

Solving the Iterative Learning Control Problem with Reinforced Machine Learning in a Conjugate Basis Space



DARTMOUTH
ENGINEERING
THAYER SCHOOL

An Honors Thesis
in

Engineering Sciences

by Noah Dunleavy

Thayer School of Engineering
Dartmouth College
Hanover, New Hampshire

March 2025

Advisor _____
Professor Minh Q. Phan

Author _____
Noah Dunleavy

ABSTRACT

The tracking of a system’s output to some goal trajectory is a common industrial problem. Repetitive tasks conducted in a controlled manufacturing environment utilize complex machinery susceptible to noise and model characteristics not captured in their design or modelling process. Iterative Learning Control (ILC) leverages this repetitious process in the presence of unknown, yet repeatable, disturbances to improve the output of each trial.

Constructing a controller which brings about this reduction in error is difficult to do without a system model. Reinforcement Learning (RL) helps overcome this by providing techniques to build controllers purely from input-output data. The number of data points needed to extract such a controller is the squared sum of the number of states and number of inputs.

When a system is translated into its ILC format, the number of effective states and inputs is scaled up by the number of steps in the manufacturing process. This exponentially increases the number of trials that would need to be run to produce a controller through RL. It is then desirable to reverse this increase in dimensions.

To accomplish this, we employ basis functions on the the system input and outputs. We find that the number of basis functions describing the input must be less than or equal to the number describing the output, and the input necessary to produce our desired output must be in the space of the input basis functions – this requirement is not true for the output.

As we cannot know from the beginning what our goal input is, we must be able to dynamically grow our basis space representation in an efficient manner. To do so, we derive conjugate basis functions that are defined for a specific system.

We end with a methodology that allows for one to start with a low-dimension representation of a problem, learn the controller in the basis space through RL, and increase the dimensions as needed without compromising or changing the efficacy of previously learned parameters.

Acknowledgements

This project could not have been accomplished without the help and support of so many incredible people.

I would like to start off by thanking Professor Phan. I took ‘ENGS26: Control Theory’ with him in the Spring of my Sophomore year and by the end of the term had changed my major concentration. He presented topics in clear, understandable, and intuitive ways which I tried to replicate in this thesis. It was in his class ‘ENGS145: Modern Control Theory’ where the base ideas for this Thesis were formed, and ‘ENGS149: System Identification’ where I finally mustered the courage to ask him to be my advisor. I do not think a better mentor for a project such as this exists. I challenge you to find any other Professor who would be willing to stay around for hours on a Friday afternoon-turned-night to help work through proofs on the board, identify sources of errors, and provide paths forward for every problem that arose. It is my hope that the presented work is worthy of his time and dedication to my studies.

Next must come my partner Audrey Herrald. She has sat through hours of me frowning at a notebook, scratching at a chalkboard, banging at my keyboard, and generally thinking aloud - without complaint; all while going through Med School herself. She has heard all the material to follow more times than anyone, read more drafts than I even knew I went through, and has been a key figure of support to ensure I maintain priorities, focus, proper spelling, and sanity throughout this process.

Then comes my parents, Drs. Katherine and Keith Dunleavy. I have been truly fortunate in life to have the most amazing, loving, and supportive parents. All through my life, no matter what I did or how I did it, they have been there for me. Cheering me on from Little League to Robotics Competitions to Track Meets. No matter how much work I have ever had (or little I claimed to have), they provided me with every tool I needed to succeed. They have taught me through example what it means to work hard, take pride in what you

do, and care for those around you. Mom, Dad - I love you.

Of course many others played their part in this process and deserve recognition. My Grandparents - Ginghy and Grandad - for always checking in with me (and helping make sure I was never falling behind) and my sisters for letting me take over the entire kitchen table to do work whenever home. I would like to specifically thank John DeForest, Alexander Zhelyazkov, and Emily Lukas for reading early drafts of my Introduction.

Finally, thank you to all the friends, family, and mentors who were with me along the way. This would not have been possible without you all.

Contents

Abstract	ii
Acknowledgements	iii
List of Figures	xii
1 Introduction	1
1.1 Purpose of Background	2
1.2 Introduction to Continuous State Space	2
1.2.1 Example — State Space Formulation	3
1.3 Discretization of a Continuous Model	9
1.3.1 Example — Discretization	10
1.4 Defining Control	12
1.4.1 Example — Basic Control with Pole Placement	13
1.5 Linear Quadratic Regulator Controller	20
1.5.1 Example — LQR	23
1.6 Iterative Learning Control	28
1.6.1 Example — ILC	31
1.7 Reinforcement Learning	37
1.7.1 Policy Iteration	43
Example — Policy Iteration	48
1.7.2 Input Decoupling	51
Example — Input Decoupling	55
1.8 Summary	58
2 Methods and Experimentation	59
2.1 Reinforcement Learning on Iterative Learning Control	60
2.1.1 Example — Policy Iteration on ILC	61
2.1.2 Example — Input Decoupling on ILC	72
2.1.3 Summary of RL on ILC	78
2.2 Basis Functions	79
2.2.1 What are Basis Functions	79
2.2.2 Chebyshev Polynomials	81
Example — Matlab Creation of Chebyshev Polynomials	82
2.2.3 Basis Requirements in Reinforcement Learning	86

Demonstration of Requirements	89
FIPO	90
PIFO	95
FIPO vs PIFO	98
FISO	100
SIFO	103
FISO vs SIFO	106
Summary	106
2.2.4 Creating a Dynamic Basis Space from Learned Inputs	106
Example – Rolling Basis Space	107
Summary of Dynamic Spaces	113
2.2.5 Summary of Basis Function	113
2.3 Conjugate Basis Functions	114
2.3.1 What are Conjugate Basis Functions	114
2.3.2 Deriving Conjugate Basis Functions	115
2.3.3 Extracting Conjugate Basis Functions	117
The Batch Approach	118
Batch Example	120
The Iterative Approach	124
Iterative Example	127
Properties	132
3 Results	133
3.1 RL on ILC in a Conjugate Basis Space	134
3.1.1 Tying it all together	134
3.1.2 The Conjugate LQR Controller	135
Example – LQR in The Conjugate Space	135
3.1.3 Dynamic Conjugate Basis Space	140
Fixed Resolution: $\Phi_y = I$	140
Example – Full Resolution Identity on Output	140
Fixed Resolution $\Phi_y = \Phi^b$	144
Example – Full Resolution Conjugate Basis on Output	144
Fixed Resolution $\Phi_y = \underline{y}^*$	149
Example – $\Phi_y = \underline{y}^*$	149
Growing $\Phi_y = \Phi^{\overline{b}}$	151
Example – Growing $\Phi_y = \Phi^b$	152
4 Summary	155
5 Recommendations for Future Work	158
A Derivations	162
A.1 LQL Derivation	163

B Presentation	168
B.1 Final Presentation	168
C.2 Background	200
C Matlab Simulations and Examples	199
C.1 Github	199
C.2 Background	200
C.3 Basis Functions	230
C.4 Derivation and Demonstration of Conjugate Basis Functions	245
C.5 RL on Conjugate Basis	255
C.6 RL on ILC	271
D Matlab Functions	289
D.1 Controllability Check	290
D.2 Decoupled Learning for Iterative Learning Control	291
D.3 Default Figure Properties	299
D.4 Discounted LQR Solution	301
D.5 Draw a Goal	303
D.6 Figure Saving	306
D.7 Generate Chebyshev Polynomials	311
D.8 Batch Generate Conjugate Basis Functions	312
D.9 Iteratively Generate Conjugate Basis Functions	315
D.10 ILC Simulation	319
D.11 P Matrix from ABCD Model	323
D.12 Plot Basis Coefficients	326
D.13 Plot Controller History	333
D.14 Plot Dual-Mass System	337
D.15 Plot Iterative Learning Control Problem	341
D.16 Policy Learning for Iterative Learning Control	349
D.17 Generate Random Numbers in a Range	357
D.18 Plot Poles	358
Bibliography	362

