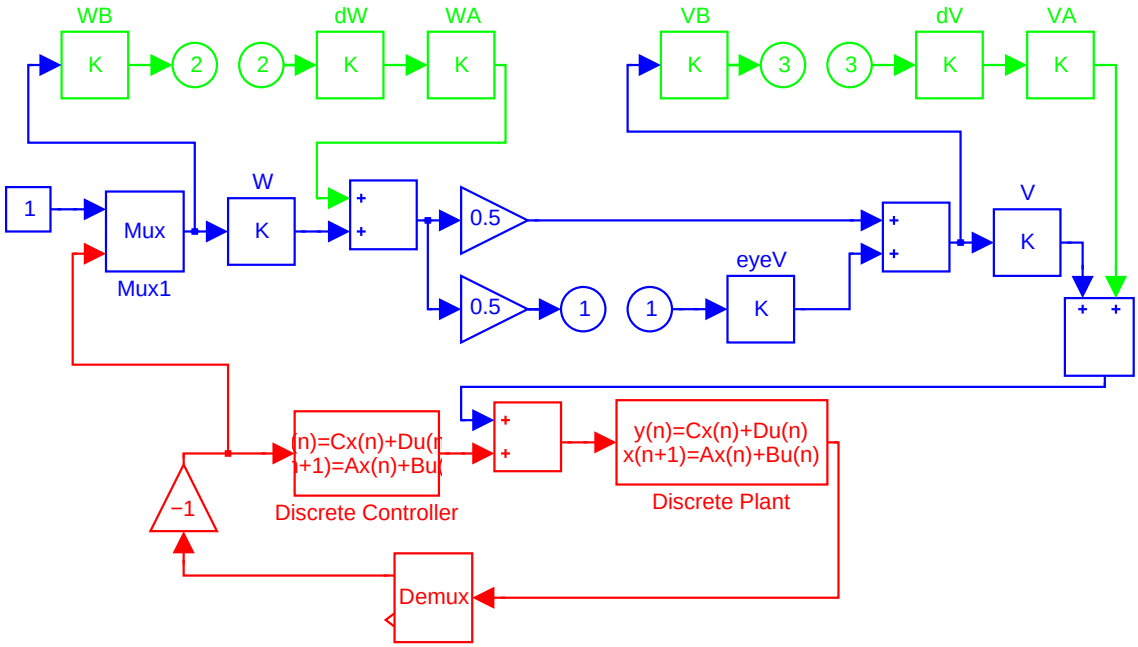


Figure 6.20: Task 2: Dynamic Stability with μ-analysis

We also repeat the learning with IQC-analysis to arrive at similar results. The Simulink diagram for IQC is shown in Figure 6.21. For IQC-analysis, we again make slight modifications to the Simulink diagram such as the IQC performance block, the IQC odd-slope nonlinearity block and the IQC slowly time-varying block. Using the IQC stability command, the optimizer finds a feasible solution to the constraint problem; thus the system is guaranteed to be stable. Again, this reinforces the identical stability result obtained with μ-analysis.

We perform three different training scenarios with Task 2. The first two training scenarios involve the stable reinforcement learning algorithm with μ-analysis and IQC-analysis, respectively. In the third training scenario, we train with only reinforcement learning and no stability analysis. All three training scenarios result in similar control performance; all three produce similar weights for the actor network. The bottom half of Figure 6.19 depicts the stable training episode using μ-analysis but the other two scenarios produce almost identical simulation diagrams. However, there is one important difference in the three scenarios. While all three scenarios produce a stable controller as an end product (the final neural network weight values), only the stable μ-analysis and IQC-analysis scenarios retain stability *throughout* the training. The stand-alone reinforcement learning scenario actually produces unstable intermediate neuro-controllers during the learning process.

For the stand-alone reinforcement learning scenario (the one without the dynamic stability guarantees) we demonstrate the actor net's instability at a point during training. Figure 6.22 depicts a simulation run of Task 2 at an intermediate point during training (the red shows the other state variable, velocity, and the teal represents the control signal, u). Clearly, the actor net is not implementing a good control solution; the system has been placed into an unstable limit cycle, because

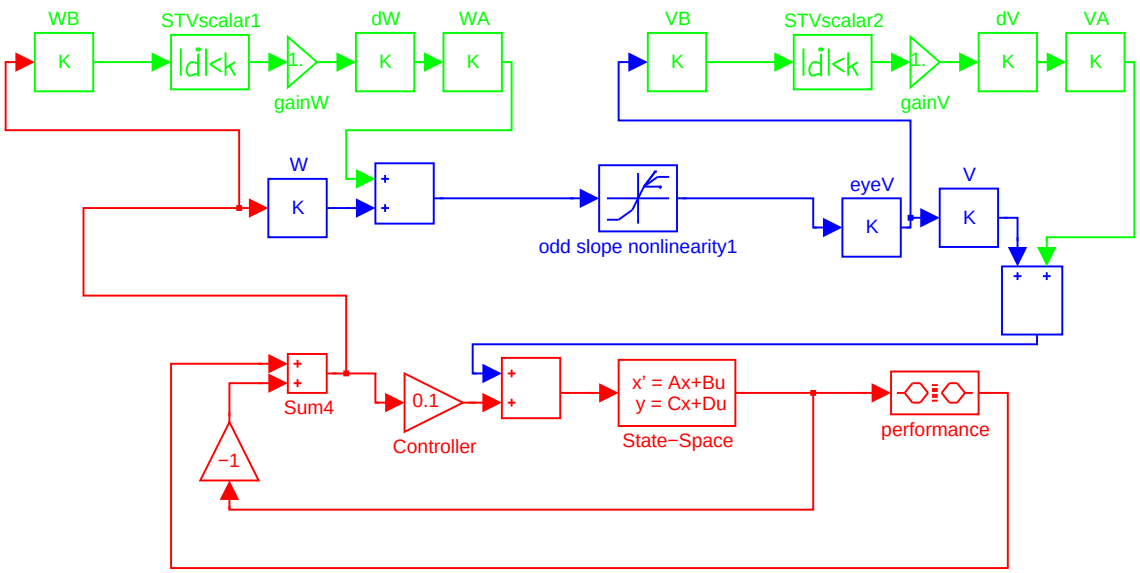


Figure 6.21: Task 2: Dynamic Stability with IQC-analysis

of the actor network. Notice the scale of the y-axis compared to the stable control diagram of Figure 6.19. This is exactly the type of scenario that we must avoid if neuro-controllers are to be useful in industrial control applications. To verify the instability of this system, we use these temporary actor network weights for a static stability test. μ-analysis reports $\mu = 1.3$ and the IQC-analysis is unable to find a feasible solution. Both of these tests indicate that the system is indeed unstable. Again, we restate the requirement of stability guarantees both for the final network (static weights) and the network during training (dynamic weights). It is the stable reinforcement learning algorithm which uniquely provides these guarantees.

In summary, the purpose of Task 2 is to construct a control system with dynamics adequately simple to be amenable to introspection, but also adequately complex to introduce the possibility of learning/implementing unstable controllers. We see in this task, that the restrictions imposed on weights from the the dynamic stability analysis are necessary to keep the neuro-control system stable during reinforcement learning.

6.4 Case Study: Distillation Column Control Task

The primary objective of this case study is to illustrate the true advantage of the stable reinforcement learning algorithm. Let us briefly review the motivation of the algorithm as we discuss how this case study demonstrates the effectiveness of a neural network based, learning controller with stability guarantees. Recall the important distinction between the actual physical system being controlled (the physical plant) and the mathematical model of the plant used to construct a controller. The mathematical model will have different dynamics than the plant, because the model is LTI (linear, time-invariant) and because the model is limited in the accuracy with which it can reproduce the dynamics of the true physical plant. Controllers designed for the LTI model may not perform well, and worse, may be unstable when applied on the physical plant. This is the fundamental difficulty that robust control is aimed at solving. However, also recall that robust control sacrifices some performance as a trade-off for guaranteeing stability on the physical plant. The stable reinforcement learning controller of this dissertation seeks to regain some of the lost performance while still maintaining stability.

In control applications, we again emphasize the distinction between the true physical plant being controlled and the mathematical model of the plant used to design a controller. Recall that an LTI

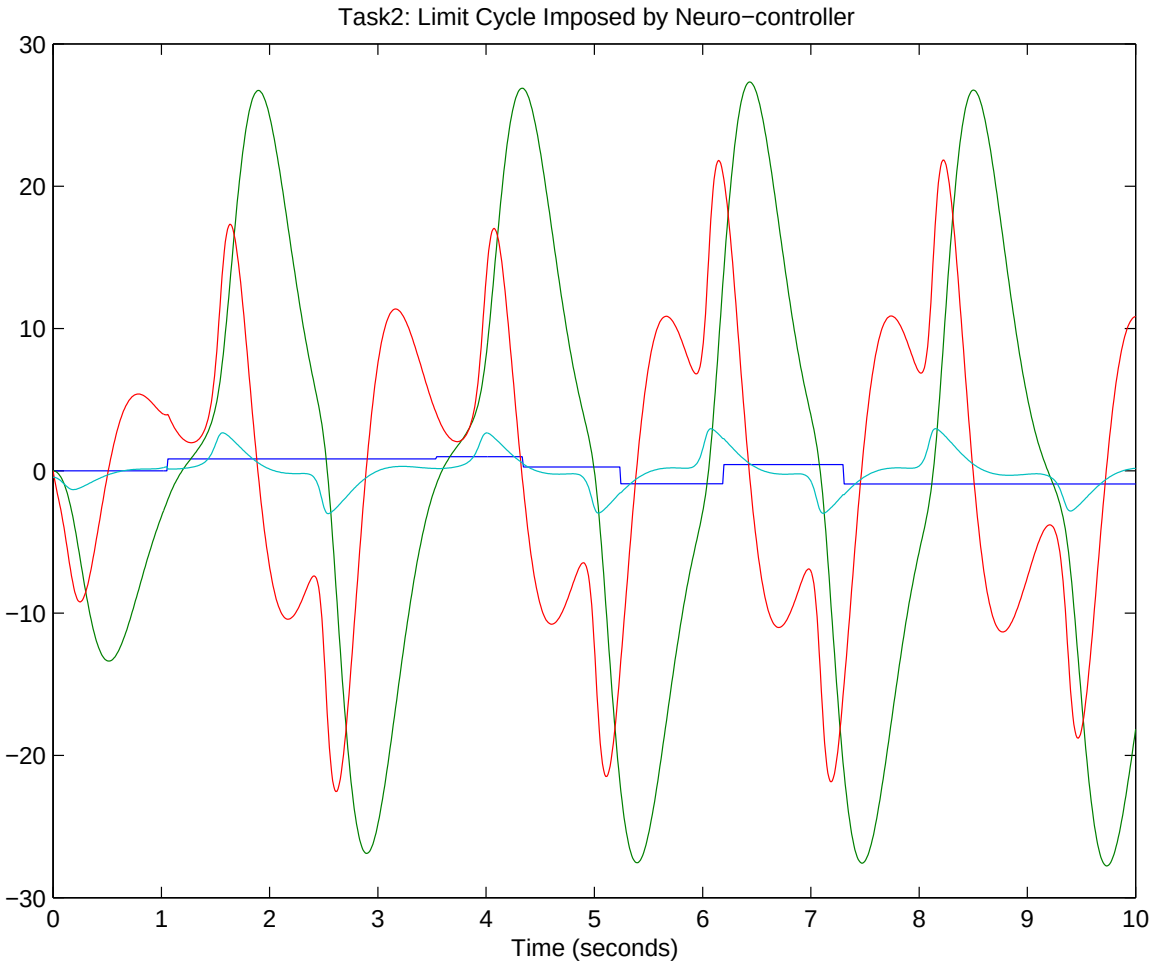


Figure 6.22: Task 2: Unstable Simulation Run

mathematical model is constructed to approximate the plant; then a stable controller is designed for the model. Because there is a difference between the physical plant and the mathematical model, the controller will operate differently on each system. If the difference is slight, then the controller should provide excellent tracking performance for both the model and the physical plant. However, if the difference between plant and model is not negligible, then the controller, which is designed for and hence operates well on the model, may provide poor performance when implemented on the physical plant. In addition to the performance issue, the controller may provide *unstable* control on the physical plant.

The tools of robust control were developed to solve these model/plant difference problems. Robust control introduces uncertainty into the plant model so that the mathematical model approximates not only the physical plant, but a whole class of *possible* physical plants. By specifying values for the uncertainty parameters, the model approximates some specific physical plant particular to those parameters. If enough uncertainty is built into the model, then there will necessarily be some specific set of parameters which *exactly* implements the dynamics of the true physical plant. It is not important that we compute these exact parameters (in fact, it is impossible), it is only important that this set of parameters exists for our mathematical model. The second step in robust control is to design a controller which provides the best possible control performance for the entire set of possible parameterized plants. That is, the controller is designed to work well with all possible physical plants that can be specified by the model. Furthermore, robust control also guarantees the

stability of the controller when implemented on any physical plant that can be realized by some set of uncertainty parameters from the model.

Figure 6.23 illustrates the difference between plant model and physical plant. Imagine that we can depict a plant as a point in plant-space; the physical plant occupies one particular point in plant-space. The plant model, without uncertainty, occupies another point in plant-space. Because the model is LTI and because our approximation has limited accuracy, the model point and the plant point are typically different. In robust control we add uncertainty to the plant model. Now the model specifies not a particular point in plant-space, but an entire region in plant-space. Each particular set of values for the model uncertainty parameters specifies one point in this model region. This region is depicted as the circle around the plant point in Figure 6.23; it is the set of all possible realizable plants from the mathematical model. If the uncertainty is large enough (the circle is wide enough), then the true physical plant is a member of the set of realizable plants – the mathematical model *covers* the physical plant.

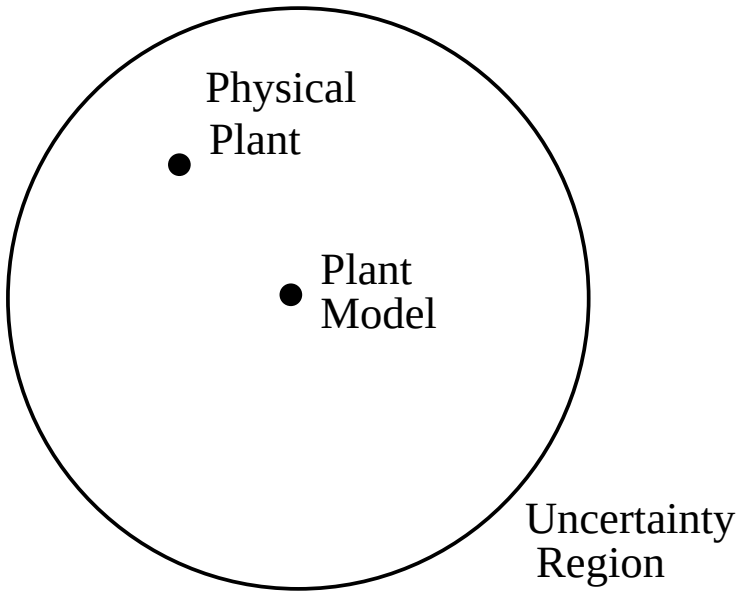


Figure 6.23: Mathematical Model vs Physical Plant

Robust control achieves its stability guarantees at a cost. Necessarily, the model uncertainty must be large enough to be certain that the physical plant is covered by the model. Often the model is overly conservative in that it specifies plants with more unstable control dynamics than exist in the real physical plant. As a result of being overly conservative, the robust controller must sacrifice a degree of aggressiveness; this controller must lose some of its tracking performance in order to ensure stability. The stable reinforcement learning algorithm regains some of this lost performance while still retaining the stability guarantees.

The distillation control column will bring all these issues to the forefront. As this case study is a bit lengthy and complex, we break the analysis down into the following steps. First, we discuss the dynamics of the distillation column process and show why it is a difficult control problem. We then present a decoupling controller, a typical approach to solving this control problem, and show why the decoupling controller fails to solve this control problem; namely, the differences between the LTI model and the physical plant make the decoupling controller ineffective. A robust controller is then designed and we see how the robust controller addresses the short comings of the decoupling controller. Finally, we apply the stable reinforcement learning controller on top of the robust controller; the reinforcement learner is able to regain some of the lost performance margin sacrificed in the robust control design.

6.4.1 Plant Dynamics

Figure 6.24 is Skogestad's depiction of the distillation column [Skogestad and Postlethwaite, 1996]. Without concerning ourselves with the rather involved chemistry, we summarize the dynamics of the distillation column. The two output variables, y_1 and y_2, are the concentrations of chemical A and chemical B, respectively. We control these concentrations by adjusting two flow parameters: u_1 = L flow and u_2 = V flow. The reference inputs, r_1 and r_2, and the outputs are scaled so that $r_1, r_2, y_1, y_2 \in [0, 1]$.

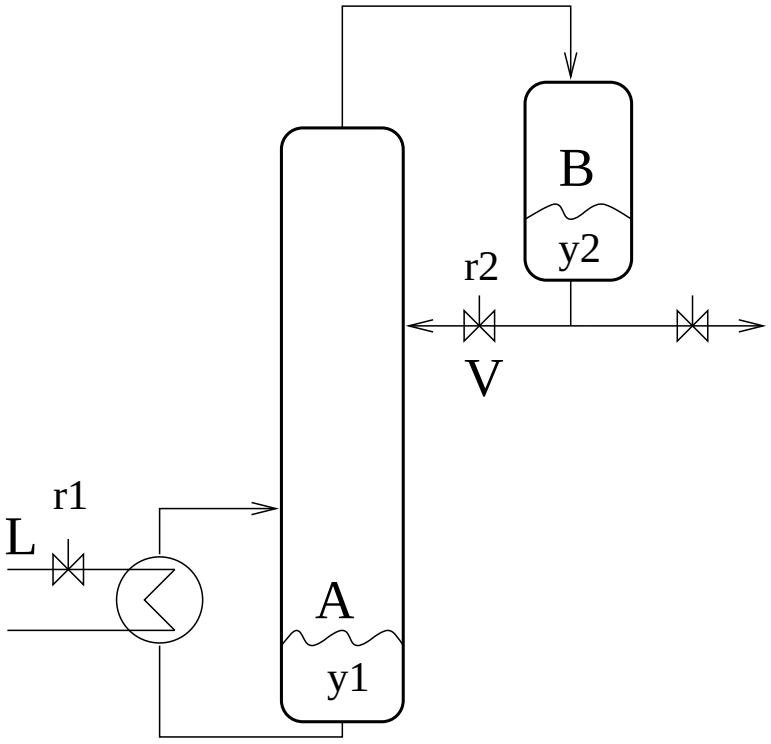


Figure 6.24: Distillation Column Process

Again, we model the distillation column process with the block diagram in Figure 6.25. Since the distillation column has two outputs and two control variables, we use a 2x2 matrix to capture the dynamics of the plant, G. Skogestad has sampled a real distillation column to arrive at the following LTI model:

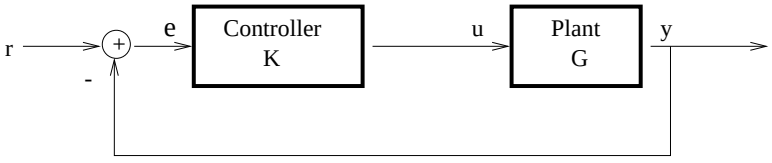


Figure 6.25: Distillation Column Process: Block Diagram

$$G(s) = \begin{pmatrix} \frac{87.8}{75s+1} & \frac{-86.4}{75s+1} \\ \frac{108.2}{75s+1} & \frac{-109.6}{75s+1} \end{pmatrix} \tag{6.10}$$

$$Y(s) = G(s)U(s) \tag{6.11}$$

Since we implement the neuro-controller using a digital system, we approximate Skogestad's continuous-time plant given above with the following discrete-time, state space system:

$$x(k+1) = Ax(k) + Bu(k) \tag{6.12}$$
$$y(k) = Cx(k) + Du(k) \tag{6.13}$$

where

$$\begin{aligned} A &= \begin{pmatrix} 0.99867 & 0 \\ 0 & 0.99867 \end{pmatrix} \qquad B = \begin{pmatrix} -1.01315 & 0.99700 \\ -1.24855 & 1.26471 \end{pmatrix} \\ C &= \begin{pmatrix} -0.11547 & 0 \\ 0 & -0.11547 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \end{aligned} \tag{6.14}$$

See [Skogestad and Postlethwaite, 1996; Phillips and Harbor, 1996] for details on converting from continuous-time plants to discrete-time plants. The sampling interval, k, is one second. In order to see why this is a difficult control problem, Skogestad computes the singular value decomposition of the plant, G. We can ignore the $\frac{1}{75s+1}$ term in the denominator and compute the SVD of only the numerator of G:

$$G_{num} = \begin{pmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{pmatrix} \tag{6.15}$$

$$G_{num} = \begin{pmatrix} 0.625 & -0.781 \\ 0.781 & 0.625 \end{pmatrix} \begin{pmatrix} 197.2 & 0 \\ 0 & 1.39 \end{pmatrix} \begin{pmatrix} 0.707 & -0.708 \\ -0.708 & -0.707 \end{pmatrix} \tag{6.16}$$

From the SVD, Skogestad points out that inputs aligned in opposite directions ($[0.707, -0.708]^T$) produce a large response in the outputs (indicated by singular value of 197.2). Conversely, inputs aligned in the same direction ($[-0.708, -0.707]^T$) produce a minimal response in the output (singular value = 1.39). The distillation column plant is highly sensitive to changes in individual inputs, but relatively insensitive to changes in both inputs. Control engineers call this plant *ill-conditioned* meaning the ratio of the largest and smallest singular values is much larger than unity. Thus, this plant is a rather challenging control problem.

Here again we return to the distinction between the plant model and the physical plant. The system, G, given by Equation 6.10 is a *model* of a physical plant. Skogestad collected data on the steady-state behavior of a real distillation column and then constructed G as an LTI model to approximates the physical plant. There are two primary reasons why the model and the physical plant will differ. First, the model must be LTI to apply the robust control design tools; the physical plant almost certainly contains some non-LTI dynamics. Second, because a finite amount of data has been collected from the physical plant, our model only approximates the physical plant; Skogestad selects the type of LTI model and the model parameters for a statistical best-fit with the given plant data. But, the model will never be an exact fit for the physical plant.

In order to apply the robust control design tools, we must incorporate uncertainty into the plant model so that the model covers the dynamics of the physical plant. Skogestad, who has the original data set available, selects the type and amount of uncertainty appropriate for this model. Multiplicative uncertainty is incorporated to each input control path, u_1 and u_2, in the amount of $\pm$ 20% gain. Figure 6.26 shows the system with 20% gain uncertainty on each input.

Keep in mind that we will never know the true dynamics of the physical plant. We can, however, be certain that the dynamics of the physical plant can be exactly matched by choosing a possibly time-varying function which increases/decreases the input path by a maximum of 20%. The size of the uncertainty, $\pm$ 20% on the input path, is typical for a control problem like the distillation column.

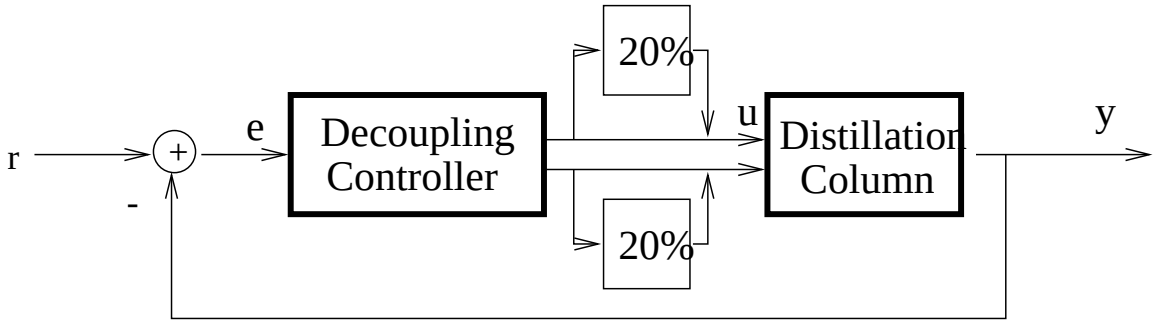


Figure 6.26: Distillation Column Model with Input Gain Uncertainty

At this point, we would design a controller using the LTI model with uncertainty. Then we implement the controller on the real distillation column to test its effectiveness on the physical plant. However, we do not have the distillation column available (nor did Skogestad) and thus we cannot test our controller on the real system. Instead, we construct a second model of the system to represent the real system. To simulate our physical distillation column, we use the original model G and then amplify input u_1 by 20% while decreasing input u_2 by 20%. Notice this falls within the bounds of the uncertainty and thus should be covered by any controller design which accounts for the uncertainty in the original LTI model. In summary, we design controllers on the original LTI model G and then test the controllers on the simulated physical plant given by $G|u_1 \rightarrow 1.2u_1, u_2 \rightarrow 0.8u_2$.

We present three controllers for the distillation column task. The first two controllers, given by Skogestad, are a decoupling controller and a robust controller. The difference between the LTI model and the [simulated] physical plant demonstrates the problems with conventional control techniques. Skogestad's robust controller solves many of the problems with the decoupling controller. It is apparent that Skogestad selected this control problem specifically to motivate the approach of robust control design. The third controller we present is learned with our stable reinforcement learning scheme. Here we show that the learned controller offers the same advantages as the robust design and is able to achieve slightly improved tracking performance.

6.4.2 Decoupling Controller

Now that we have shed light on the plant G and its challenging dynamics, we present Skogestad's decoupling controller [Skogestad and Postlethwaite, 1996]. The decoupling controller uses advanced techniques representative of current approaches to control design. The controller *decouples* the two inputs in order to overcome the ill-conditioned nature of the plant. Simply, the decoupling controller will invert the dynamics of the plant in attempt to have input u_1 affect only output y_1 and input u_2 affect only output y_2. The decoupling controller is essentially the inverse matrix G^{-1}. The details of the decoupling controller are given as:

$$K_{decup}(s) = \frac{0.7}{s} G^{-1}(s) = \frac{0.7(1+75s)}{s} \begin{pmatrix} 0.3994 & -0.3149 \\ 0.3943 & -0.3200 \end{pmatrix} \tag{6.17}$$

We implement Skogestad's decoupling controller and plot its dynamic response when applied to the LTI model. Figure 6.27 shows the plant response to a step change in one of the inputs. As seen in the diagram, the output y_1, shown in red, quickly rises to track the step change in the input $u_1 = 0 \rightarrow 1$, shown in blue. Output y_2 (cyan), has been "decoupled" from input u_1 and thus remains constant at $y_2 = 0$. The decoupling controller appears to serve this difficult control task well.

Notice that the decoupling controller perfectly inverts the LTI model of the plant – not the physical plant. The dynamics of the physical plant are unknown and thus we cannot design the decoupling controller to exactly invert the physical plant. As shown in Figure 6.27, we expect the performance of the decoupling controller to be excellent on the LTI model. The hope is that the

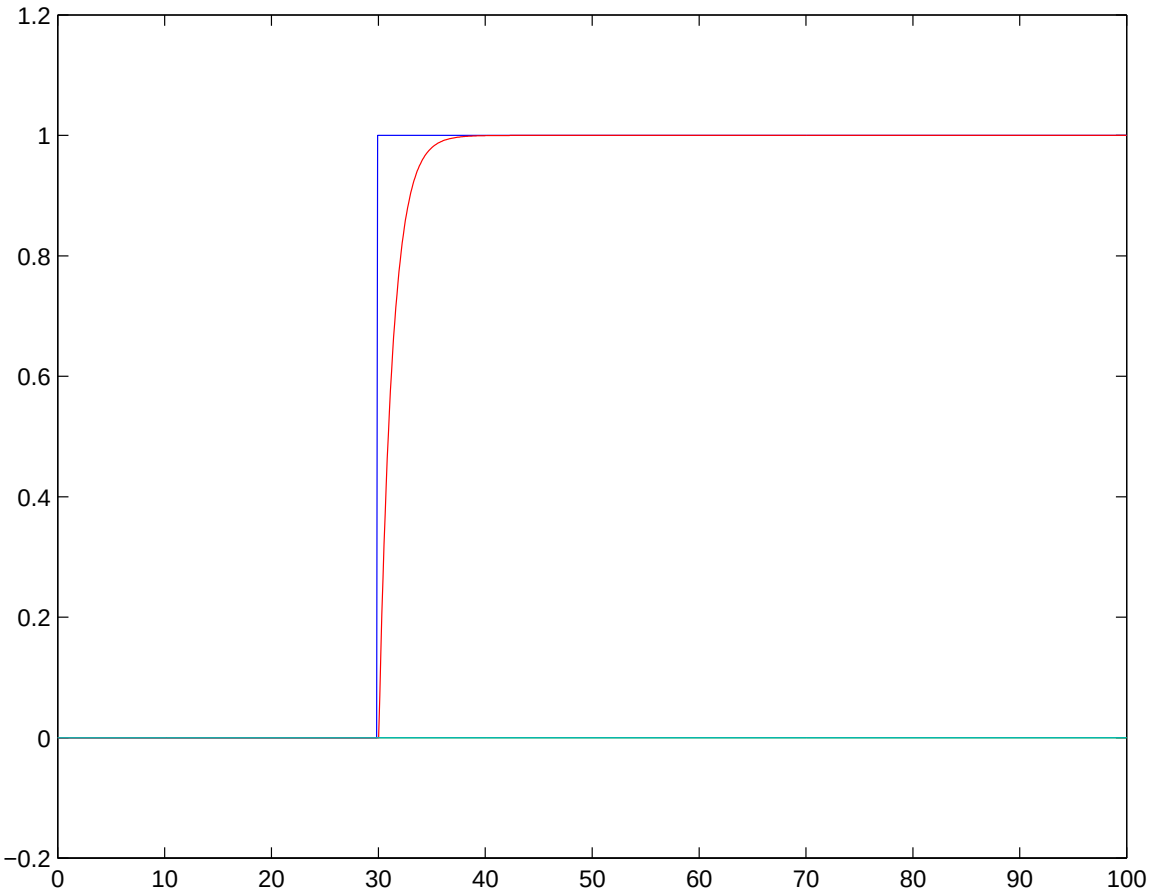


Figure 6.27: Step Response: LTI Model with Decoupling Controller

physical plant has dynamics similar enough to the LTI model that the controller will perform well on the physical plant also.