List of Figures

1.1	Dual Spring-Mass-Damper System	4
1.2	Position data from a open-loop continuously-modelled Dual-Spring-Mass system	8
1.3	Position data from an open-loop discretely-modelled Dual-Spring-Mass system, overlaid with the continuous model to show exactness of the relationship	11
1.4	Zoomed-in view of the discrete-continuous model to show that at the discretely modelled Δt time-step samples, the relationship is exact	11
1.5	Pole locations for a Dual-Spring-Mass system when manually placed at locations $0.5 \pm 0.5i$ and $-0.7 \pm 0.1i$	14
1.6	Position of Mass 1 and Mass 2 for the Dual-Spring-Mass system under closed-loop, state-feedback controller F . F is designed such that the poles of $(A + BF)$ are at $0.5 \pm 0.5i$ and $-0.7 \pm 0.1i$. Inputs 1 and 2 are generated as $u(k) = Fx(k)$	16
1.7	Pole locations for a Dual-Spring-Mass system when manually placed them at the origin to produce a deadbeat controller	18
1.8	Position of Mass 1 and Mass 2 for our Dual-Spring-Mass system under a deadbeat closed-loop, state-feedback controller F . F is designed such that the poles of $(A + BF)$ are at 0. Inputs 1 and 2 are generated as $u(k) = Fx(k)$. Under deadbeat control, it can be seen that control is achieved under n steps	19
1.9	Illustration of Principle of Optimality, with nodes A , $B1$, $B2$, B , and C with paths between labelled with their associated costs. Even though to go from $A \rightarrow B1$ is only a cost of 2, $B1 \rightarrow C$ costs 12 making the total path cost of 14. Also see that the path of $A \rightarrow B2 \rightarrow C$ may have a final step of cost 1, but the first step has a cost pf 15. The central path through B then could result in costs of 15, 14, 11, or 10. Clearly going through B is the optimal way to proceed. It can then further be seen that the optimal way to go from $B \rightarrow C$ is a subset of the optimal path of $A \rightarrow C$	22
1.10	Pole Locations of a $Q/R = 100$ LQR Controller on our Dual-Spring-Mass System	24
1.11	Positions of Mass 1 and 2 under a state-feedback controller defined under LQR parameters of $Q/R=100$. Control is achieved within 200 samples, and under maximum input amplitudes of 55N	25

1.12	Pole Locations of a $Q/R = 10$ LQR Controller on our Dual-Spring-Mass System	26
1.13	Positions of Mass 1 and 2 under a state-feedback controller defined under LQR parameters of $Q/R=10$. Control is achieved within 800 samples, and under maximum input amplitudes of 9N	27
1.14	Error Progression of an ILC problem when using a perfect knowledge controller $\mathcal{L} = 0.8P^+$ such that the poles of the system under $(I - P\mathcal{L})$ are guaranteed to be within the unit circle and relatively close to the origin for rapid convergence.	33
1.15	The progression of Input-Output trials under our controller of $\mathcal{L} = 0.8P^+$. Trial 1 can be seen to be the open-loop response, and by Trial 10 it can be seen that the output is captured with zero error	34
1.16	Progression of shaped outputs (where Mass 1 Position is the x-coordinate and Mass 2 Position is the y-coordinate) under our ILC controller $\mathcal{L} = 0.8P^+$	35
1.17	Application of ILC Controller $\mathcal{L} = 0.8P^+$ on our Dual-Spring-Mass system to learn the output ‘Dartmouth’. It can be seen that initial conditions and arbitrariness of different goals has no impact on the efficacy of ILC	36
1.18	Progression of Mass 1 and 2 Positions under controller $\mathcal{L} = 0.8P^+$, learning ‘Dartmouth’. It can be seen that Mass 1, the x-position, gradually increases throughout the each trial whereas Mass 2, the y-position, simply moves back-and-forth / up-and-down	37
1.19	Input-Output Data of our Dual-Spring-Mass system under 5 Policy Iteration Trials of 36 samples/steps each. Learning parameters of $Q/R = 100$ and $\gamma = 0.8$. After 5 controllers, the learning stops and exploration $v(k)$ is no longer applied to the input for the final 20 trials.	50
1.20	Progression of controller weights through Policy Iteration Trials. For two inputs, there are two rows of the controller F to describe how to weight their respective inputs from the associated samples collected states.	51
1.21	Input-Output Data of Dual-Spring-Mass system under Input Decoupled Learning. 5 passes are made on each input, of which we have two, and requires 25 trials each. Learning is halted for the final 20 trials which can be seen by both inputs being smooth at the same time.	57
1.22	Progression of Controller Weights through Input-Decoupled trials. Notice how for the dual-input system, the weights for a given input are only updated every other trial.	57
2.1	Select Controller Weights on Select Inputs for an ILC problem through Policy Iteration Trials. 5 Controllers are learned, for an ILC system on the Dual-Spring-Mass system of trial length $p = 10$ – each controller update requires 1600 ILC trials	64
2.2	Error Magnitude of Output through Policy Iteration Trials, where $Q/R = 100$. Observe that after the first controller is learned and applied, there is a sharp reduction in error but due to the relatively high R in its definition, subsequent reductions in error are much slower.	65

2.3	Input-Output Progressions through Policy Iteration Trials of $Q/R = 100$. Observe Trial 1 to be the open-loop response, and subsequent trials to be progressing towards the goal output.	65
2.4	Shaped Output through Policy Iteration Trials of $Q/R = 100$. Observe the progression towards the desired output, but also the extreme number of trials that would be needed to properly further the learning.	66
2.5	Select Controller Weights on Select Inputs progression through Policy Iteration Trials when $Q/R = 1 \times 10^8$. Due to the necessary amplified exploration $v(k)$, observe the tendency for weights to converge much less smoothly than before, displaying behaviors of overshoot and lag.	69
2.6	Error Magnitude of Output through Policy Iteration Trials with $Q/R = 1 \times 10^8$. Observe how initial errors creep up much further than the earlier shown $Q/R = 100$ trial, but the application of the first trial drastically reduces error , such that it would be zero were it not for the extreme input exploration terms.	69
2.7	Progression of Input-Output Data in a Dual-Spring-Mass system under ILC derived from RL when $Q/R = 1 \times 10^8$	70
2.8	Shaped Output through Policy Iteration Trials when $Q/R = 1 \times 10^8$	71
2.9	Select Controller Weights on Select Inputs through Input Decoupling Trials to learn the ILC Controller when $Q/R = 100$. Notice how each controller update takes 441 ILC trials, and each input controller is only updated at that rate.	74
2.10	Error Magnitude of Output through Input Decoupling Trials	75
2.11	Input-Output Data progression through Input Decoupled Learning trials when $Q/R = 100$	76
2.12	Shaped Output through Input Decoupling Trials when $Q/R = 100$	77
2.13	Select Chebyshev Polynomials	83
2.14	Example Signal constructed from $\eta = 20$ Chebyshev Polynomials of length $\ell = 100$	84
2.15	Example Chebyshev Weights to Generate Signal in Fig. 2.14	85
2.16	Goal Inputs deconstructed from an \underline{u} explicitly constructed from basis functions to be in Φ_u	91
2.17	Deconstructed \underline{y}^* 's vs the goal constructed from α^* . α^* is found by inverting the the \underline{y}^* in the space Φ_y . By then attempting to go back, this highlights the inability of Φ_y to perfectly capture \underline{y}^*	92
2.18	Progression of Errors on coefficients through perfect-knowledge controller trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$. Even though $\underline{y}^* \notin \Phi_y$, the associated coefficient errors can still go to zero.	93
2.19	Progression of Coefficients through perfect-knowledge controller trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$	93
2.20	Progression of outputs through ILC trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$	94
2.21	Goal Outputs deconstructed from a \underline{y}^* explicitly constructed from basis functions to be in Φ_y	95
2.22	Deconstructed Goal \underline{u}^* 's from $\Phi_u \beta^*$, where β^* was backed out of \underline{y}^* with perfect knowledge.	97

2.23 Progression of coefficient errors through trials when $\underline{u}^* \notin \Phi_u$ but $\underline{y}^* \in \Phi_y$. Contrast this with the earlier example where $\underline{u}^* \in \Phi_u$, we see now that the error of α can still go to zero, but the error on β reaches a non-zero steady state.	97
2.24 Progression of Coefficients through trials when $\underline{u}^* \notin \Phi_u$ but $\underline{y}^* \in \Phi_y$	98
2.25 Progression of Outputs through ILC trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$	99
2.26 Deconstructed Goal Outputs for $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \Phi_y$	100
2.27 Progression of coefficient errors through trials when $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \Phi_y$	101
2.28 Progression of coefficients through trials when $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \bar{\Phi}_y$	101
2.29 Progression of Outputs through ILC trials when $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \Phi_y$. Even though \underline{u}^* could in theory be fully described by Φ_u , the controller is unable to capture \underline{y}^*	102
2.30 Deconstructed Goal Inputs for $\underline{u}^* = \Phi_u$ and $\underline{y}^* \in \Phi_y$	103
2.31 Progression of coefficient errors through trials when $\underline{u}^* = \Phi_u$ and $\underline{y}^* \in \Phi_y$	103
2.32 Progression of coefficients through trials when $\underline{u}^* = \Phi_u$ and $\underline{y}^* \in \bar{\Phi}_y$	104
2.33 Progression of Positions through ILC trials when $\underline{u}^* = \Phi_u$ and $\underline{y}^* \in \Phi_y$. While the learned shape may appear arbitrary, the important take-away is that the error is zero, and the learned output matches that exactly of the goal.	105
2.34 Progression of e_α through rolling basis space episodes	109
2.35 Progression of e through rolling basis space episodes	110
2.36 Progression of deconstructed goal inputs and the generated shaped output through rolling basis space episodes. While the output can be seen to be close to our goal, even after working our way through the entirety of the input basis space, we still fail to properly capture our \underline{y}^* . Failure to capture \underline{u}^* can best be seen on the fringes of the input signals.	111
2.37 Progression of β through rolling basis space episodes. After sufficient trials have been made to cover all of Φ_{full} , it can be seen that β_1 sits around a value of 1 - meaning that the learned input is predominately that of the previously learned pass. However, there are still tiny additions from other components, evident by the non-zero β_{2-4}	112
2.38 Progression of β through rolling basis space episodes. After 7 iterations, Φ is forced to have \underline{u}^* in it. It is demonstrated that the controller continues to properly identify that as the correct input. It can be seen, given this, that the earlier example never actually finds \underline{u}^* before being moved away – once \underline{u}^* is found, it is not lost.	112
2.39 Output Generated from Batch Generated Basis Space Φ with weights β^* overlaid on the Goal Output. As the \underline{u}^* was generated within a space spanned by all of the \underline{u}^e s that constructed the basis space, the perfect output is captured.	124
2.40 Output Generated from Iteratively Generated Basis Space Φ with weights β^* when $\eta = 1$ overlaid on the Goal Output	130
2.41 Output Generated from Iteratively Generated Basis Space Φ with weights β^* when $\eta = 3$ overlaid on the Goal Output	131