Figure 6.28 depicts the step response of the decoupling controller on the simulated physical plant. Although the decoupling controller performs well on the plant model, the tracking performance of the decoupling controller on the physical plant is rather poor. Output y_1 rises to a lofty height of 6.5 before decaying back to its desired value of 1.0. Even worse, output y_2 rockets up past 7.0 before dropping back to 0; the "decoupling controller" is clearly not decoupling anything in the physical plant.

The poor performance of the decoupling controller on the real plant is a result of the decoupling controller being highly tuned to the dynamics of the LTI model; it exploits the model's dynamics in order to achieve maximal performance. Even though the dynamics of the physical plant are very close (20% uncertainty on the inputs is not a substantial uncertainty), the decoupling controller's performance on the simulated physical plant is quite different. The plant's ill-conditioned nature comes back to strike us here; it is fundamentally a difficult control problem and the uncertainty will render "optimal designs" like the decoupling controller *non-robust*.

6.4.3 Robust Controller

To address the problems encountered when the decoupling controller is implemented on the physical plant, Skogestad designs a robust controller [Skogestad and Postlethwaite, 1996]. Here we duplicate Skogestad's work in designing a robust controller using the Matlab μ-synthesis toolbox [Balas et al.,

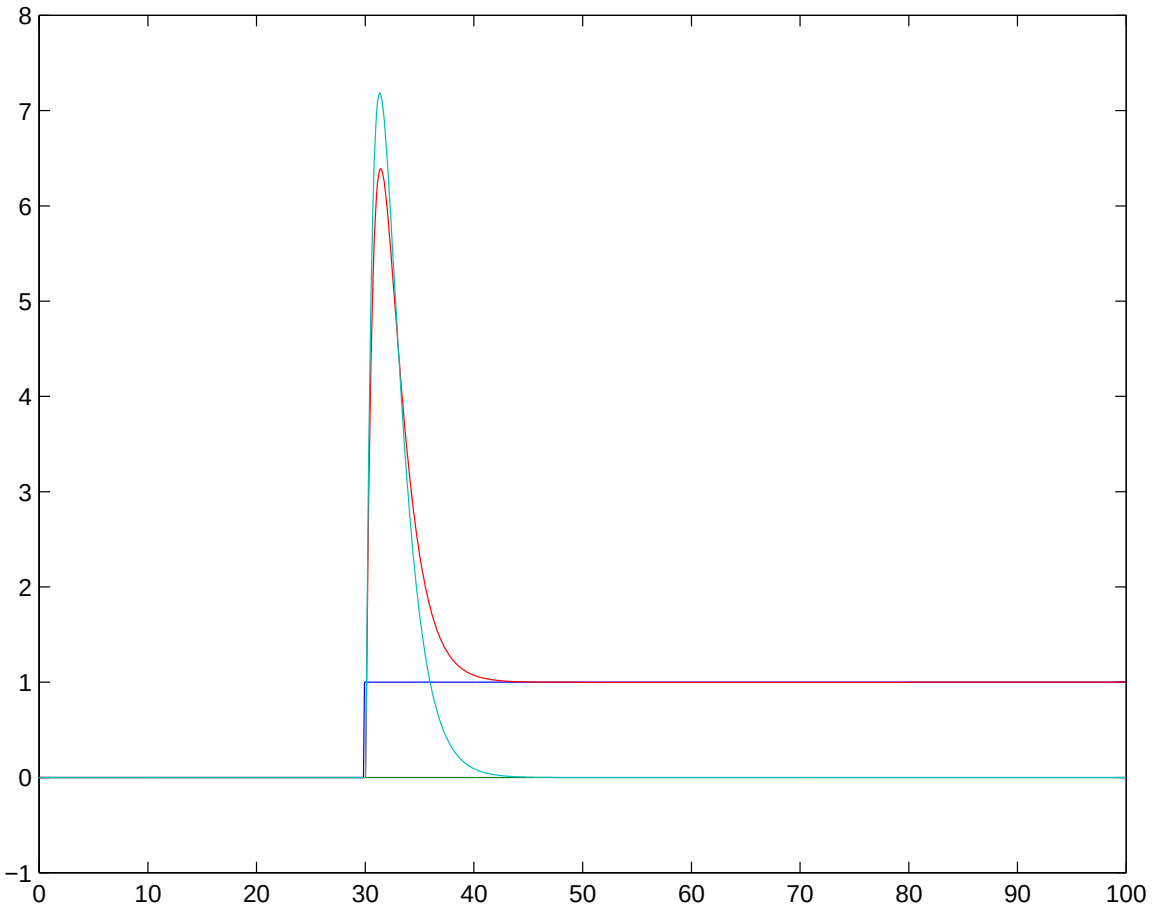


Figure 6.28: Step Response: Physical Plant with Decoupling Controller

1996; Skogestad and Postlethwaite, 1996]. The resulting robust controller is an eighth order (8 internal states) controller that is too complex to process analytically. We implement the robust controller and see that good performance is achieved for the LTI model (Figure 6.29) and the simulated physical plant (Figure 6.30).

It is important to note that the robust controller does not match the performance of the decoupling controller on the LTI model. Once again, this is because the decoupling controller exploits dynamics in the plant model to achieve this extra performance. The robust controller is "prohibited" from exploiting these dynamics by the uncertainty built into the model. Thus, the robust controller will not perform as well as a controller optimally designed for one particular plant (such as the LTI model). However, the robust controller will perform fairly well for a general class of plants which possess dynamics similar to the LTI model. In summary, we sacrifice a margin of performance for the *robustness* of a robust controller.

One of the criticisms of robust control is that the performance sacrifice might be larger than necessary. A degree of conservativeness is built into the robustness design process in order to achieve the stability guarantees. The stable reinforcement learning controller of the next subsection attempts to regain some of this lost performance.

6.4.4 Stable Reinforcement Learning Controller

Next, we apply the stable reinforcement learning controller with the goal of regaining some of the performance lost in the robust control design. We add a neuro-controller to the existing robust

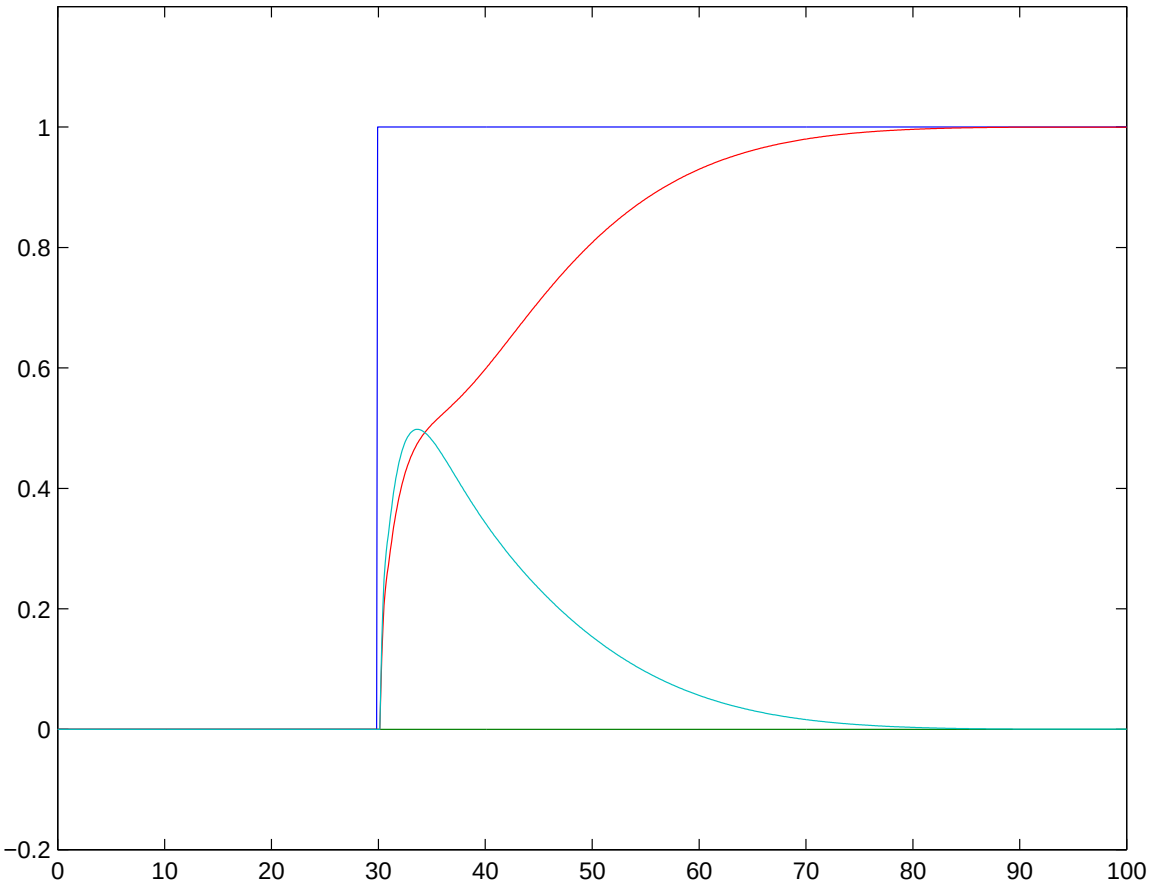


Figure 6.29: Step Response: LTI Model with Robust Controller

controller to discover the non-LTI dynamics which exist in the physical plant but not the LTI model. The neuro-controller learns, via reinforcement learning, while interacting with the simulated physical plant. In effect, the reinforcement learner discovers more information about the dynamics of the physical plant and exploits this extra information not available to the robust controller.

In the previous case studies, the state information of the system is small. Task 1 had just one state variable (x the position) while Task 2 had three state variables (position/velocity in the plant and a state variable for the controller). Furthermore, the dynamics of these first two case studies were simple enough that the neuro-controller could learn good control functions without using all the state information; only the tracking error was required (which is captured in the position state variable mentioned above). For the distillation column, the *state* of the system is quite large. To capture the full state of the system at any point in time we require the following:

- the two reference inputs: r_1 and r_2
- the internal state of the robust controller: 8 states
- the internal state of the plant: x_1 and x_2

There are a total of twelve state variables. To train on the full state information requires a actor net with 13 inputs (one extra input for the bias term) and a critic net with 14 inputs (two extra inputs for the "actions" of the actor net). These networks need an extraordinary amount of memory and training time to succeed in their neuro-control role. This issue is addressed in Chapter 7. Consequently, we select a small subset of these states for use in our network. To the actor net, we

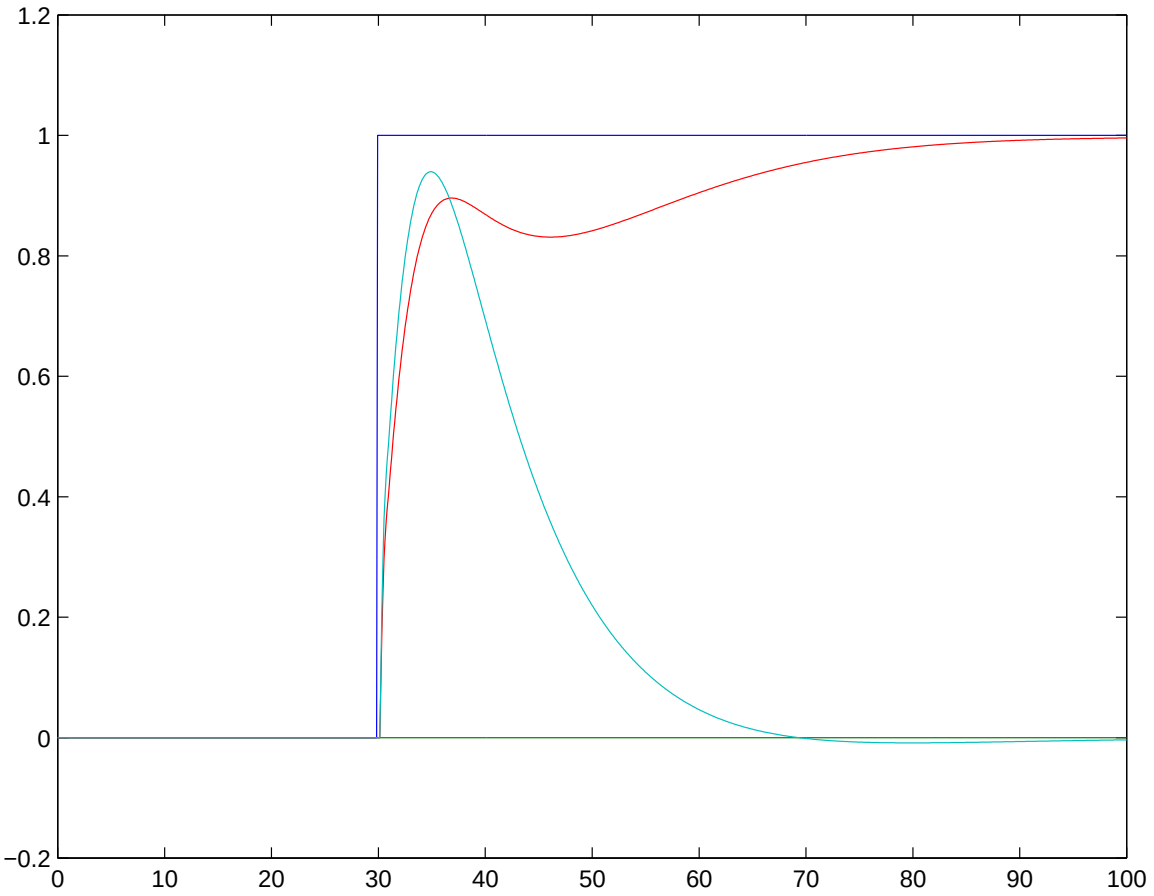


Figure 6.30: Step Response: Physical Plant with Robust Controller

use the two tracking errors (e_1, e_2 – which, again are essentially duplicates of the concentrations x_1, x_2) as the two inputs to the actor neural network. The actor network has two output units for the control signals $\hat{u}_1$ and $\hat{u}_2$. We select four hidden units for the actor network as this proved to be the most effective at learning the required control function quickly.

The critic net is a table look-up. It is a four dimensional table with inputs of the state (e_1, e_2) and the action $(\hat{u}_1, \hat{u}_2)$. The resolution is ten for each dimension resulting in 10^4 "entries" in the table. The actor-critic network is trained for 500,000 samples (representing 500,000 seconds of elapsed simulation time) during which one of the two reference inputs (r_1, r_2) is flipped $\{0 \rightarrow 1, 1 \rightarrow 0\}$ every 2,000 seconds. The learning rates are $\alpha = 0.01, \beta = 0.001$ for the critic and actor networks, respectively.

We perform two training runs. The first training run contains no robust stability constraints on the neuro-controller. Reinforcement learning is conducted without regard for bounding boxes and dynamic stability analysis. As a result, the actor network implements several unstable controllers during the non-robust training run. An example of such a controller is shown in Figure 6.31. We conduct a μ-analysis static stability test on these particular actor network weight values and find $\mu = 1.2$. Recall, $\mu > 1$ implies unstable behavior. This compares with the $\mu = 0.22$ achieved for the final weight values obtained during the stable training run discussed below.

In the second training run, we use the full stable reinforcement learning algorithm. Using μ-analysis for our dynamic stability test, we transform the distillation column into the Simulink diagram in Figure 6.32. By using the dynamic stability theorem, we guarantee that the neuro-controller learned via reinforcement learning will never produce an unstable control situation. After training

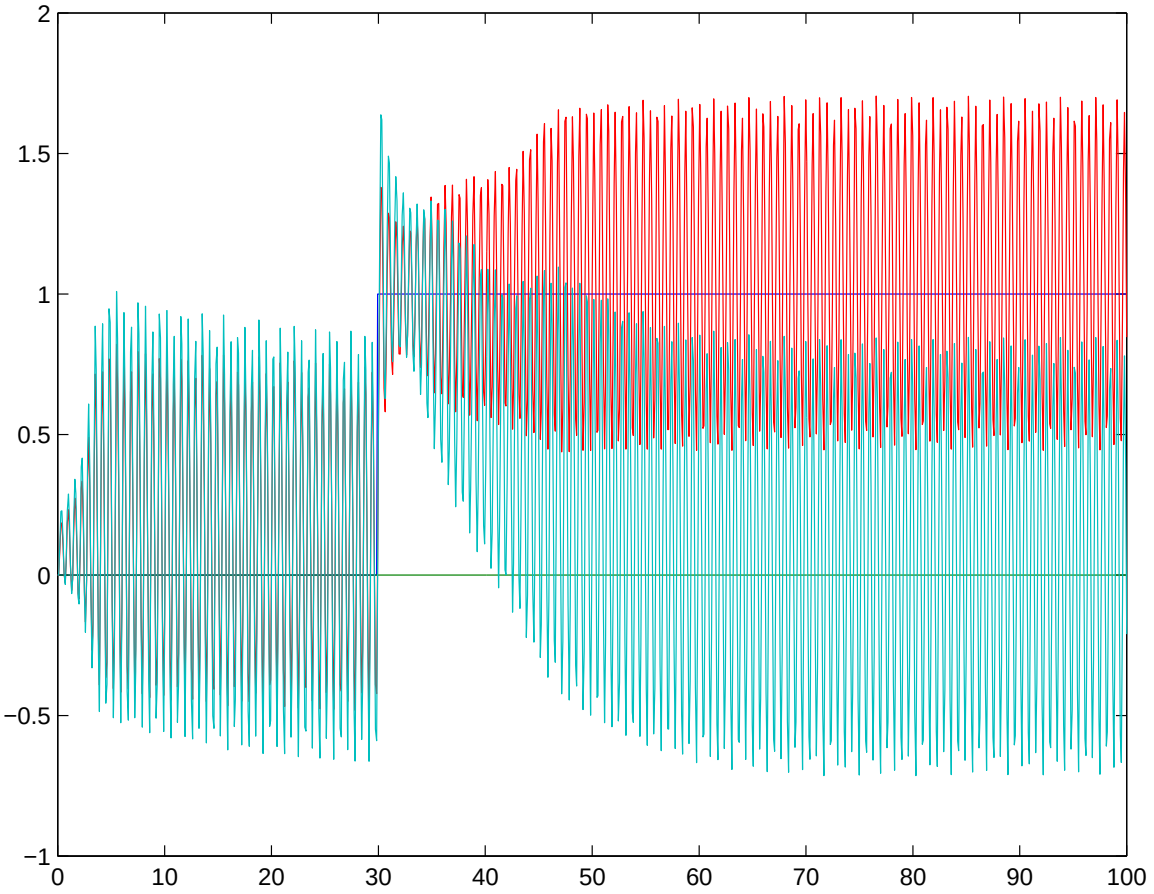


Figure 6.31: Perturbed Distillation Column with Unstable Neuro-controller

is complete, the network improves the tracking performance as shown in Figure 6.33. Compare this result to the robust controller alone in Figure 6.30. Although the two graphs seem similar, the addition of the neuro-controller improves the mean square tracking error of 0.286 with the robust controller to 0.243. This is clearly a distinct gain in tracking performance of approximately 15%.

In the following table, we summarize the tracking performance of various controllers by measuring the sum squared tracking error.

Sum Squared Tracking Error

	Plant Model	**Real Plant**
Decoupling Controller	$1.90\text{x}10^{-2}$	$6.46\text{x}10^{-1}$
Robust Controller	$2.57\text{x}10^{-1}$	$2.86\text{x}10^{-1}$
Neuro-Controller	Not Applicable	$2.43\text{x}10^{-1}$

In summary, the decoupling controller performs quite well on the plant model, but its performance on the physical plant is unacceptable. We would expect similar results from "optimal control" methods such as H_2 optimal design and H_∞ optimal design [Skogestad and Postlethwaite, 1996].

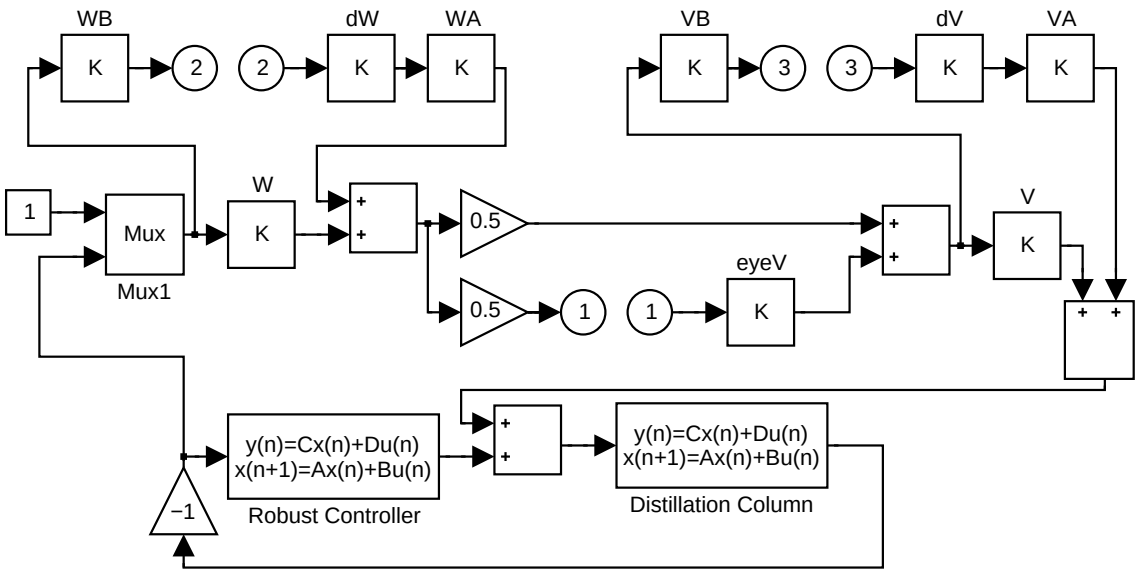


Figure 6.32: Simulink Diagram for Distillation Column

The robust controller does not perform nearly as well as the highly optimized decoupling controller on the LTI model. However, when applied to the physical plant, the robust controller halves the tracking error of the decoupling controller. Even more impressive than the reduction in tracking error is the significantly better step response of the robust controller. (Compare Figure 6.28 and Figure 6.30, be wary of the changed y-axis in Figure 6.28). Finally, we add the neuro-controller. By applying the stable reinforcement learning algorithm, the system retains stability and we are able to improve the tracking performance over the robust controller by 15%. It should be noted that we encounter considerable difficulty in achieving this performance gain; much of the difficulty is attributed to the massive learning experience and sensitive dependence on learning algorithm parameters. These problems are explored in the concluding chapter (see Chapter 7).

6.5 Case Study: HVAC Control Task

The HVAC (Heating, Ventilation, and Air Conditioning) problem is a difficult control problem receiving much attention in recent and past research. The present methods for control are satisfactory in most cases, but there is significant room for improvement, both in terms of human comfort and particularly energy savings. HVAC systems are highly nonlinear with widely varying dynamics at different operating points. It is difficult, if not impossible, to construct LTI models of the system which exhibit dynamics similar to the physical plant dynamics. Such systems also incur highly variable gains at different operating points. The different components of an HVAC system, (heating coils, fans, dampers, etc) are highly interactive and cannot be modeled as isolated units. HVAC systems depend heavily on unpredictable scheduling; changes in weather conditions and unpredictable human activities contribute to the difficulty of the HVAC problem. Traditional adaptive control techniques are often ineffective, because these techniques make assumptions about the underlying dynamics of the system and the form of the system.

There has been some success with the introduction of neural networks into the HVAC control scheme since the networks excel at discovering the unmodeled, nonlinear dynamics. There has also been some initial success using reinforcement learning algorithms to further tune the control process. This research suggests that neuro-control is well-suited for the HVAC control problem [Anderson et al., 1996].

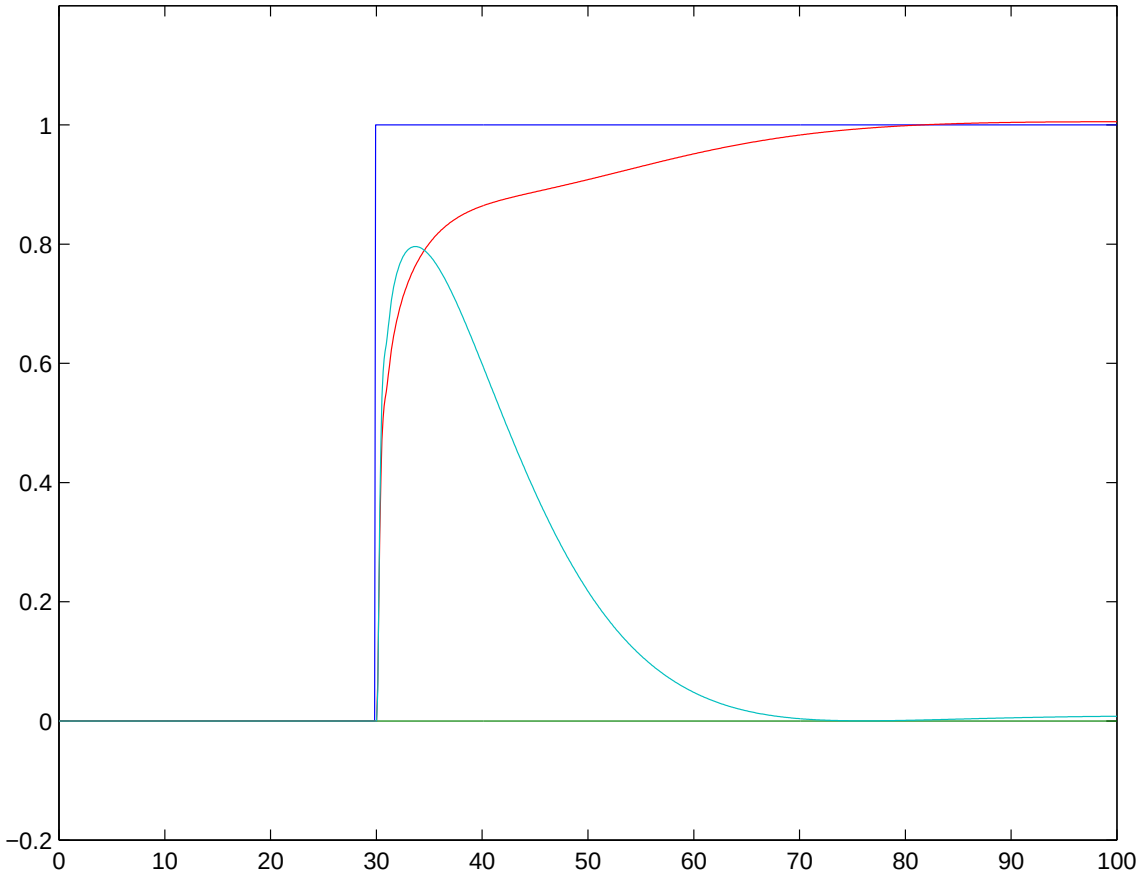


Figure 6.33: Perturbed Distillation Column with Neuro-controller

The research for this dissertation is concurrent with a three year National Science Foundation grant to study the application of robust control and reinforcement learning to the HVAC control problem [Anderson et al., 1998]. In this HVAC study, we are constructing an actual physical heating coil as a laboratory for neuro-control testing. This is among the first attempts at implementing neuro-control schemes on real physical HVAC hardware. At the time of this writing, the heating coil construction is still in progress and is not available for testing. Future experiments with the heating coil hardware are discussed in Chapter 7. Figure 6.34 depicts hardware of the heating coil experimental laboratory when it will be completed in the future.

In lieu of hardware experiments, we construct a nonlinear software model of the system. We adopt a heating coil model from Underwood and Crawford and then alter the parameters to fit our hardware [Underwood and Crawford, 1991]. The nonlinear model exhibits many of the HVAC difficulties discussed above and is therefore suitable for testing and comparing different control schemes including those which are neural network based and those which are designed according to robust control principles. Thus, this case study is tested with the nonlinear software model and not on the physical HVAC system.

6.5.1 HVAC Models

In this subsection, we present the detailed nonlinear software model used for the experiments for this case study. The heating coil (plant) has three internal state variables, three external state variables, and one input variable.

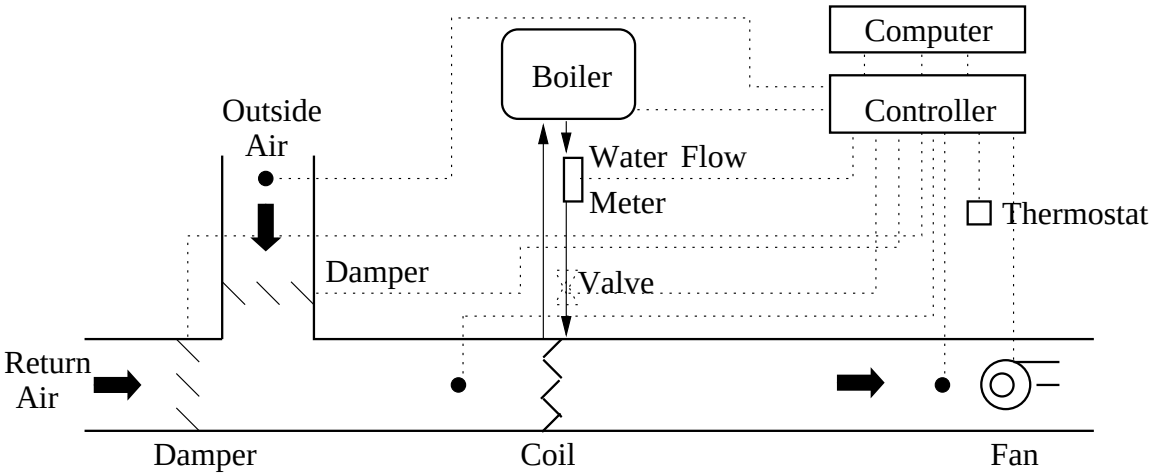


Figure 6.34: HVAC Hardware Laboratory

The three external state variables are the temperature of the air at the intake damper, T_{ai}, the temperature of the water at the input coil, T_{wi}, and the flow rate of the air moving through the duct, F_a. The two temperatures are determined by ambient environmental conditions and the air flow rate is constant and determined by the fan and the ducts. None of these three state variables change due to our control scheme, hence we call them external state variables.