2.42	Output Generated from Iteratively Generated Basis Space Φ with weights β^* when $\eta = 8$ overlaid on the Goal Output. It is demonstrated that we do not to explore the full space, as done in the batch example, and can instead learn one basis at a time. By learning iteratively, we can stop once we have determined our space to be good enough for our desired output.	131
3.1	Error Progression through ILC Trials with a Perfect Knowledge Controller in a Full Conjugate Basis Space	138
3.2	Shaped Output Progression through ILC Trials with a Perfect Knowledge Controller in a Full Conjugate Basis Space	139
3.3	Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs. For a fixed output basis $\Phi_y = I$, 20 ϕ s are tried to capture the input, and the associated F_{LQR}^γ is learned (but not applied).	142
3.4	Shaped Output Progression through Application of a Final Controller that was Learned One ϕ at a time. The resultant controller is then built out of the individually learned F_{LQR}^γ s for each ϕ	143
3.5	Error Progression through Policy Learning ILC Trials with Scaled Conjugate Output Basis	147
3.6	Pole locations of the system under a controller found with a fixed $\Phi_y = 10\Phi$ and by rotating through $\Phi_u = \Phi$. Of the 20 poles, 18 are exactly 1 while 2 are 0.9991 and 0.9998. This is due to the required R values.	148
3.7	Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs and $\Phi_y = \underline{y}^*$. Given the relatively high R , 0 error is not achieved quickly but it can be seen to be being reduced.	150
3.8	Complete Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs and $\Phi_y = \underline{y}^*$. It can now be seen that the tall controller constructed one element at a time, while efficient to build, does not bring about desired results on no error.	151
3.9	Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs and outputs	154

List of Acronyms

FIFO Full Input, Partial Output Basis Functions

FISO Full Input, Single Output Basis Functions

ILC Iterative Learning Control

LQR Linear Quadratic Regulator

LQL Linear Quadratic Optimal Learning Control

PIFO Partial Input, Full Output Basis Functions

RL Reinforcement Learning

SIFO Single Input, Full Output Basis Functions

ZOH Zero-Order Hold

Chapter 1

Introduction

1.1 Purpose of Background

This section is dedicated to providing the base information of Modern Control Theory referenced in this Thesis.

To those familiar with the Thayer School of Engineering's curriculum, it is very similar to the content of ENGS145 - Modern Control Theory.

We begin with system formulation and representation in the matrix form, and how we resolve the difference between the continuous nature of the world and the discrete limitations of computers. Here, we use the Zero-Order Hold (ZOH) approach.

Next the idea of pole placement is introduced, and it is demonstrated that the further from the origin the poles are, the longer control takes. A deadbeat controller is used to highlight this, which is the time-optimal solution for any system.

One will note that the deadbeat controller, while time optimal, requires significant control effort that may not be realistic or safe for a real system. That leads us to our introduction of the Linear Quadratic Regulator (LQR) controller, which minimizes a cost function defined by system inputs and states.

Next we introduce the ILC problem and show that it can learn to generate any goal output (so long as permitted by the physical characteristics of the system), regardless of initial conditions or noise.

Finally, we address the assumption of perfect knowledge not typically possible in the real world. The machine learning process of RL is shown via the Policy Iteration and Input Decoupling method. They can be shown to find the LQR controller as defined by its cost function.

1.2 Introduction to Continuous State Space

A key step for Control Theory is construction of a system model, commonly represented as A_c , B_c , C and D . A_c captures the impact that the current state will have on the next

state and B_c captures how inputs will impact the next state. The matrix C captures how states are translated to measured outputs and D captures how inputs are directly measured on outputs. Any linear system, regardless of complexity and variations, can be modelled exactly as follows:

$$\dot{x}(t) = A_c(t)x(t) + B_c(t)u(t) + \omega_x(t) \quad (1.1)$$

$$\dot{y}(t) = C(t)x(t) + D(t)u(t) + \omega_u(t) \quad (1.2)$$

where $A_c(t)$, $B_c(t)$, $C(t)$, and $D(t)$ are the matrices describing the system dynamics, and the ω terms represent noise (like a residual in a regression). $x(t)$ is the state vector of dimensions $n \times 1$, where n is the number of states. There will be one state for every energy storing element in the system. $u(t)$ is the input vector of dimensions $r \times 1$, where r is the number of inputs. $y(t)$ is the output vector of dimensions $m \times 1$, where m is the number of outputs. As such, A_c is $n \times n$, B_c is $n \times r$, C is $m \times n$ and D is $m \times r$. Note that the matrices can be expressed in a time-variant form (a function of time), however for the entirety of this paper all matrices will be time-invariant. That is: $A_c(t) = A_c$ and the same goes for B_c , C , and D .

1.2.1 Example — State Space Formulation

Most, if not all, of the world's physical systems can be modelled as a spring-mass-damper system. The mass and spring system that will be repeatedly referenced in this project is thus a damped, two-mass-spring system, as seen in Fig 1.1

We have two masses, connected in series with springs and dampers, and bounded on one end with a wall. Constructing the equations of motion for the system simply follows Newton's second law ($F = ma$), employing Hooke's ($F = -kx$) and Damping Laws ($F = -cv$). Recognizing that the derivative of position is velocity, and the derivative of

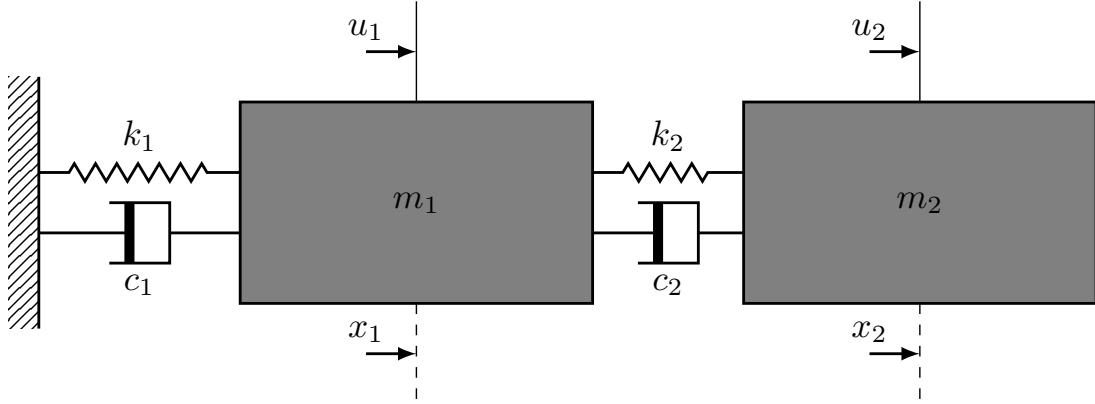


Figure 1.1: Dual Spring-Mass-Damper System

velocity is acceleration, it can be shown that the equations of motion (EoM) for each mass are as shown in Eqs. 1.3 and 1.4.

$$\begin{aligned}\ddot{x}_1(t) &= x_1(t) \left(\frac{-k_1 - k_2}{m_1} \right) + x_2(t) \left(\frac{k_2}{m_1} \right) \\ &\quad + \dot{x}_1(t) \left(\frac{-c_1 - c_2}{m_1} \right) + \dot{x}_2(t) \left(\frac{c_2}{m_1} \right) + u_1(t) \left(\frac{1}{m_1} \right)\end{aligned}\tag{1.3}$$

$$\begin{aligned}\ddot{x}_2(t) &= x_1(t) \left(\frac{k_2}{m_2} \right) + x_2(t) \left(\frac{-k_2}{m_2} \right) \\ &\quad + \dot{x}_1(t) \left(\frac{c_2}{m_2} \right) + \dot{x}_2(t) \left(\frac{-c_2}{m_2} \right) + u_2(t) \left(\frac{1}{m_2} \right)\end{aligned}\tag{1.4}$$