The single plant input variable is the valve setting on the water coil. By changing the valve setting, we can increase or decrease the flow rate of the water in the heating coil. Indirectly, we also affect the output air temperature, because the flow rate of the water determines how much thermodynamic energy can be delivered from the boiler to the heating coil in the duct. The valve setting, an input to the plant, is the output from the controllers.

The three internal state variables for the system change as a result of the valve setting. The flow rate of water, F_w, is obviously directly affected by the valve setting. In turn, this also affects the temperature of the water leaving the coil, T_{wo}, and ultimately the temperature of the air leaving the coil, T_{ao}. T_{ao} is the state variable that we desire to control. Our control performance is determined by how closely the output air temperature tracks the reference signal, which, in the HVAC case, is the desired thermostat setting or the set point.

The discrete-time, dynamic, nonlinear, HVAC model is specified by the following update equations:

$$F_w = 6.72\text{x}10^{-10}u^3 - 2.30\text{x}10^{-6}u^2 + 2.18\text{x}10^{-3}u, \tag{6.18}$$

$$\begin{aligned} T_{wo} = {} & T_{wo} + 0.649F_wT_{wi} - 0.649F_wT_{wo} \\ & - 0.012T_{wi} - 0.012T_{wo} + 0.023T_{ai} + 0.104F_wT_{ai} \\ & - 0.052F_wT_{wi} - 0.052F_wT_{wo} + 0.028F_aT_{ai} \\ & - 0.014F_aT_{wi} - 0.014F_aT_{wo}, \end{aligned} \tag{6.19}$$

$$\begin{aligned} T_{ao} = {} & T_{ao} + 0.197F_aT_{ai} - 0.197F_aT_ao \\ & + 0.016T_{wi} + 0.016T_{wo} - 0.032T_{ai} \\ & + 0.077F_aT_{wi} + 0.077F_wT_{wo} - 0.015F_wT_{ai} \\ & + 0.022F_aT_{wi} + 0.022F_aT_{wo} - 0.045F_aT_{ai} \\ & + 0.206T_{ai(k-1)} - 0.206T_{ai}, \end{aligned} \tag{6.20}$$

where u is the valve setting and $T_{ai(k-1)}$ refers to the air input temperature on the previous time step.

To construct traditional and robust controllers for the system, we must also derive an LTI model. Due primarily to the complex dynamics of HVAC systems, a single LTI model is not adequate for approximating the dynamics of the nonlinear system. Consequently, we limit ourselves to constructing an LTI model that is reasonably accurate for only a limited operating range (around a set point temperature with static environmental variables). We use a Taylor Series expansion about the desired operating point to construct the LTI model of the system. Recall, we use the LTI model for designing controllers and then use the nonlinear model (instead of the hardware) for testing the stability and performance of the controllers. The following parameters specify the operating point for the Taylor Series expansion:

$$u = 972.9 \quad F_W = 0.2785 \tag{6.21}$$

$$T_{wo} = 55.45 \quad T_{wi} = 78.0 \tag{6.22}$$

$$T_{ao} = 45.0 \quad T_{ai} = 12.0 \tag{6.23}$$

The resulting linear model is specified by:

$$F_w = 0.2785 - (3.863\text{x}10^{-4}(u - 972.9)), \tag{6.24}$$

$$T_{wo} = 93.5445(F_w - 0.2785) + 0.792016(T_{wo} - 55.45) + 55.45, \tag{6.25}$$

$$\begin{aligned} T_{ao} = {} & 0.8208(T_{ao} - 45.0) + 45.0 + 0.0553(T_{wo} - 55.45) \\ & + 7.9887(F_w - 0.2785). \end{aligned} \tag{6.26}$$

6.5.2 PI Control

As PI control is a dominant trend in the HVAC industry, we construct a PI controller (proportional plus integral) using state-of-the-art tuning laws [Cock et al., 1997]. The tracking performance of the PI controller when implemented on the nonlinear model is shown in the top time-plot of Figure 6.35a. The control performance is quite good as the the PI controller has been finely tuned to suit this particular nonlinear model. The PI controller we used is given by:

$$u = \begin{cases} 670 & \text{if} \quad u < 670 \\ K_p e + \int K_i de & \text{if } 670 < \quad u < 1400 \\ 1440 & \text{if } 1400 < \quad u \end{cases} \tag{6.27}$$

where $K_p = 135$ and $K_i = 13$. As indicated by the equations above, the controller has hard limits at 670 and 1400 to reflect the maximum valve opening and minimum valve opening, respectively.

6.5.3 Neuro-control

In constructing a reinforcement learning controller for the heating coil, we must decide which state variables to include as input signals to the neuro-controller. The PI controller receives only the tracking error, e (which is essentially the same as the internal state variable T_{ao}). There are other state variables governing the dynamics of the plant; there is extra information in these state variables that could be exploited by a controller to provide better control performance. We must decide which state variables to use as inputs to the neuro-controller. By including more state variables, we provide extra information to the controller; this information may or may not be useful for improved control performance. However, additional control inputs require a larger neural network and hence additional training time. We must make a difficult and complex decision about which subset of state variables provide the best combination of control performance and learning speed.

We construct two different neuro-controllers. In the first agent, we use minimal state information of the plant and system; specifically, we submit only the tracking error as input to the actor network. In effect, this agent can only learn to improve the control by adjusting the P (proportional)

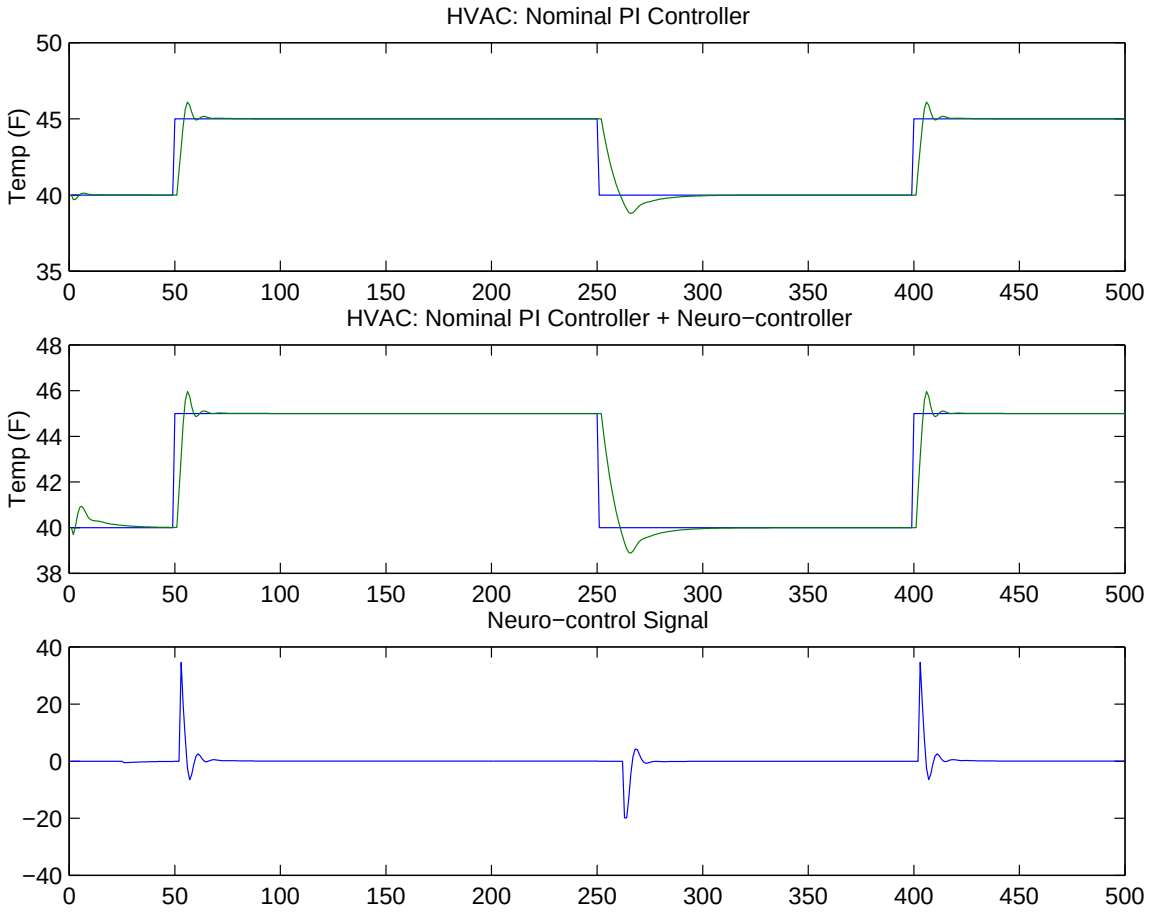


Figure 6.35: HVAC Step Response

component of the existing nominal PI controller. The second agent operates on the tracking error and the three internal state variables of the nonlinear model: F_w, T_{ao}, and T_{wo} [5].

To arrive at the ideal size neural network (number of hidden layers) we test several different configurations and find the following architectures to be the best in terms of learning and control performance. Specifically, ten hidden units are used in the actor net for each neural network. We increase the number of learning trails to two million with small learning rates. The first neuro-controller, with only the tracking error and bias as inputs, uses a 10x10 table look-up for the critic net. The second neuro-controller, with a total of four inputs, uses a four dimensional table with a resolution of 10 in each dimension (10x10x10x10).

The sum squared tracking error of each of the controllers is shown in Table 6.5.3. The first neuro-controller, with only the tracking error as input, is not able to improve the control performance over the nominal PI controller alone. The second neuro-controller, with the full state information of the plant, is able to improve control performance over the nominal PI controller by a slim 1.5%.

Figure 6.35 shows the step response of the second neuro-controller (with full state information); it is virtually identical to the step response of the nominal PI controller in Figure 6.35. This neuro-

[5]For these experiments we hold the ambient, external state variables constant

	sum squared tracking error
PI Controller	$5.26\text{x}10^{-1}$
Neuro-Controller 1	$5.28\text{x}10^{-1}$
Neuro-Controller 2	$5.18\text{x}10^{-1}$

Table 6.1: Tracking Performance

controller learns to produce no output control signal for most cases. Only during a step transition in the reference input, the neuro-controller outputs a very small control signal to be added to the PI control signal. The neuro-controller output is shown in Figure 6.35.

There are several important aspects of this case study worthy of brief discussion. First, the nominal PI controller provides fairly good tracking performance already. PI controllers have been used in control applications for much of the recent past; consequently, there are many excellent "tuning laws" available to achieve relatively good control performance. Again, due to system dynamics, we cannot achieve "perfect" control defined as a zero tracking error. Because the PI controller is near optimal, there is little room for the reinforcement learning neuro-controller.

Second, the first neuro-controller with only the tracking error as input does not improve control performance over the PI controller alone. We should expect this since the neuro-controller is essentially attempting to find a better proportional component for the PI controller; by acting only on the tracking error, the neuro-controller is augmenting the *proportional* term in the PI controller. We expect the PI tuning laws to identify the nearly optimal proportional term already. Hence, there really is no expectation for control performance improvement by the first neuro-controller acting on the proportional term alone.

Third, this case scenario is somewhat of an apples and oranges comparison. As in the distillation column case study, we should augment a robust controller with our stable reinforcement learning agent. Instead, here we attempt to augment the PI controller. Although this PI controller does appear to implement stable control, we are not mathematically guaranteed of the controller's stability properties as is the case with a nominal robust controller. This case study nicely illustrates this fundamentally different approach to control. The PI controller is tuned for optimal performance at the expense of stability guarantees. Typically, the control designer will then "back off" the aggressiveness of the PI controller; this usually results in a stable control scheme but we are still not guaranteed of this result. In robust control, we start with a mathematical guarantee of stability and then attempt to find the best controller. The stable reinforcement learning algorithm is an attempt to improve control performance over an already robust controller, not a non-robust controller like this PI controller.

As a fourth point, we note that the second neuro-controller applied to the heating coil uses additional state information that is not available to the PI controller. By exploiting this extra information, the neuro-controller may be able to implement better control performance. However, the addition of more state variables to a neuro-controller might not always be the best solution; it may be the case that a better performing controller can be found by using fewer state variables. The reason for this counter-intuitive relationship is that the added state information increases the complexity of the feedback loop which, in turn, allows more possibilities for unstable control. This may limit the size of neural network weights in order to guarantee control. Essentially, by using fewer state variables we may have less instability to deal with and hence have greater flexibility in the neuro-controller. This issue is revisited in the concluding remarks of Chapter 7.

As a concluding remark on this case study, we might ask ourselves the question, if the PI controller provides excellent control performance, then why are we interested in applying our robust neuro-control scheme to this task? The primary reason involves the difference between the real physical plant and the plant model. The physical plant will have different (and unknown) dynamics from the plant model. We still "tune" a PI controller for the physical plant, but we expect the performance of this controller to be substantially less than the performance of the PI controller on the plant model. Essentially, there is likely to be more room for improved control performance when the physical plant is involved. Our research group will test this hypothesis when the heating coil laboratory

construction is complete. The other crucial distinction, which is mentioned above, is that the PI controller has no mathematical guarantee of stable behavior.

Chapter 7

Concluding Remarks

7.1 Summary of Dissertation

The primary objective of this dissertation is a theoretical result in which we combine reinforcement learning and robust control to implement a learning neuro-controller guaranteed to provide stable control. We discuss how robust control overcomes stability and performance problems in optimal control which arise due to differences in plant models and physical plants. However, robust control is often overly conservative and thus sacrifices some performance. Neuro-controllers are frequently able to achieve better control than robust designs, because they have nonlinear components and are adaptable on-line. However, neuro-control is not practical for real implementation, because the difficult dynamic analysis is intractable and stability cannot be assured.

We develop a *static stability* test to determine whether a neural network controller, with a specific fixed set of weights, implements a stable control system. While a few previous research efforts have achieved similar results to the static stability test, we also develop a *dynamic stability* test in which the neuro-controller is stable even while the neural network weights are changing during the learning process. We also prove the correctness of both the static and dynamic stability tests.

A secondary objective of this dissertation is to demonstrate that the theoretical results concerning neuro-control stability are practical to implement in real control situations; the implementation of our stable neuro-controller does not violate any of the assumptions in the proofs of static and dynamic stability. The dynamic stability theorem leads directly to the stable reinforcement learning algorithm. Our algorithm is essentially a repetition of two phases. In the stability phase, we use μ-analysis or IQC-analysis to compute the largest amount of weight uncertainty the neuro-controller can tolerate without being unstable. We then use the weight uncertainty in the reinforcement learning phase as a restricted region in which to change the neural network weights.

A non-trivial aspect of our second objective is to develop a suitable learning agent architecture. In this development, we rationalize our choice of the reinforcement learning algorithm, because it is well suited to the type of information available in the control environment. It performs the trial-and-error approach to discovering better controllers, and it naturally optimizes our performance criteria over time. We also design a high-level architecture based upon the actor-critic design in early reinforcement learning. This dual network approach allows the control agent to operate both like a reinforcement learner and also a controller. We address neuro-dynamic difficulties peculiar to our control situation; we solve these problems by selecting a low-level architecture with a two-layer, feed forward, neural network as the actor, and a discrete, local, table look-up network as the critic.

We apply our agent and stable reinforcement learning algorithm to four case studies. The first two case studies, a first-order task, and a second-order task, are relatively simple control problems. However, their simplicity permits a detailed examination of how the stable reinforcement learning algorithm operates. We then apply the agent to a challenging distillation column control task. In this task we first see how robust control greatly improves upon the standard optimal control techniques. We then apply the stable reinforcement learning agent to the same task and improve the tracking performance by 15% over the robust controller alone while still maintaining stability. We also apply

our agent to an HVAC model. We use this case study as an example of where our stable learning agent might not perform better than other techniques which have no stability guarantees.

In spite of the success we demonstrate here, the stable, reinforcement learning controller is not without some drawbacks. First, more realistic control tasks with larger state spaces require correspondingly larger neural networks inside the controller. This increases the complexity of the neuro-controller and also increases the amount of training time required of the networks. For the simulated HVAC and distillation column tasks, the training requires significant time on high speed computers. In real life, the training time on a physical system could be prohibitively expensive as the system must be driven through all of its dynamics multiple times. Second, the robust neuro-controller may not provide control performance which is better than other "easier" design methods. This is likely to be the case in situations where the physical plant and plant model closely match each other or cases in which differences between the model and plant do not greatly affect the dynamics. The distillation column is specifically chosen as an example, because small differences between the plant and the model result in huge differences in dynamic responses; this is the ideal situation for our application. These problems and others are addressed in more detail in the remainder of this chapter. We discuss possible ways to overcome some of these problems and to more fully understand the limitations of the neuro-controller by introducing directions in future work.

7.2 Future Work with μ and IQC

In this dissertation, we use μ-analysis and IQC-analysis as tools to compute the stability of a system containing a neural network which is recast as an LTI block and an uncertainty block. Essentially, we use μ and IQC as a litmus tests for stability – either the system is stable or it is not stable. In addition to the binary indication (stable/not stable) we receive a little more information in the case of μ-analysis. This stability analysis tool produces a number which gives us an approximate idea of how stable or unstable the system is [1]. It is critical to note that *no additional* information is produced as a result of μ or IQC. A very promising direction of future research is to investigate both μ-analysis and IQC-analysis to see if additional stability information is available to assist in selecting a neuro-controller. In this section we look at a few of these concepts.

We examine a class of issues which we refer to as neural network balance issues. Consider the standard two-layer, feed forward neural network that we employ as the actor (controller). It is critical to realize that there is not a one-to-one mapping between neural networks and output functions. Two networks of exactly the same dimensions can have different weight values and still produce exactly the same output function. Similarly, networks can have a different number of hidden units and also still produce the same output function. In some circumstances the neural network functions are approximately identical; in other cases, the output functions are exactly equivalent. We use the term *neural network balance* to refer to the fact that we can shuffle, or re-balance, the weights in a neural network to achieve the same output function.

Given the fact that we can achieve the same neural network output function with different neural networks of both different sizes and the same size, we need to examine possible motivations for selecting one neural network over another. From a control standpoint, the only critical aspect of the neural network is its output function. Two different networks which compute the same output function are equivalent from the control perspective; there is no reason to choose one network over another. However, from a stability standpoint there might be substantial differences in the stability properties of two neural networks producing the same output function. The network which is more stable is the more desirable choice.

We have conducted a preliminary experiment along these lines. Consider the two-layer, feed forward, *tanh* hidden layer, neural network labelled as Network A in Figure 7.1. Network A has one input, one hidden unit, and one output; $W_{1\text{x}1} = w$ is the single input weight and $V_{1\text{x}1} = v$ is the single output weight. We convert Network A into an LTI block with uncertainty and then place the

[1] With $\mu = 1$ we are just barely unstable. The smaller μ is below 1, the more stable the system. The more μ is greater than 1, the more unstable the system.

converted network into a small feedback control system. We compute the stability of the feedback system by finding the μ value for the system.

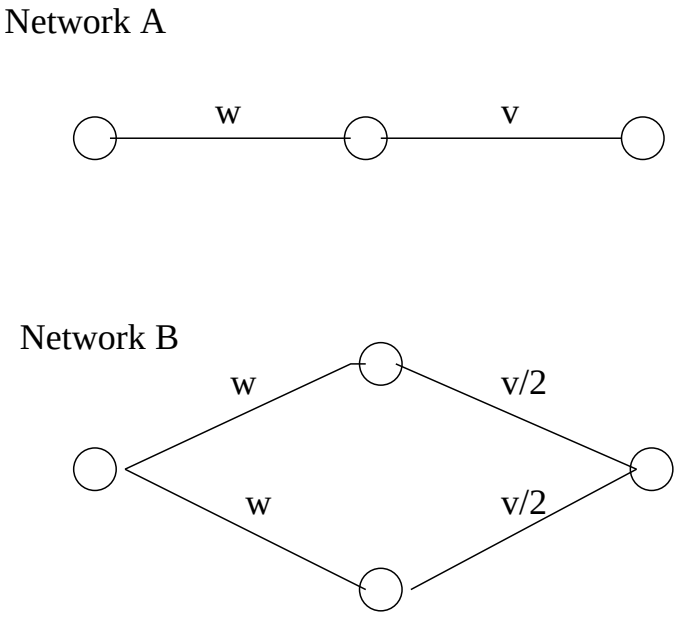


Figure 7.1: Balancing Networks

We construct a second neural network, Network B, also shown in Figure 7.1. This network is identical to Network A except that it has two hidden units instead of one. The input weight for both hidden units is w. This is the same input weight as used in Network A. Thus all three hidden units (one hidden unit in Network A and two hidden units in Network B) all implement the same function, or feature, at the hidden layer. The two output weights for Network B are both set to $\frac{v}{2}$. It is not hard to see that Network A and Network B produce exactly the same output function; network B simply uses two hidden units instead of Network A's single hidden unit. We have taken the hidden unit in Network A and split it to form Network B's two hidden units. We then convert Network B into LTI plus uncertainty and compute its μ number. It turns out that the μ value for both networks is identical. In fact, we repeat this experiment by forming additional networks with five and ten hidden units with output weights of $\frac{v}{5}$ and $\frac{v}{10}$ respectively. These networks also produce the same μ value. All these networks are equivalent both in terms of their output functions and in there stability properties.

The results of the above mini-experiment are not surprising; all the "re-balancing" was performed within the LTI block and had no effect on the uncertainty. We do not have to know much about μ-analysis or IQC analysis to be able to predict that splitting the output weights will have no effect on the stability of a neural network in the feedback system. However, a more difficult question is to ask what happens to the stability analysis when re-balancing is done at the input layer. There are at least two immediate variants of this case. We could have two neural networks with the same number of hidden units but different input/output weight values. The second case involves networks with different numbers of hidden units (and different input/output weight values). In both these cases, we could formulate neural networks that produce identical or nearly identical output functions but might possess widely varying μ values when incorporated as part of a feedback system.

These are key issues certainly worthy of investigation. If we could discover a relationship between stability and network weight balancing, then we could select networks with fewer (or more) hidden layers in an attempt to produce neuro-controllers with higher stability margins. We might also discover a neural network weight re-balancing algorithm to systematically adjust the network weights in an attempt to keep the same output function but increase the stability margin.

A separate but related issue of weight balancing is computing a μ-gradient. The back propagation algorithm used to train the weights in the neural network performs a gradient descent search through the weight space. The gradient is provided by the value function in the critic network. This gradient determines the direction of movement through the weight space. There must certainly also exist a *stability gradient* in the weight space. There will be directions of weight updates which increase the stability of the system and directions of weight updates which decrease the stability of the neuro-control system. By combining the two gradients, we can train the neural network in a direction that both improves control performance (value function gradient) and maintains stability (stability gradient).

For the actor network we select the two-layer, feed forward neural network. This network is both continuous and has hidden units which extend globally over the input space. We opted for this network architecture in the actor net, because these continuous/global properties are desirable for both computing good control functions and because there are amenable to the stability analysis. However, other neural network architectures are also possible. Though they are likely to introduce a set of extra complications, these other network architectures might have different stability properties. Specifically, we believe it might be advantageous to use local, continuous units for the network such as those in a radial basis function network with gaussian units. Because the units do not extend across large regions of the input space, stability complications might be localized to some parts of the inputs space; other "safer" and more stable regions might be able to experience additional training without adversely affecting the stability of the overall system. Certainly this issue is worthy of further investigation.

Clearly the relationship between the neural network function, the neural network structure, and the computation of μ (and IQC) opens a veritable Pandora's Box of unanswered questions. There is ample opportunity to extend our research in this direction. These questions could develop relationships and theory in new areas of neuro-control.

7.3 Future Work with Neural Network Architecture

While the primary goal of this dissertation is the establishment of the static stability theorem and the dynamic stability theorem, the secondary objective is to demonstrate that these two theorems are applicable in practical control situations. Chapter 5 outlines the design of a learning agent which is able to achieve this latter objective. During the agent's development, we encountered a number of technical problems for which we were able to find a solution. While these solutions overcame our difficulties, perhaps these were not the best solutions. We return to address a few of these difficulties and point out other, yet unexplored, alternatives for the design of the stable reinforcement learning agent.

A number of these alternatives can be categorized under the heading of "issues with neural network size selection". For each specific control problem, we select an appropriate size for the actor network (and hence for the critic network). Here, we use network size to refer to both the number of hidden units and the number of input inputs. First we address those size issues concerning only the number of hidden units. There are valid task-specific reasons to increase the number of hidden units and valid neural network related reasons to decrease the number of hidden units. The primary reason to increase the number of hidden units in the actor network is to incorporate more function approximation resources in the network. To some extent, the complexity of the control function that we desire to learn with the actor network dictates how many hidden *tanh* units are required within the hidden layer. A more complicated control function will require more hidden units to achieve the same level of approximation accuracy. There are also valid reasons to keep the number of hidden units small. Empirical evidence indicates that a network with more hidden units requires more training time to converge. These issues certainly are not new to this dissertation but they do play a key role in selecting a good neuro-controller. A number of good references indicate active research in this area of neural networks. See [Vidyasagar, 1997; Vapnik, 1995; Vapnik, 1998] for a review of current research in network size selection, function approximation, and training time.

In addition to the network size issues discussed above, we have additional choices to make con-

cerning neural network design in control situations. These additional choices stem from the large number of system state variables in many control systems. At any moment in time, each control system is completely identified by the state information in the system. This state information includes state variables of the plant, state variables of the nominal controller, reference input signals, external disturbance input signals, and possibly other sources of state information. The total number of state variables in a particular system can grow to be quite large. The distillation column process has a total of 12 state variables. (Actually, there were 22 initially until we reduced the robust controller from 18th order to 8th order using the sysbal Matlab command).

The goal of the actor network (controller) is to use the state information to produce a control signal that is both stable and results in good tracking performance. Minimally, we use the current tracking error as an input to the neuro-controller; this is the case with the example tasks in the first two case studies and one of the networks in the HVAC task. However, the other state variables in the system likely contain information that can be used to make better control decisions. We could include a subset of the state variables as additional inputs to the actor network. However, all state variables are not created equal. From the neuro-controller's perspective of making sound control decisions, some state variables contain more information than others. Neural networks designers face a choice of which of these state variables to use in the neural network. One possibility is to use all state variables. This has the advantage of ensuring the network has all available information; but, this approach is saddled with the major drawback of increased training time. Those state variables that do not possess control-decision information will act as noise in the input training data. This is especially true if only a few of the state variables contain a majority of the control information. Not only does the network have to filter through the noisy input channels to discover those rich in control information, but the network will also be unnecessarily larger in order to accommodate the extra inputs. The other approach is to choose a subset of the state variables for input to the actor network. The challenge is to find those state variables containing the information most relevant to making improved control decisions.

7.4 Future Work with HVAC

Our results in applying stable neuro-control to HVAC are preliminary. As discussed in Section 6.5, we are in the process of constructing a physical HVAC plant. At the time of this dissertation work, only the HVAC plant model is available for testing. The PI controller designed for this model already achieves excellent performance. Thus, the neuro-controller is unable to improve upon this performance significantly. When construction of the physical plant is complete, we can then develop a better HVAC heating coil model by performing empirical step-response studies on the physical plant. Then an LTI model can be developed from these studies. At that point, we can apply modern optimal control design techniques as well as robust control techniques to the plant model. Upon testing them on the physical plant, we will be able to ascertain whether their performance is acceptable and whether a neuro-controller, trained on the physical plant, is able to further improve control performance.

Appendix A

Stability Analysis Tools

A.1 μ-analysis

The commands here indicate how we used the μ toolbox. Further details are available in the code listings in Appendix B.

$$[\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}] = \mathsf{dlinmod}('\mathsf{task1\_mu1}', 0.01); \tag{A.1}$$

$$\mathsf{sys} = \mathsf{pck}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}); \tag{A.2}$$

dlinmod is the command which actually converts the diagram into an LFT system where 'task1_mu1' is the name of the diagram file and 0.01 is the sampling period of the discrete-time plant. pck is a Matlab command which places the LFT into a more convenient format; the LFT is stored in a variable called sys.

Next we will perform μ-analysis on the LFT system. The following Matlab commands compute μ for the system:

$$\mathsf{om} = \mathsf{logspace}(-2, 2, 100); \tag{A.3}$$

$$\mathsf{blk} = [1\ 1]; \tag{A.4}$$

$$\mathsf{sysf} = \mathsf{frsp}(\mathsf{sys}, \mathsf{blk}, 0.01); \tag{A.5}$$

$$\mathsf{bnds} = \mathsf{mu}(\mathsf{sysf}, \mathsf{blk}); \tag{A.6}$$

where om is the frequency range that we will compute μ over. blk is the format of the structed uncertainty; in this case the uncertainty is a 1x1 block because we have only one hidden unit in the neural network. frsp computes the frequency response of the system and stores it in a vector sysf. Finally mu is the command which computes μ for this frequency response [Balas et al., 1996].

A.2 IQC-analysis

IQC-analysis is straight forward as all the work is done in constructing the Simulink diagram. Once the Simulink diagram is complete, simply run *iqc_gui* with the diagram name supplied as an argument. The IQC command reads the Simulink diagram from the disk; changes made to the open file will not be incorporated unless those changes are first saved.