From here we select our ‘states’. For every energy-storing element in a system there will be one state. Our system stores energy as kinetic energy in the masses, and potential energy in the springs - we have four energy-storing elements and thus four states. It is most common and logical in spring-mass problems to select the position and velocity of

the masses as these states:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} \quad (1.5)$$

Now for our inputs: these also are commonly and easily expressed as direct scalars of themselves. As such, our input vector is as follows:

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad (1.6)$$

Recall the idea of our model is to capture what the change in states will be – given current states and inputs. In the continuous format, the change in states is captured in the time derivative of the state vector, as shown in Eq. 1.7 below

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} \quad (1.7)$$

With all this matrix information, it is now time to construct our continuous state-space model of the form seen in Eq. 1.1. Through matrix multiplication we arrive upon the following:

$$\dot{x} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{-k_1-k_2}{m_1} & \frac{k_2}{m_1} & \frac{-c_1-c_2}{m_1} & \frac{c_2}{m_1} \\ \frac{k_2}{m_2} & \frac{-k_2}{m_2} & \frac{c_2}{m_2} & \frac{-c_2}{m_2} \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{1}{m_1} & 0 \\ 0 & \frac{1}{m_2} \end{bmatrix} u \quad (1.8)$$

Recognizing this simple system is already messy, further substitutions can be made by formatting the masses, spring constants, and damping coefficients into Mass (1.9), Stiffness

(1.10), and Damping (1.11) Matrices. These are known as physical parameter matrices.

$$M = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \quad (1.9)$$

$$K = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} \quad (1.10)$$

$$C = \begin{bmatrix} c_1 + c_2 & -c_2 \\ -c_2 & c_2 \end{bmatrix} \quad (1.11)$$

This allows us to express our equations of motion in a single line as in Eq. 1.12

$$\begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} = M^{-1}K \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + M^{-1}C \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} + M^{-1} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad (1.12)$$

and our state-space model can now be written as

$$\dot{x} = \begin{bmatrix} 0_{2 \times 2} & I_{2 \times 2} \\ -M^{-1}K & -M^{-1}C \end{bmatrix} x + \begin{bmatrix} 0_{2 \times 2} \\ -M^{-1} \end{bmatrix} u \quad (1.13)$$

This is exactly the format for the state-space model we described earlier in Eq. 1.1.

Now we put some numbers to our example so we can simulate behavior. We will define our system as follows

$$m_1 = 1 \text{ kg} \quad m_2 = 0.5 \text{ kg} \quad (1.14)$$

$$k_1 = \frac{100 \text{ N}}{\text{m}} \quad k_2 = \frac{200 \text{ N}}{\text{m}} \quad (1.15)$$

$$c_1 = \frac{1 \text{ Ns}}{\text{m}} \quad c_2 = \frac{0.5 \text{ Ns}}{\text{m}} \quad (1.16)$$

such that our physical parameter matrices are

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix} \quad (1.17)$$

$$K = \begin{bmatrix} 300 & -200 \\ -200 & 200 \end{bmatrix} \quad (1.18)$$

$$C = \begin{bmatrix} 1.5 & -0.5 \\ -0.5 & 0.5 \end{bmatrix} \quad (1.19)$$

Plugging these values into Eq. 1.13, we can write our system model as

$$\dot{x} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -300 & 200 & -1.5 & 0.5 \\ 400 & -400 & 1 & -1 \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 2 \end{bmatrix} u \quad (1.20)$$

To explicitly complete the connection to Eq. 1.1, we can see

$$A_c = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -300 & 200 & -1.5 & 0.5 \\ 400 & -400 & 1 & -1 \end{bmatrix} \quad B_c = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 2 \end{bmatrix} \quad (1.21)$$

We now only need one more piece of information to completely describe this system's behavior, and that is its initial conditions. We will choose to displace the first mass one meter to the right of its resting position and have the second mass moving to the right at

two meters per second.

$$x_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} \quad (1.22)$$

Armed with the information to completely model the system, we now decide on our outputs. Recall Eq. 1.2 relating the current state and inputs to the output y , via matrices C and D . For our system we will choose to only record the block positions (x_1 and x_2) as our outputs. Monitoring those two states is represented simply in matrix form

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} u \quad (1.23)$$

where we will once again explicitly note

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (1.24)$$

The resulting outputs from simulating out this system can be seen in Fig. 1.2a and Fig. 1.2b

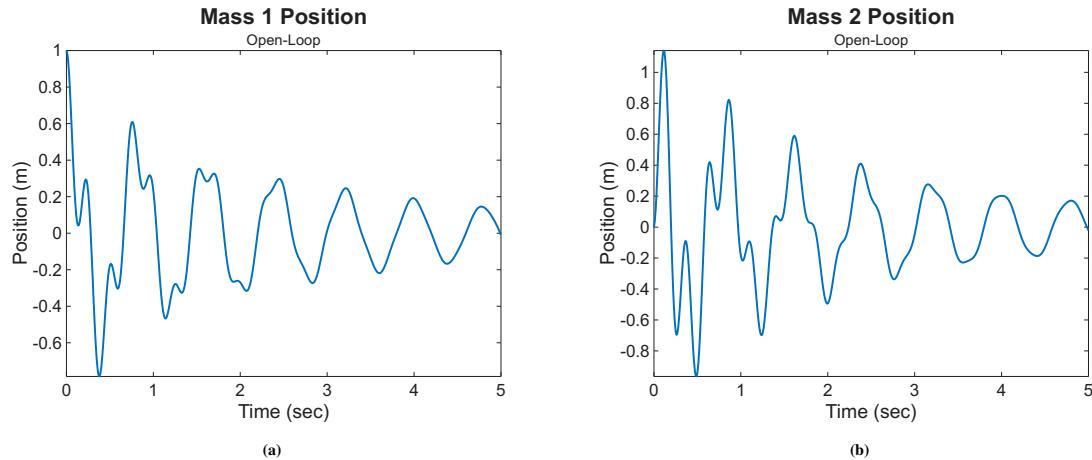


Figure 1.2: Position data from an open-loop continuously-modelled Dual-Spring-Mass system

1.3 Discretization of a Continuous Model

So far, we have been dealing with the ideal scenario of continuous time. While the models we constructed are exact, it is infeasible to collect outputs and apply inputs to a system at an infinite rate implied by a continuous model. Even if we were able to, it would be inefficient and impractical to run an infinite amount of calculations in an infinitely small time span. Digital systems fix this by discretizing their actions and outputs at a sampling rate denoted by Δt (typically in units of seconds). That is, every Δt a new output is collected and input applied. To maintain the exact nature of the above model, matrices $A_c(t)$ and $B_c(t)$ must be ‘discretized’ through the ZOH method. The equations for doing so are shown in Eqs. 1.25 and 1.26

$$A = e^{A_c \Delta t} \quad (1.25)$$

$$B = \int_0^{\Delta t} e^{A_c \alpha} d\alpha B_c \quad (1.26)$$

What this relationship now constitutes is rather than applying instantaneous inputs, inputs are applied every Δt and held for Δt . The response of the system between samples is not fully captured in the model, but at every discrete time step k the model-to-nature relationship is exact. The output collection matrices C and D do not need to be adjusted. We re-write our continuous state-space models from Eqs. 1.1 and 1.2 as

$$x(k+1) = Ax(k) + Bu(k) \quad (1.27)$$

$$y(k) = Cx(k) + Du(k) \quad (1.28)$$

An important notation distinction in discrete systems is the use of k instead of a time value. k represents discrete samples, occurring every Δt , but is unitless. To convert from a sample number k to a continuous time, simply multiply the sample number by the sample rate.

$$t = k\Delta t \quad (1.29)$$

As with all sampling and discretization, one must be wary of Nyquist sampling. A given system will have inherent ‘modes’ - frequencies which it is easily excited and operates at. If your sampling rate Δt is not sufficiently small, the exactness of the model will fail to capture crucial system dynamics.

Providing all the above steps and criteria are observed, we will be left with a model that exactly matches that of a continuous system at the time steps specified.

1.3.1 Example — Discretization

Continuing with our earlier model, we now seek to discretize it for practical computational techniques. We will set $\Delta t = 0.01$ seconds, and apply equations 1.25 and 1.26. The resulting discrete matrices are

$$A = \begin{bmatrix} 0.985 & 0.0099 & 0.0099 & 0.0001 \\ 0.0198 & 0.9802 & 0.0001 & 0.0099 \\ -2.9398 & 1.9522 & 0.9704 & 0.0147 \\ 3.9191 & -3.9306 & 0.0295 & 0.9704 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0.0001 \\ 0.0099 & 0.0001 \\ 0.0001 & 0.0198 \end{bmatrix} \quad (1.30)$$

Note that in this perfect information scenario, we can verify our sufficient Δt by examining the continuous-time system matrix A_c . The imaginary components of the eigenvalues are the natural frequencies of the system that A_c describes. In our case, we have conjugate pairs with frequencies of 25.2 and 7.9 rad/sec. As we only care about the highest frequency (and sign does not matter), we convert 25.2 rad/sec to 4.0143 Hz. To avoid Nyquist sampling, we must sample at more than two times this rate, or over 8.0286 Hz. This corresponds with a sampling interval of 0.1246 seconds, which we are well below with our 0.01 second interval.