Appendix B

Software Listing

This appendix contains most of the code used to generate the results obtained for this dissertation. All code is written in the Matlab programming language for compatability with the μ and IQC toolboxes.

runit.m

This command simulates the distillation column task with the nominal controller only.

```
function [x,y,u,e] = runit(r,AK,BK,CK,DK)
%[x,y,u,e] = runit(r,AK,BK,CK,DK)
% r is a fixed reference input

[kk,sr] = size(r);
[k1,k1] = size(AK);

A = [0.99867, 0; 0, 0.99867];
B = [-1.01315, 0.99700; -1.24855, 1.26471];
C = [-0.11547, 0; 0, -0.11547];
D = [0, 0; 0, 0];

e = 0;
x = zeros(2,sr);
k = zeros(k1,sr);
y(:,1) = C * x(:,1);

start = 300;


for i = 1:sr-1
   err = r(:,i) - y(:,i);
   k(:,i+1) = AK*k(:,i) + BK*err;
   u(:,i) = CK*k(:,i) + DK*err;
   u(1,i) = u(1,i) * 1.2;
   u(2,i) = u(2,i) * 0.8;
```

```
   x(:,i+1) = A*x(:,i) + B*u(:,i);
   y(:,i+1) = C*x(:,i) + D*u(:,i);
   err = r(:,i) - y(:,i);
   if ( i > start )
      e = e + sum(abs(err));
   end;
end;

u(:,sr) = u(:,sr-1);
err = r(:,sr) - y(:,sr);
e = e + sum(abs(err));
```

runitnn.m

This command simulates the distillation column task with the neuro-controller and the nominal controller.

```
function [x,y,u,e] = runitnn(W,V,r,AK,BK,CK,DK)
%[x,y,u,e] = runit(W,V,r,AK,BK,CK,DK)
% r is a fixed reference input

[kk,sr] = size(r);
[k1,k1] = size(AK);

A = [0.99867, 0; 0, 0.99867];
B = [-1.01315, 0.99700; -1.24855, 1.26471];
C = [-0.11547, 0; 0, -0.11547];
D = [0, 0; 0, 0];

e = 0;
x = zeros(2,sr);
k = zeros(k1,sr);
y(:,1) = C * x(:,1);

start = 300;

for i = 1:sr-1
   err = r(:,i) - y(:,i);
   k(:,i+1) = AK*k(:,i) + BK*err;
   u(:,i) = CK*k(:,i) + DK*err;

   c(1,1) = 1;
   c(2,1) = err(1);
   c(3,1) = err(2);
   [un, v] = feedf(c,W,V);

   u(:,i) = u(:,i) + un;

   u(1,i) = u(1,i) * 1.2;
   u(2,i) = u(2,i) * 0.8;

   x(:,i+1) = A*x(:,i) + B*u(:,i);
   y(:,i+1) = C*x(:,i) + D*u(:,i);
   err = r(:,i) - y(:,i);
   if ( i > start )
      e = e + sum(abs(err));
   end;
end;

u(:,sr) = u(:,sr-1);
err = r(:,sr) - y(:,sr);
e = e + sum(abs(err));
```

both.m

This command trains the neuro-controller (actor net and critic net) without the stability constraints.

```
function [Q,W,V] = both(Q,q,W,V,N,a1,a2,AK,BK,CK,DK)
%[Q,W,V] = both(Q,q,W,V,N,a1,a2,AK,BK,CK,DK

A = [0.99867, 0; 0, 0.99867];
B = [-1.01315, 0.99700; -1.24855, 1.26471];
C = [-0.11547, 0; 0, -0.11547];
D = [0, 0; 0, 0];

sum_err = 0;
t = 0;
r = [0; 0];
x = [0; 0];
y = [0; 0];
sk = length(AK);
k = zeros(sk,1);

for i = 1:N

   %change reference signal
   rold = r;
   if ( mod(i,2000) == 1 )
      if ( rand < 0.5 )
         r(1,1) = 1 - r(1,1);
      else
         r(2,1) = 1 - r(2,1);
      end;
   end;

   err = r-y;
   %sum_err = sum_err + sum(abs(err));

   %compute u
   k = AK*k + BK*err;
   u = CK*k + DK*err;

   %compute un
   c(1,1) = 1;
   c(2,1) = err(1);
   c(3,1) = err(2);
   [un, v] = feedf(c,W,V);

   %random part
   if (rand < 0.1 )   urand = randn(2,1) .* 0.1;
   else         urand = [0; 0];     end;
   %urand = 0;
   un = un + urand;
   u = un;

   u(1,1) = u(1,1) * 1.2;
   u(2,1) = u(2,1) * 0.8;
```

```
   %remember old values for use in TD backprop
   if ( t > 0 )
      qvalold = qval;
      activold = activ;
   end

   %compute Q
   b(1,1) = err(1);
   b(2,1) = err(2);
   b(3,1) = un(1);
   b(4,1) = un(2);
   [qval,activ] = compute(Q,q,b);

   %TD backprop
   if ( t > 0 & sum(abs(rold - r)) == 0 )
      tar = 0.95 * qval + sum(abs(err));
      Q = learnQ(tar,qvalold,activold,Q,a1);
   end

   %compute minimum action
   yp = delta2(Q,q,b,3,4);
   [W,V] = backprop(c,v,un,yp,a2,W,V);

   %update state
   t = t + 0.1;
   x = A*x + B*u;
   y = C*x + D*u;

   fprintf([int2str(i) '.']);
   if ( mod(i,20) == 0 ) fprintf('\n'); end;

end; %outer for
```

muwv.m

This command trains the neuro-controller (actor net and critic net) with the stability constraints activated.

```
function [Q,W,V,Wmax,Wmin,Wt,Vmax,Vmin,Vt] = muwv(Q,q,W,V,N,a1,a2,muit,trace)
%[Q,W,V,Wmax,Wmin,Wt,Vmax,Vmin,Vt] = muwv(Q,q,W,V,N,a1,a2,muit,trace)


A = [1.0 0.05; -0.05 0.9];
B = [0; 1.0];
Kp = 0.01; Ki = 0.001; %0.05;

sum_err = 0;
t = 0;
r = (rand-0.5)*2; urand = 0;
x = [0; 0];


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Before we learn, make sure [W,V] are in mu bounds
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
maxmu = statmu(W,V);
if ( maxmu > 1.0 )
   disp 'Warning: mu exceeds 1 for input matrices W,V'
   disp 'Learning Halted!'
   return;
end;

if ( trace )
[h,n] = size(W);
Wt = zeros(h,n,N);
   Vt = zeros(1,h,N);
else
   Wt = 0;
   Vt = 0;
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Outer Loop iterates over dW and dV -- used mu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for kk = 1:muit

[dW,dV] = dynamu(W,V);
dW = W.*0 + 1; dV = V.*0 + 1;
Wmax = W + dW,W,  Wmin = W - dW,
Vmax = V + dV,    V,  Vmin = V - dV,

j = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Inner loop trains either till completion of iterations or until
% we hit the mu-specified boundary
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while ( j < N & gtm(W,Wmin) & gtm(Wmax,W) & gtm(V,Vmin) & gtm(Vmax,V))
```

```
   j = j + 1;

if ( trace )
    Wt(:,:,j) = W;
      Vt(:,:,j) = V;
   end;

   %change reference signal
   rold = r;
   if ( rand < 0.01 ) r = (rand-0.5) * 2.0; end;
   err = (r-x(1));

   %compute upi
   sum_err = sum_err + err;
   upi = Kp * err + Ki * sum_err;

   %compute un
   c(1,1) = err;
   [un, v] = feedf(c,W,V);

   %random part
   if (rand < 0.1 )   urand = randn * 0.05;
   else         urand = 0;     end;
   un = un + urand;
   u = upi + un;

   %remember old values for use in TD backprop
   if ( t > 0 )   yold = y;    aold = a;    end

   %compute Q
   b(1,1) = err; b(2,1) = un;
   [y,a] = compute(Q,q,b);

   %TD backprop
   if ( t > 0 & rold == r )
      tar = 0.9 * y + abs(err);
      Q = learnQ(tar,yold,aold,Q,a1);
   end

   %compute minimum action
   yp = delta(Q,q,b,2);
   Wgood = W;
   Vgood = V;
   [W,V] = backprop(c,v,un,yp,a2,W,V);

   %update state
   t = t + 0.01; x = A*x + B*u;

   fprintf([int2str(j) '.']);
   if ( mod(j,20) == 0 ) fprintf('\n'); end;

end; %outer for

W = Wgood;
```

```
V = Vgood;
end;
```

setsig.m

This command initializes a sigmoid neural network.

```
function [W,V] = setSIG(Nin,Nhid,Nout);
%[W,V] = setSIG(Nin,Nhid,Nout);
%remember to add 1 at input and hidden for bias

W = (rand(Nhid,Nin)-0.5);
V = (rand(Nout,Nhid)-0.5) .* 0.1;
```

setcmac.m
This command initializes a CMAC neural network.

```
function [Q,q] = setCMAC(n,t,mins,sizes)
%[Q,q] = setCMAC(n,t,mins,sizes)
%
%Initializes a 4D CMAC network with t tilings of nxn grids
% mins    (4,1) sized vector of minimums per dimension
% sizes   (4,1) sized vector of sizes per dimension
%DIM = 2;
%DIM = 3;
DIM = 4;
%DIM = 6;

%Q = zeros(n,n,n,n,n,n,t);
Q = zeros(n,n,n,n,t);
%Q = zeros(n,n,n,t);
%Q = zeros(n,n,t);
q = zeros(n,DIM,t);

for k = 1:DIM
   incr(k) = sizes(k) / (n);
   off(k) = incr(k) / t;
end;

for i = 1:t
  for j = 1:n
for k = 1:DIM
      q(j,k,i) = (j-1) * incr(k) + (i-1) * off(k) + mins(k);
    end;
  end;
end;
```

backprop.m

This command implements the backpropagation learning algorithm for sigmoid networks.

```
function [W, V] = backprop(x,h,y,yp,alpha,W,V);
%[W, V] = backprop(x,h,y,yp,alpha,W,V);
% Inputs:  x input vector      (n,1)
%          h hidden tanh       (h,1)
%          y output vector     (o,1)
%          yp output target    (o,1)
%          alpha learning rate
%          W input weights     (h,n)
%          V output weights    (o,h)
% Outputs  W, V new

d2 = yp - y;
dV = alpha .* d2 * h';

d1 = (1 - h.*h) .* (V' * d2);
dW = alpha .* d1 * x';

V = V + dV;
W = W + dW;
```

feedf.m

This command performs the feedforward operation on a sigmoid neural network.

```
function [y, h] = feedf(x,W,V)
%[y, h] = feedf(x,W,V)
% Inputs:   x is input vector          (n,1)
%           W is input side weights    (h,n)
%           V is output side weights   (o,h)
% Outputs:  y is output vector         (o,1)
%           h is tanh hidden layer     (h,1)
% Note: h is required for back prop training

h = W * x;
h = tanh(h);
y = V * h;
```

activate.m

This command computes the activation for a CMAC network with particular inputs.

```
function a = activate(q,x);
%a = activate(q,x);
%
%x is the (2,1) input vector
%q is the CMAC resolution vector
%a is the (t,2) activation vector

[n,pp,t] = size(q);
%DIM = 2;
%DIM = 3;
DIM = 4;
%DIM = 6;

for i = 1:t
   %find x dimension by searching backward
   for k = 1:DIM
   j = n;
     while (j > 1) & ( x(k) < q(j,k,i) )
        j = j - 1;
   end;
     a(i,k) = j;
   end;
end;
```

compute.m

This command computes the output of a CMAC neural network.

```
function [y,a] = compute(Q,q,x)
%[y,a] = compute(Q,q,x)

[n,pp,t] = size(q);
a = activate(q,x);
y = 0;
for i = 1:t
%   y = y + Q( a(i,1), a(i,2), a(i,3), a(i,4), a(i,5), a(i,6), i);
    y = y + Q( a(i,1), a(i,2), a(i,3), a(i,4), i);
%   y = y + Q( a(i,1), a(i,2), a(i,3), i);
%   y = y + Q( a(i,1), a(i,2), i);
end;
```

learnq.m

This command trains the CMAC network using the TD-error.

```
function Q = learnq(tar,cur,activ,Q,alpha)
%Q = learnq(tar,cur,activ,Q,alpha)

%[t1,n,t]       = size(Q);
%[t1,t2,n,t]    = size(Q);
 [t1,t2,t3,n,t] = size(Q);
%[t1,t2,t3,t4,t5,n,t] = size(Q);

diff = (tar - cur) * alpha / t;

for i = 1:t
   x1 = activ(i,1);
   x2 = activ(i,2);
   x3 = activ(i,3);
   x4 = activ(i,4);
%  x5 = activ(i,5);
%  x6 = activ(i,6);
%  Q(x1,x2,x3,x4,x5,x6,i) = Q(x1,x2,x3,x4,x5,x6,i) + diff;
   Q(x1,x2,x3,x4,i) = Q(x1,x2,x3,x4,i) + diff;
%  Q(x1,x2,x3,i) = Q(x1,x2,x3,i) + diff;
%  Q(x1,x2,i) = Q(x1,x2,i) + diff;
end;
```

delta.m

This command computes the double gradient of the CMAC network.

```
function d = delta(Q,q,x,k)

RES = 20;
[n,dim,t] = size(q);
minn = q(1,k,1);
maxx = q(n,k,1) + q(n,k,1) - q(n-1,k,1);
incr = ( maxx - minn ) / 20;

mina = x(k) - incr;
maxa = x(k) + incr;
if ( mina < minn )
   mina = minn;
end;
if ( maxa > maxx )
   maxa = maxx;
end;

us = linspace(mina,maxa,RES);

for i = 1:RES
   x(k) = us(i);
   v(i) = compute(Q,q,x);
end;

[mv, mp] = min(v);
d = us(mp);
```

diagfit.m

This command computes diagonalization matrices required for the μ Simulink diagrams.

```
function [A,B,T] = diagFit(m,n)
%[A,B,T] = diagFit(m,n)
%      or
%[A,B,T] = diagFit(X)
%
% m,n is the dimension of a matrix X
% let r = m*n
% T is the r by r diagonal matrix that has each
%   entry of X in it (across first, then down)
% example: X = | 3  7 |   T = | 3  0  0  0 |
%  m=2         | 4 -1 |       | 0  7  0  0 |
%  n=2                        | 0  0  4  0 |
%                             | 0  0  0 -1 |
%
% diagFit computesmatrices A and B such that A*T*B = X.
% Produces test matrix T = diag(1..r) for testing.


if (nargin < 2)
   [x,n] = size(m);
   clear m;
   m = x;
   clear x;
end;

r = m * n;

A = zeros(m,r);
B = zeros(r,n);
for i = 1:r
   ka = floor( (i-1)/n ) + 1;
   kb = mod(i-1,n) + 1;
   A(ka,i) = 1;
   B(i,kb) = 1;
end;

T = diag(1:r);
```

statmu.m

This command computes the static stability test.

```
function [maxmu, bnds] = statmu(W,V)
%[maxmu, bnds] = compmu(W,V)
%Computes static mu
%Remember to change variables like
% T sampling rate (0.01)
% om logspace (-2,2,100)
% filename task1_mu


nhid = length(V);
blk = [nhid, 0];
om = logspace(-2,2,100);
set_param('task1_mu/W','K',mat2str(W));
set_param('task1_mu/V','K',mat2str(V));
set_param('task1_mu/eyeV','K',mat2str(eye(nhid)));

disp 'Computing static mu';
[a,b,c,d] = dlinmod('task1_mu',0.01);
sys = pck(a,b,c,d);
sysf = frsp(sys,om,0.01);
bnds = mu(sysf,blk,'s');
maxmu = max(bnds(:,1));
```

dynamu.m

This command computes the dynamics stability test by computing the allowable neural network weight uncertainty.

```
function [dW,dV] = dynamu(W,V)
%[dW,dV] = dynamu(W,V)
%Remember to change variables like
% T     sampling rate (0.01)
% om   logspace (-2,2,100)
% filename task1_mu

[h,n] = size(W);
[jj,h] = size(V);
mW = n * h;
mV = h * jj;

dW = eye(mW);
dV = eye(mV);

sumW = sum(sum(abs(W)));
sumV = sum(sum(abs(V)));
sumT = sumW + sumV;

for i = 1:h
   for j = 1:n
k = (i-1)*n + j;
      dW(k,k) = abs(W(i,j) / sumT);
      dV(i,i) = abs(V(i) / sumT);
   end;
end;

iW = dW;
iV = dV;

minf = 1;
maxf = 1;

blk = [h, 0];
blk = [blk; ones(mW + mV,2)];
%blk(:,1) = blk(:,1) .* -1;
om = logspace(-2,2,100);
[WA,WB] = diagfit(W);
[VA,VB] = diagfit(V);

%set parameters in simulink model 'task1_mu3'
set_param('task1_mu3/W','K',mat2str(W));
set_param('task1_mu3/V','K',mat2str(V));
set_param('task1_mu3/WA','K',mat2str(WA));
set_param('task1_mu3/WB','K',mat2str(WB));
set_param('task1_mu3/VA','K',mat2str(VA));
set_param('task1_mu3/VB','K',mat2str(VB));
set_param('task1_mu3/eyeV','K',mat2str(eye(h)));
set_param('task1_mu3/dW','K',mat2str(dW));
set_param('task1_mu3/dV','K',mat2str(dV));
```

```
%Compute initial mu
set_param('task1_mu3/dW','K',mat2str(dW));
set_param('task1_mu3/dV','K',mat2str(dV));
[a,b,c,d] = dlinmod('task1_mu3',0.01);
sys = pck(a,b,c,d);
sysf = frsp(sys,om,0.01);
disp 'Computing mu';
bnds = mu(sysf,blk,'s');
maxmu = max(bnds(:,1));
s = sprintf('mu = %f for scale factor %f',maxmu,minf);
disp(s);

if ( maxmu < 1 )
   while ( maxmu < 1 )
      temp = maxmu;
      minf = maxf;
      maxf = maxf * 2;
      dW = iW .* maxf;
      dV = iV .* maxf;
    set_param('task1_mu3/dW','K',mat2str(dW));
set_param('task1_mu3/dV','K',mat2str(dV));
[a,b,c,d] = dlinmod('task1_mu3',0.01);
sys = pck(a,b,c,d);
sysf = frsp(sys,om,0.01);
      disp 'Computing mu';
bnds = mu(sysf,blk,'s');
      maxmu = max(bnds(:,1));
      s = sprintf('mu = %f for scale factor %f',maxmu,maxf);
      disp(s);
   end;
   maxmu = temp;
else
   while ( maxmu > 1 )
      if (minf < 0.01)
         disp 'Warning: dynamu cannot find dW,dV with mu < 1'
         disp 'Halt Learning'
         dW = iW .* 0;
         dV = iV .* 0;
         return
      end;
      maxf = minf;
      minf = minf * 0.5;
      dW = iW .* minf;
      dV = iV .* minf;
set_param('task1_mu3/dW','K',mat2str(dW));
    set_param('task1_mu3/dV','K',mat2str(dV));
[a,b,c,d] = dlinmod('task1_mu3',0.01);
sys = pck(a,b,c,d);
sysf = frsp(sys,om,0.01);
      disp 'Computing mu';
bnds = mu(sysf,blk,'s');
      maxmu = max(bnds(:,1));
      s = sprintf('mu = %f for scale factor %f',maxmu,minf);
```

```
      disp(s);
   end;
end;

while ( maxmu < 0.95 | maxmu > 1 )

   if ( maxmu < 1 )
      safe = minf;
      minf = (maxf-safe)/2 + safe;
   else
      maxf = minf;
      minf = (maxf-safe)/2 + safe;
end;
   if (minf < 0.01)
      disp 'Warning: dynamu cannot find dW,dV with mu < 1'
      disp 'Halt Learning'
      dW = iW .* 0;
      dV = iV .* 0;
      return
   end;

   dW = iW .* minf;
   dV = iV .* minf;
set_param('task1_mu3/dW','K',mat2str(dW));
set_param('task1_mu3/dV','K',mat2str(dV));
[a,b,c,d] = dlinmod('task1_mu3',0.01);
sys = pck(a,b,c,d);
sysf = frsp(sys,om,0.01);
   disp 'Computing mu';
bnds = mu(sysf,blk,'s');
   maxmu = max(bnds(:,1));
   s = sprintf('mu = %f for scale factor %f',maxmu,minf);
   disp(s);
end;

dW = WA * dW * WB;
dV = VA * dV * VB;
```

REFERENCES

Aggarwal, J. K. and Vidyasagar, M. (1977). *Nonlinear Systems Stability Analysis*. Dowden, Hutchinson, and Ross Inc.

Anderson, C., Hittle, D., Katz, A., and Kretchmar, R. (1996). Reinforcement learning combined with pi control for the control of a heating coil. *Journal of Artificial Intelligence in Engineering*.

Anderson, C., Hittle, D., and Young, P. (1998). National science foundation grant proposal: CMS-980474.

Anderson, C. W. (1992). *Neural Networks for Control*, chapter Challenging Control Problems. Bradford.

Anderson, C. W. (1993). Q-learning with hidden-unit restarting. In *Advances in Neural Information Processing Systems 5: NIPS'93*, pages 81–88.

Astrom, K. J. and Wittenmark, B. (1973). On self-tuning regulators. *Automatica*, 9:185–189.

Balas, G. J., Doyle, J. C., Glover, K., Packard, A., and Smith, R. (1996). *μ-Analysis and Synthesis Toolbox*. The MathWorks Inc., 24 Prime Park Way Natick, MA 01760-1500, 1 edition.

Barto, A. G. (1992). *Neural Networks for Control*, chapter Connectionist Learning for Control. Bradford.

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1996). Learting to act using real-time dynamic programming. *Artificial Intelligence: Special Issue on Computational Research on Interaction and Agency*, 72(1):81–138.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846.

Bass, E. and Lee, K. Y. (1994). Robust control of nonlinear systems using norm-bounded neural networks. *IEEE International Conference on Neural Networks – Conference Proceedings*, 4:2524–2529.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

Cock, K. D., Moor, B. D., Minten, W., Brempt, W. V., and Verrelst, H. (1997). A tutorial on PID-control. Technical Report ESAT-SIST/TR 1997-08, Katholieke Universiteit Leuven.

Crites, R. H. and Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8: NIPS'96*.

Desoer, C. A. and Vidyasagar, M. (1975). *Feedback Systems: Input-Output Properties*. Academic Press Inc.

Doyle, J. C., Francis, B. A., and Tannenbaum, A. R. (1992). *Feedback Control Theory*. Macmillan Publishing Company.

Franklin, J. A. and Selfridge, O. G. (1992). *Neural Networks for Control*, chapter Some New Directions for Adaptive Control Theory in Robotics. Bradford.

Gahihet, P., Nemirovski, A., Laub, A. J., and Chilali, M. (1995). *LMI Control Toolbox*. MathWorks Inc.

Gleick, J. (1987). *Chaos: Making a New Science*. Penguin Books.

Hassoun, M. H. (1995). *Fundamentals of Artificial Neural Networks*. The MIT Press, Cambridge, MA.

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. Macmillan College Publishing Inc.

Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company.

Jordan, M. I. (1988). Sequential dependencies and systems with excess degrees of freedom. Technical Report UM-CS-88-027, University of Massachusetts.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4.

Kalkkuhl, J., Hunt, K. J., Zbikowski, R., and Dzielinski, A. (1997). *Applications of Neural Adaptive Control Technology*. World Scientific.

Kawato, M. (1992). *Computational Schemes and Neural Network Models for Formation and Control of Multijoint Arm Trajectory*, chapter Connectionist Learning for Control. Bradford.

Kohonen, T. (1997). *Self-Organizing Maps*. Springer-Verlag.

Kretchmar, R. M. and Anderson, C. W. (1997). Comparison of cmacs and radial basis functions for local function approximation in reinforcement learning. In *ICNN'97: Proceedings of the International Conference on Neural Networks*. ICNN.

Kretchmar, R. M. and Anderson, C. W. (1999). Using temporal neighborhoods to adapt function approximator in reinforcement learning. In *IWANN'99: International Workshop on Artificial Neural Networks*. IWANN.

Levin, A. U. and Narendra, K. S. (1993). Control of nonlinear dynamical systems using neural networks: Controllability and stabilization. *IEEE Transactions on Neural Networks*, 4(2):192–206.

Megretski, A., KAO, C.-Y., Jonsson, U., and Rantzer, A. (1999). *A Guide to IQCβ: Software for Robustness Analysis*. MIT / Lund Institute of Technology, http://www.mit.edu/people/ameg/home.html.

Megretski, A. and Rantzer, A. (1997a). System analysis via integral quadratic constraints. *IEEE Transactions on Automatic Control*, 42(6):819–830.

Megretski, A. and Rantzer, A. (1997b). System analysis via integral quadratic constraints: Part II. Technical Report ISRN LUTFD2/TFRT–7559–SE, Lund Institute of Technology.

Miller, W. T., Glanz, F. H., and Kraft, L. G. (1990). Cmac: An associative neural network alternative to backpropagation. *Proceedings of the IEEE*, 78:1561–1567.

Moore, A. W. (1995). The parti-game algorithm for variable resolution reinforcement learning in multi-demensional state spaces. *Machine Learning*, 21.

Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1(1):4–27.

Packard, A. and Doyle, J. (1993). The complex structured singular value. *Automatica*, 29(1):71–109.

Parks, P. C. (1966). Lyapunov redesign of model reference adaptive control systems. *IEEE Transactions on Automatic Control*, 11:362–367.

Pavlov, P. I. (1927). *Conditioned Reflexes*. Oxford University Press.

Phillips, C. L. and Harbor, R. D. (1996). *Feedback Control Systems*. Prentice Hall, 3 edition.

Royas, R. (1996). *Neural Networks: A Systematic Introduction*. Springer.

Rugh, W. J. (1996). *Linear System Theory*. Prentice-Hall Inc., 2 edition.

Rumelhart, D., Hinton, G., and Williams, R. (1986a). *Parallel Distributed Processing*, volume 1, chapter Learning internal representations by error propagation. Bradford Books.

Rumelhart, D., Hinton, G., and Williams, R. (1986b). *Parallel Distributed Processing*. Bradford Books, The MIT Press, Cambridge, MA.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3.

Sanner, R. M. and Slotine, J.-J. E. (1992). Gaussian networks for direct adaptive control. *IEEE Transactions on Neural Networks*, 3(6):837–863.

Singh, S. and Bertsekas, D. (1996). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems 8: NIPS'96*.

Skinner, B. F. (1938). *The Behavior of Organisms*. Appleton-Century.

Skogestad, S. and Postlethwaite, I. (1996). *Multivariable Feedback Control*. John Wiley and Sons.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press.

Suykens, J. and Bersini, H. (1996). Neural control theory: an overview. *Journal A*, 37(3):4–10.

Suykens, J. and Moor, B. D. (1997). NLq theory: a neural control framework with global assymptotic stability criteria. *Neural Networks*, 10(4).

Suykens, J., Moor, B. D., and Vandewalle, J. (1993a). Neural network models as linear systems with bounded uncertainty applicable to robust controller design. In *Proceedings of the International Symposium on Nonlinear Theory and its Application (Nolta'93)*.

Suykens, J., Moor, B. D., and Vandewalle, J. (1993b). Stabilizing neural controllers: A case study for swinging up a double inverted pendulum. In *Proceedings of the International Symposium on Nonlinear Theory and its Application (Nolta'93)*.

Suykens, J. A. K., Moor, B. D., and Vandewalle, J. (1997). Robust nlq neural control theory. In *Proceedings of the International Conference on Neural Networks (ICNN '97)*.

Suykens, J. A. K., Moor, B. L. R. D., and Vandewalle, J. (1995). Nonlinear system identification using neural state space models, applicable to robust control design. *International Journal of Control*, 62(1):129–152.

Suykens, J. A. K., Vandewalle, J. P., and DeMoor, B. L. (1996). *Artificial Neural Networks for Modelling and Control of Non-Linear Systems*. Kluwer Academic Publishers.

Tesauro, G. J. (1994). Td-gammon, a self-teaching backgammon program, achieves master level play. *Neural Computation*, 6(2).

Thorndike, E. L. (1911). *Animal Intelligence*. Hafner.

Underwood, D. M. and Crawford, R. R. (1991). Dynamic nonlinear modeling of a hot-water-to-air heat exchanger for control applications. *ASHRAE Transactions*, 97(1):149–155.

Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag.

Vapnik, V. N. (1998). *Statistical Learning Theory*. John Wiley and Sons, Inc.

Verrelst, H., Acker, K. V., Suykens, J., Moor, B. D., and Vandewalle, J. (1997). NLq neural control theory: Case study for a ball and beam system. In *Proceedings of the European Control Conference (ECC'97)*.

Vidyasagar, M. (1978). *Nonlinear Systems Analysis*. Prentice-Hall, Inc., 1 edition.

Vidyasagar, M. (1997). *A Theory of Learning and Generalization: With Applications to Neural Networks and Control Systems*. Springer-Verlag.

Watkins, C. J. (1989). *Learning from delayed rewards*. PhD thesis, Cambridge University.

Werbos, P. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University.

Werbos, P. (1992). *Neural Networks for Control*, chapter Overview of Designs and Capabilites. Bradford.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time markov environments. *Information and Control*, 34:286–295.

Young, P. M. (1996). Controller design with real parametric uncertainty. *International Journal of Control*, 65(3):469–509.

Young, P. M. and Dahleh, M. A. (1995). Robust ℓ_p stability and performance. *Systems and Control Letters*, 26:305–312.

Zames, G. (1966). On the input-output stability of nonlinear time-varying feedback systems, parts i and ii. *Transactions on Automatic Control*, 11:228–238 and 465–476.

Zhou, K. and Doyle, J. C. (1998). *Essentials of Robust Control*. Prentice Hall.