Now our system is captured in discrete form, as outlined in Eq. 1.27. Running that out and overlaying it with the results of the continuous system we arrive upon outputs depicted in Figures 1.3a and 1.3b.

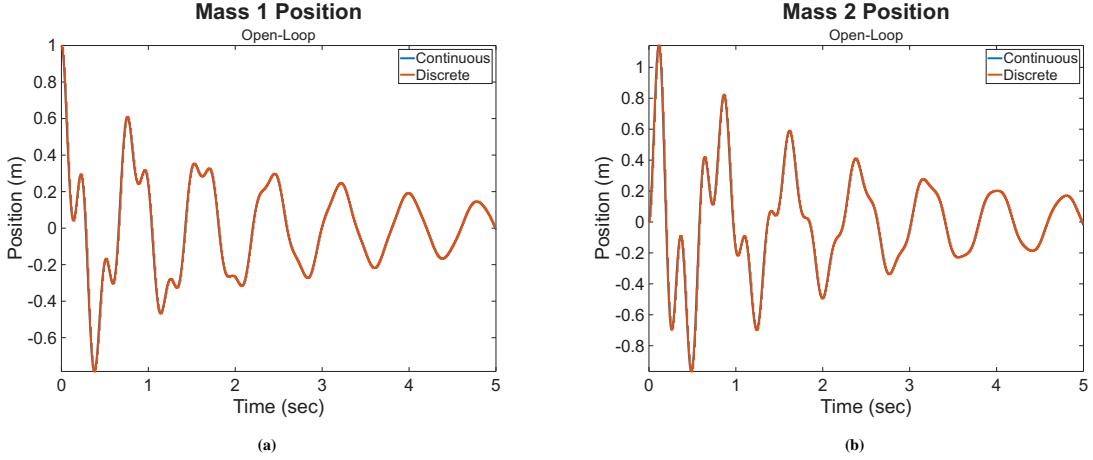


Figure 1.3: Position data from an open-loop discretely-modelled Dual-Spring-Mass system, overlaid with the continuous model to show exactness of the relationship

From this zoomed out view, it can be difficult to believe that the exact relationship does exist. Figures 1.4a and 1.4b step in to show that at every sampling interval of 0.01 seconds, the discrete model (outputs and states) match exactly that of the continuous one.

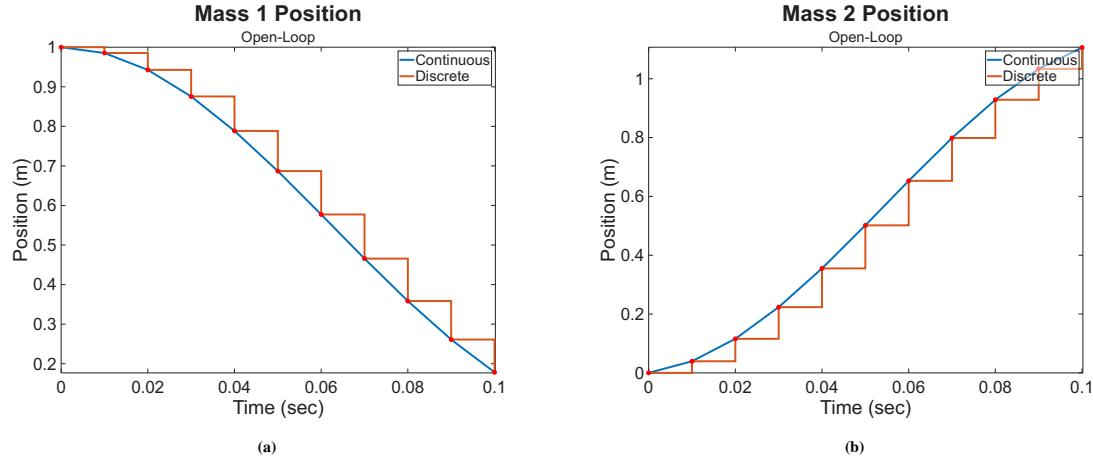


Figure 1.4: Zoomed-in view of the discrete-continuous model to show that at the discretely modelled Δt time-step samples, the relationship is exact

One thing to note – moving forward, the discrete system model's x-axis will be expressed in terms of sample number k . For example, Figure 1.4a axis will be marked with k intervals 0 through 10, as opposed to time 0 through 0.1 seconds.

1.4 Defining Control

With our model now modified to be exact computationally, we can now proceed to do something with it. From a system model, it is the goal to control the system. The simplest definition of ‘controlled’ is once all states are zero – this most classically is a system at rest. When the input to a system is some function of the system’s states and/or outputs, we describe the system as ‘closed loop’. A typical controller demarcated by F will be of dimensions $r \times n$ and is used to calculate inputs from collected data. As such, each input obeys the following control law when seeking stabilization (all states go to zero):

$$u(k) = Fx(k) \quad (1.31)$$

Following this control law, the state space equation in Eq. 1.27 can be re-written as:

$$x(k+1) = Ax(k) + BFx(k) \quad (1.32)$$

$$= (A + BF)x(k) \quad (1.33)$$

A controlled system means that $x(k) = 0$, so as k goes to infinity, we would like $x(k)$ to go to zero. Any formulation of $A + BF$ that causes $x(k+1)$ to be smaller in magnitude than $x(k)$ will eventually result in an $x(k) = 0^1$. In the scalar case this is easy to understand; suppose

$$x(k+1) = \alpha x(k) \quad (1.34)$$

Where $-1 < \alpha < 1$, then $\lim_{k \rightarrow \infty} x(k) = 0$. Taking this to a matrix-space preserves this intuition, only now instead of placing a scalar between -1 and 1, we seek to place the eigenvalues, or poles, of the system within the unit circle of the complex plane. Then regardless of any dynamics, a system will converge to a ‘controlled’ zero state over sequential samples. Poles placed at the boundary of the unit circle would denote an asymptotically stable

¹Note that the controller depends only on the system matrices A and B , and not at all on initial conditions

system – like a ball at the top of a hill, which will be stable until some force comes along and pushes it.

For any system defined by their A and B matrices, it is relevant to check if it is controllable. That is – is it possible to actually send all the n states to zero, with our selected r inputs. This can be checked by examining the Controllability Matrix. Defined as

$$\mathcal{C} = [A^{(n-1)}B, \dots, AB, B] \quad (1.35)$$

If the Controllability Matrix \mathcal{C} is full rank in rows, then the system is controllable. Full row rank means that no row of the matrix can be formulated through any linear combination of any of the other rows – in other words, all rows are independent of one another². If a matrix is said to be ‘full rank’, then it will have a rank (number of linearly independent rows/columns) equal to its smallest dimension. So a full-rank 4×2 matrix will have rank of 2.

1.4.1 Example — Basic Control with Pole Placement

There is an infinite number of controllers F that could send our system to stability. The closer the poles of $(A + BF)$ are placed to the origin, the more rapid the convergence of the system will be. To illustrate this, we will first place the poles of our controlled moderately far from the origin, as seen in 1.5. Placement is done using MATLAB’s *place* function and then verified. Poles must always come in conjugate pairs, as visible in the reflection over the imaginary axis.

The poles, placed at $0.5 \pm 0.5i$ and $-0.7 \pm 0.1i$ result in controller F_{simple}

$$F_{simple} = \begin{bmatrix} -40,364 & -67,217 & -161 & -71 \\ 5,468 & 7,726 & -10 & -73 \end{bmatrix} \quad (1.36)$$

²The same logic applies for full ‘column rank’

Pole Locations

Simple Pole Placement

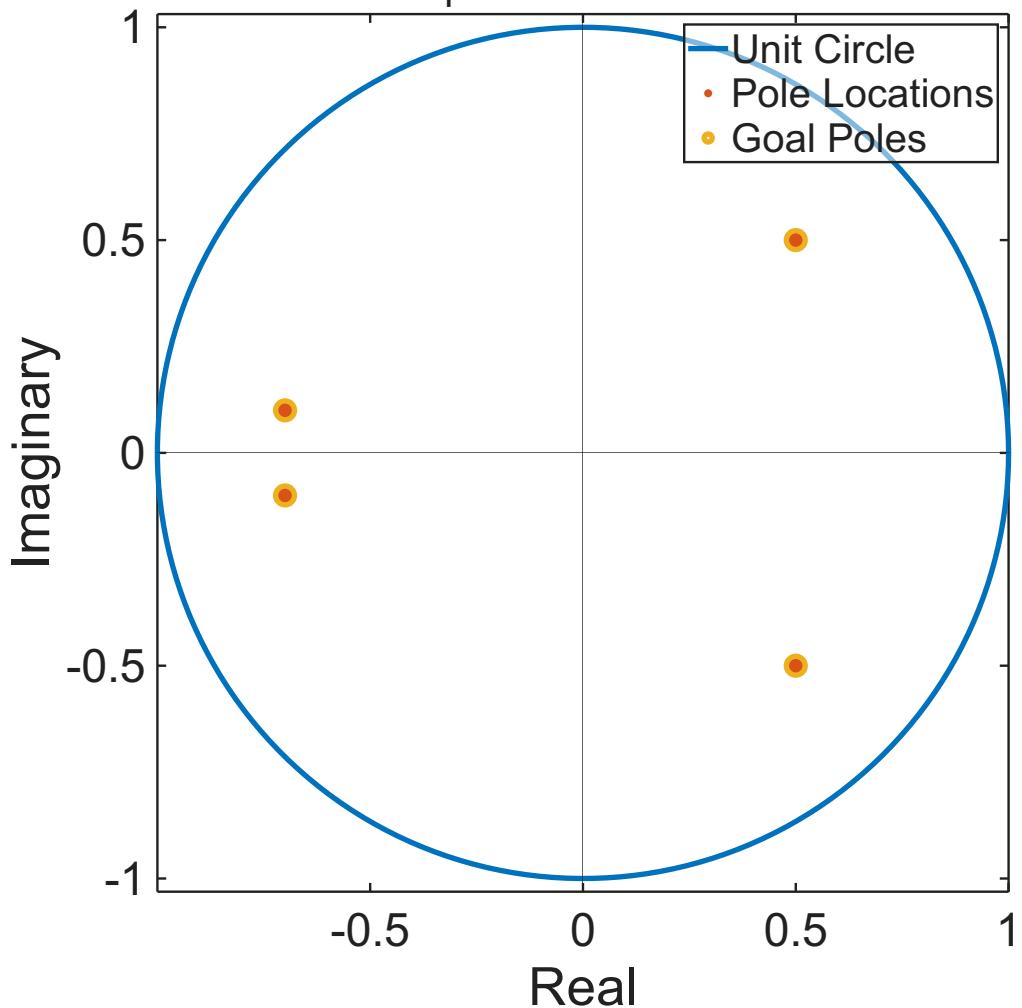


Figure 1.5: Pole locations for a Dual-Spring-Mass system when manually placed at locations $0.5 \pm 0.5i$ and $-0.7 \pm 0.1i$

The way to interpret this controller (and framework for all future controllers) is best described with an example state, for which we will use our initial condition $x(0)$ as shown in Eq. 1.22. The first row of the controller dictates how our first input (u_1 on m_1) is computed given current states $x(0)$ (see Eq. 1.22). In this case

$$\begin{aligned} u_1(0) &= -40,364 \cdot 1 + -67,217 \cdot 0 + -161 \cdot 0 + -71 \cdot 2 \\ &= -40,506 \end{aligned} \tag{1.37}$$

The same process can be repeated for u_2 and for any sample number k . The way to verbalize a controller like this is as a state-input-weight relation, where the states are the columns, the inputs the rows, and the weight the corresponding value. For example, for every unit away from control x_1 is (recall ‘control’ is a zero state), u_1 will generate -40,364 units of input. The net input will be the summed effects of each state. When the controller in Eq. 1.36 is applied to our dual-mass system, our system produces outputs seen in Figs. 1.6a and 1.6b, under the inputs shown in Figs. 1.6c and 1.6d

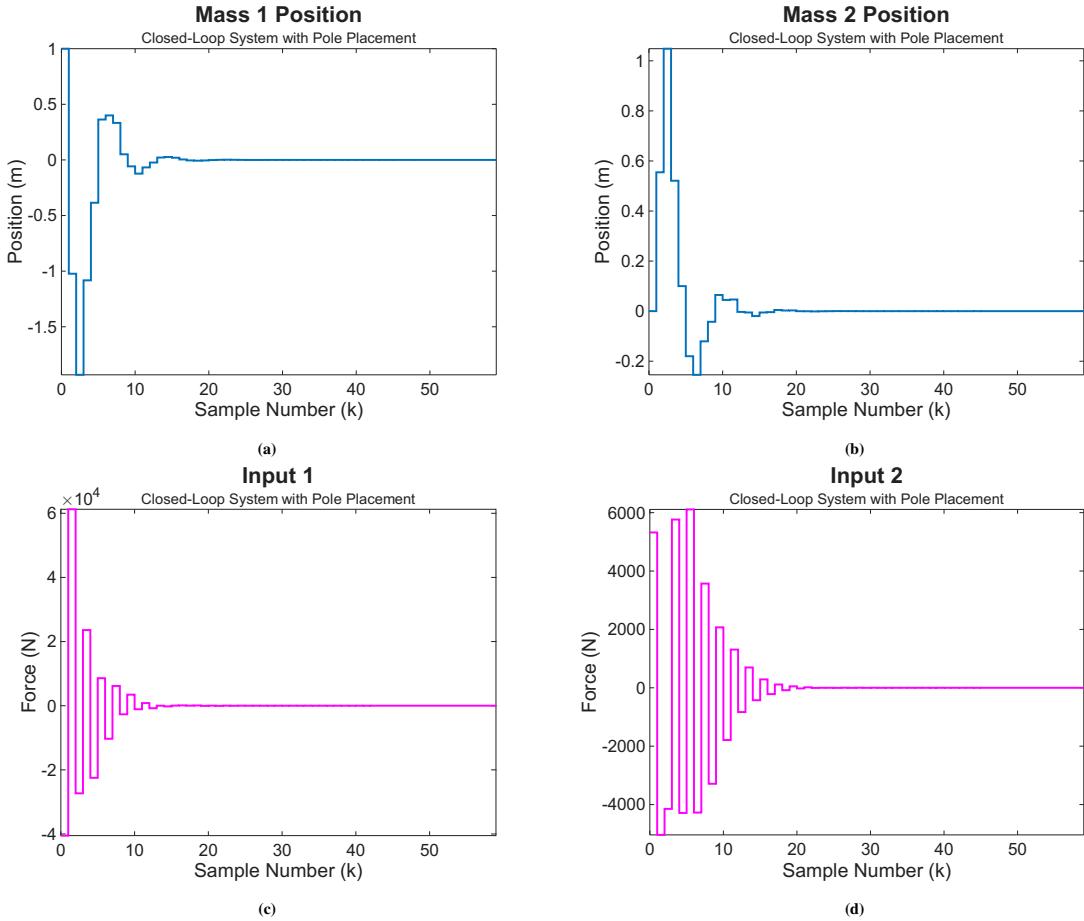


Figure 1.6: Position of Mass 1 and Mass 2 for the Dual-Spring-Mass system under closed-loop, state-feedback controller F . F is designed such that the poles of $(A + BF)$ are at $0.5 \pm 0.5i$ and $-0.7 \pm 0.1i$. Inputs 1 and 2 are generated as $u(k) = Fx(k)$

One will note that the system is stabilized after about twenty samples, or .2 seconds. The cost, however, is reflected in the inputs. What's known as the 'control effort' applied is magnitudes more than the given state.

This can be taken even further with a special type of controller known as a 'deadbeat' controller. This is a controller which places the poles of a closed-loop system at the origin³ and produces the time-optimal solution. Once again, the system poles are shown in Figure 1.7 and produces controller $F_{deadbeat}$

$$F_{deadbeat} = \begin{bmatrix} -9,800.7 & -158 & -148.8 & -0.7 \\ -158 & -4,841.9 & -0.7 & -74.3 \end{bmatrix} \quad (1.38)$$

Its application results in the outputs and inputs shown in Figures 1.8a - 1.8d

This pole placement method, while useful to demonstrate the requirements of a linear-feedback controller, is crude and often results in extreme states or inputs – if not both. It would be useful then to be able to design a controller which can be tweaked more precisely to adhere to certain system limits.

³MATLAB's *place* function does not allow for multiple poles to be placed at the same location. Either use *acker* or place poles very close to 0 ($\leq 1 \times 10^{-6}$)

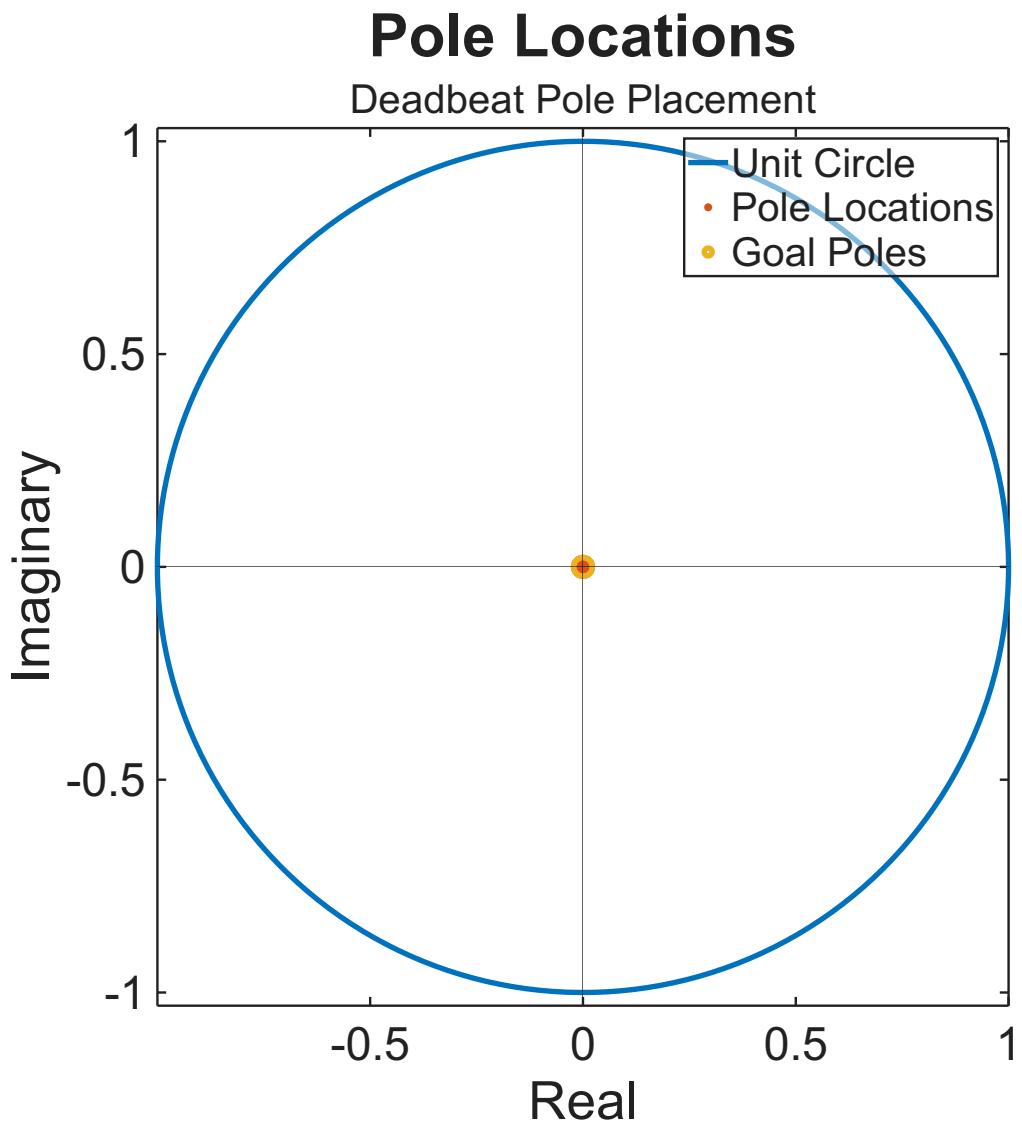


Figure 1.7: Pole locations for a Dual-Spring-Mass system when manually placed them at the origin to produce a deadbeat controller

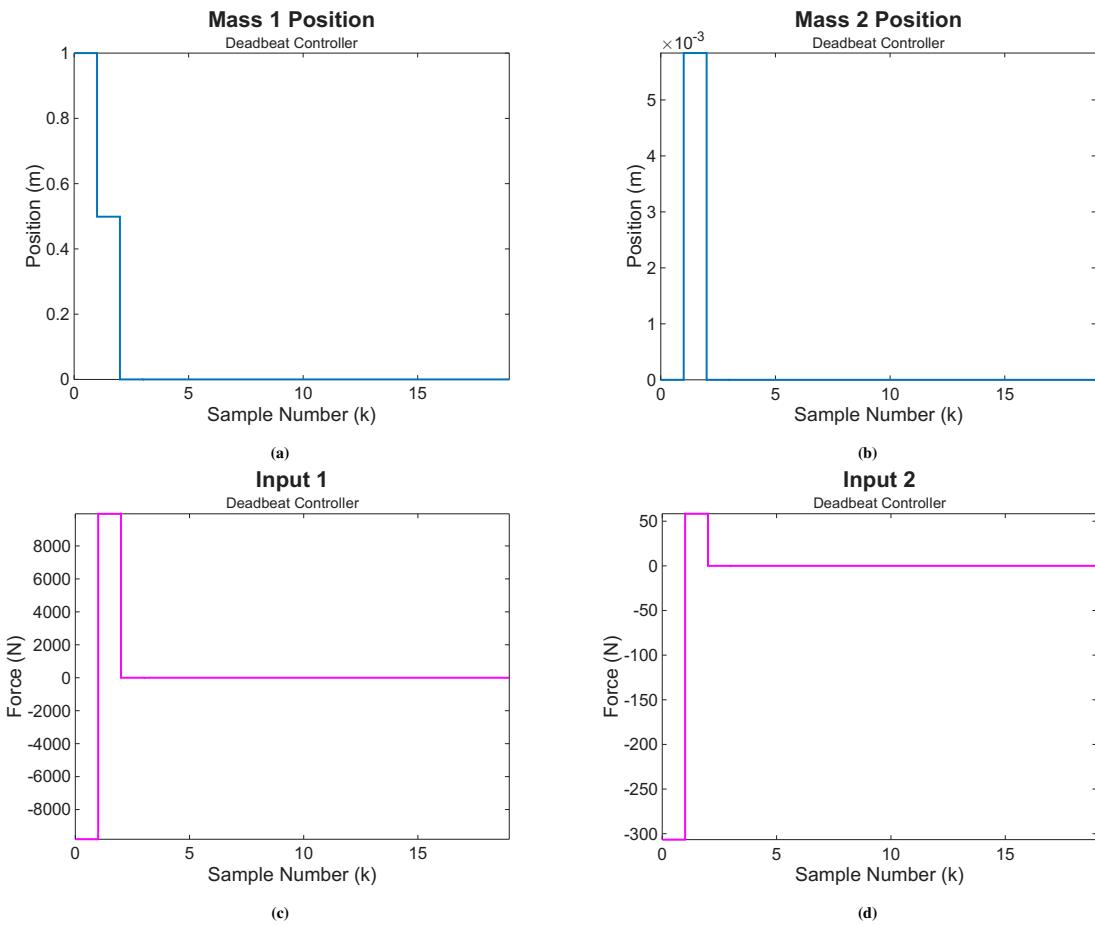


Figure 1.8: Position of Mass 1 and Mass 2 for our Dual-Spring-Mass system under a deadbeat closed-loop, state-feedback controller F . F is designed such that the poles of $(A + BF)$ are at 0. Inputs 1 and 2 are generated as $u(k) = Fx(k)$. Under deadbeat control, it can be seen that control is achieved under n steps

1.5 Linear Quadratic Regulator Controller

The LQR Controller allows for the ‘weighting’ of system states and inputs. What this means is that a controller can be designed that shies away from extreme inputs, at the expense of less controlled states, or conversely will make extreme inputs to bring a system under control.

This is done by introducing three new variables: Q , R , and γ . Q is an $n \times n$ matrix which applies relative ‘costs’ (or rewards as desired) to states of the system. Similarly, R is a $r \times r$ matrix which weights the inputs. γ is a scalar value between 0 and 1 which informs the cost function how much to discount the future versus the now (hence it is sometimes referred to as the ‘discount factor’).

Q and R are then used to define the cost function we wish to minimize. It is most common to define them as identity matrices with some associated weight, but they truly can be set as whatever so long as they are symmetric ($Q = Q^T$ and $R = R^T$). We will only use the identity approach. What each component of the cost matrices Q and R tells us is how much cost to attribute to a component of the state or input, respectively, being away from zero. It is also important to note that weights are relative: a controller defined by $Q = 100 \cdot I_{n \times n}$ and $R = 1 \cdot I_{r \times r}$ will be the exact same as one defined by $Q = 200 \cdot I_{n \times n}$ and $R = 2 \cdot I_{r \times r}$.

Each sample, k , we want to have a scalar cost as a function of our states and inputs. For a given time step, we generate a utility function $U(k)$ which produces such a scalar value.

$$U(k) = u^T(k) R u(k) + x^T(k) Q x(k) \quad (1.39)$$

In our journey to control, we will work through multiple time steps, each with their own $U(k)$, so in the whole process we will incur some net cost, J . The cost can be viewed as

the summation of all these utilities along the infinite horizon.

$$J = \sum_{k=0}^{\infty} U(k) \quad (1.40)$$

Bringing back the aforementioned discount factor γ , we modify the cost function such that we can adjust the time horizon of consequence. So long as $0 < \gamma < 1$, we can introduce it such that as k goes to ∞ , the impact of the infinite horizon utility reduces to zero⁴. This is additionally useful as we want to be able to induce stability in finite-time. This presents us with our discounted cost function

$$J = \sum_{k=0}^{\infty} \gamma^k U(k) \quad (1.41)$$

In the LQR process, we are looking for a controller that minimizes this cost function, J .

The next important idea is the Principle of Optimality. Put simply, if the optimal path from Point A to point C goes through Point B, then the optimal path from Point B to Point C is a sub-set of the path from A to C. Figure 1.9 shows a two-step process: The red path from A to C through B is optimal (with minimum cost = $4 + 6 = 10$). The principle of optimality states that if one starts from B then the optimal path to C must be the red path B-C. All other paths from B to C must cost more than 6, for example, the purple path that costs 10. The paths A-B1 and A-B2 cost less than the red path A-B, but the higher costs associated with their subsequent paths B1-C and B2-C result in higher total cost than the minimum cost. In addition, we can reason that all other paths from A to B such as the green path must cost more than 4. Otherwise, the statement that the red path A-B-C being the optimal path is contradicted⁵.

What this tells us is that no matter what state we are in for a process, so long as we make the optimal step for that current state, we will be walking along the optimal path.

⁴When $\gamma = 1$, we can only make this reduction if $U(k \rightarrow \infty)$ stabilizes and goes to 0

⁵M.Q. Phan (n.d.). “Optimal State-Space System Identification and Control”. Writing in Process.

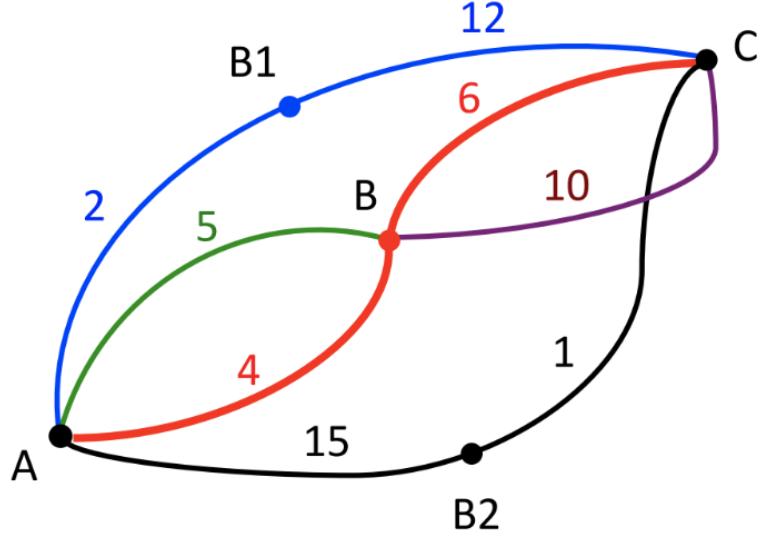


Figure 1.9: Illustration of Principle of Optimality, with nodes A , $B1$, $B2$, B , and C with paths between them labelled with their associated costs. Even though to go from $A \rightarrow B1$ is only a cost of 2, $B1 \rightarrow C$ costs 12 making the total path cost of 14. Also see that the path of $A \rightarrow B2 \rightarrow C$ may have a final step of cost 1, but the first step has a cost of 15. The central path through B then could result in costs of 15, 14, 11, or 10. Clearly going through B is the optimal way to proceed. It can then further be seen that the optimal way to go from $B \rightarrow C$ is a subset of the optimal path of $A \rightarrow C$.

It is not necessary to predict out any number of steps – by making the best decision for this moment in time, the controller will set itself up to continue to make the most optimal decisions. Solving for the Discounted LQR Controller can be quite convoluted, but it can be shown to satisfy:

$$F_{LQR}^\gamma = -\frac{1}{\sqrt{\gamma}} (B^T P B + R_\gamma)^{-1} B^T P A_\gamma \quad (1.42)$$

Where $R_\gamma = \frac{R}{\gamma}$, $A_\gamma = A\sqrt{\gamma}$, and P is the solution to the algebraic Riccati equation associated with the un-discounted LQR problem

$$P = A_\gamma^T P A_\gamma - A_\gamma^T P B (R_\gamma + B^T P B)^{-1} B^T P A_\gamma + Q \quad (1.43)$$

1.5.1 Example — LQR

It is now time to apply the logic of LQR to our system. To start, we must define our Q , R , and γ

$$Q = 100 \cdot I_{4 \times 4} \quad R = 1 \cdot I_{2 \times 2} \quad \gamma = 0.8 \quad (1.44)$$

Using the attached function *discount_LQR*⁶ will find the correct controller for the cost function (and discount factor) we use. Applying that function to our discrete system, we find

$$F_{LQR} = \begin{bmatrix} 9.9546 & -1.8952 & -2.6156 & -0.3501 \\ -25.2185 & 29.3267 & -0.8026 & -3.767 \end{bmatrix} \quad (1.45)$$

Examining where that places the poles (Figure 1.10) and the input-output data (Figures 1.11a - 1.11d), we see that control takes almost two seconds, but inputs are magnitudes smaller than that seen by the pole placement controllers.

⁶MATLAB's `dlqr` function returns a different result than the one defined by our cost function as it does not have a discount parameter.

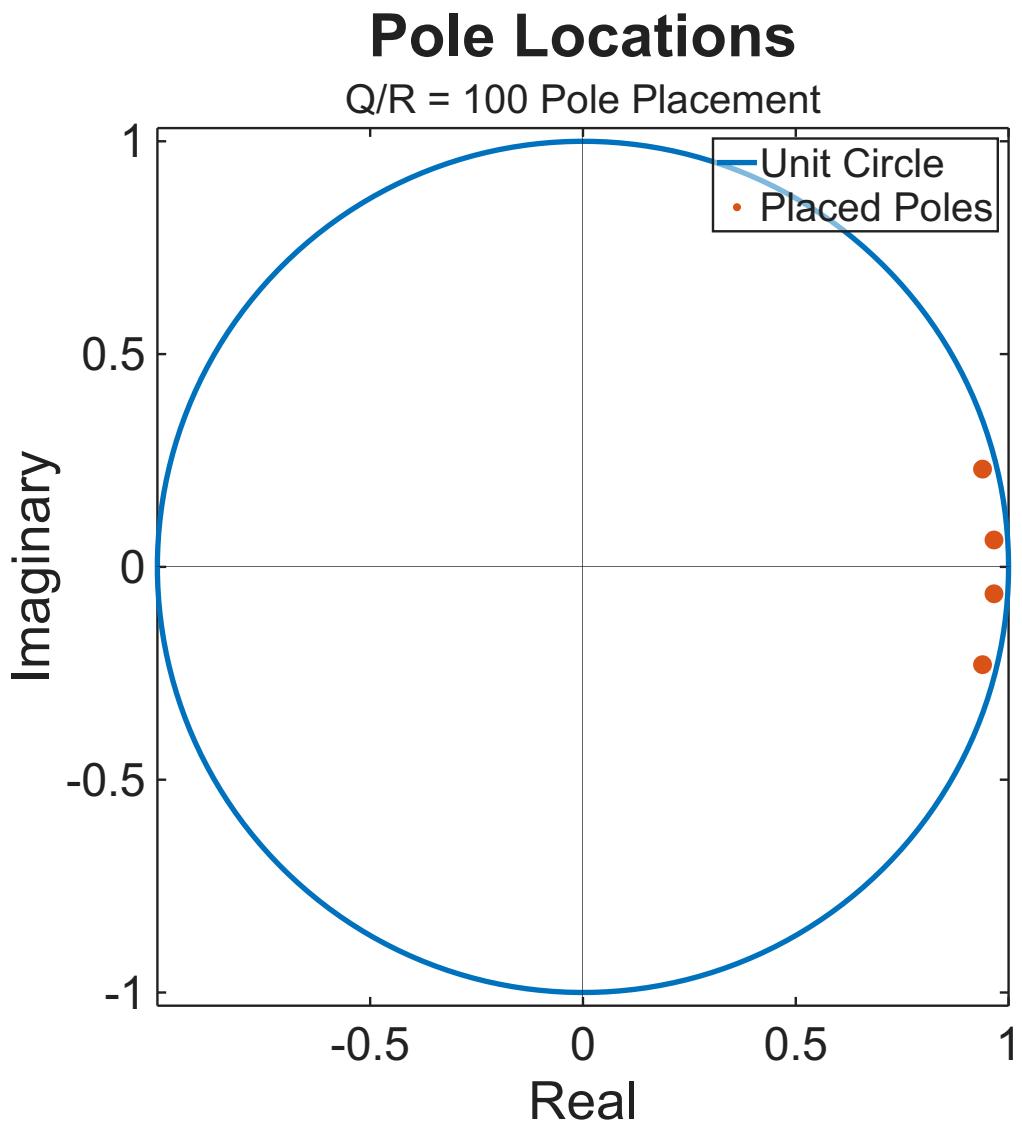


Figure 1.10: Pole Locations of a $Q/R = 100$ LQR Controller on our Dual-Spring-Mass System

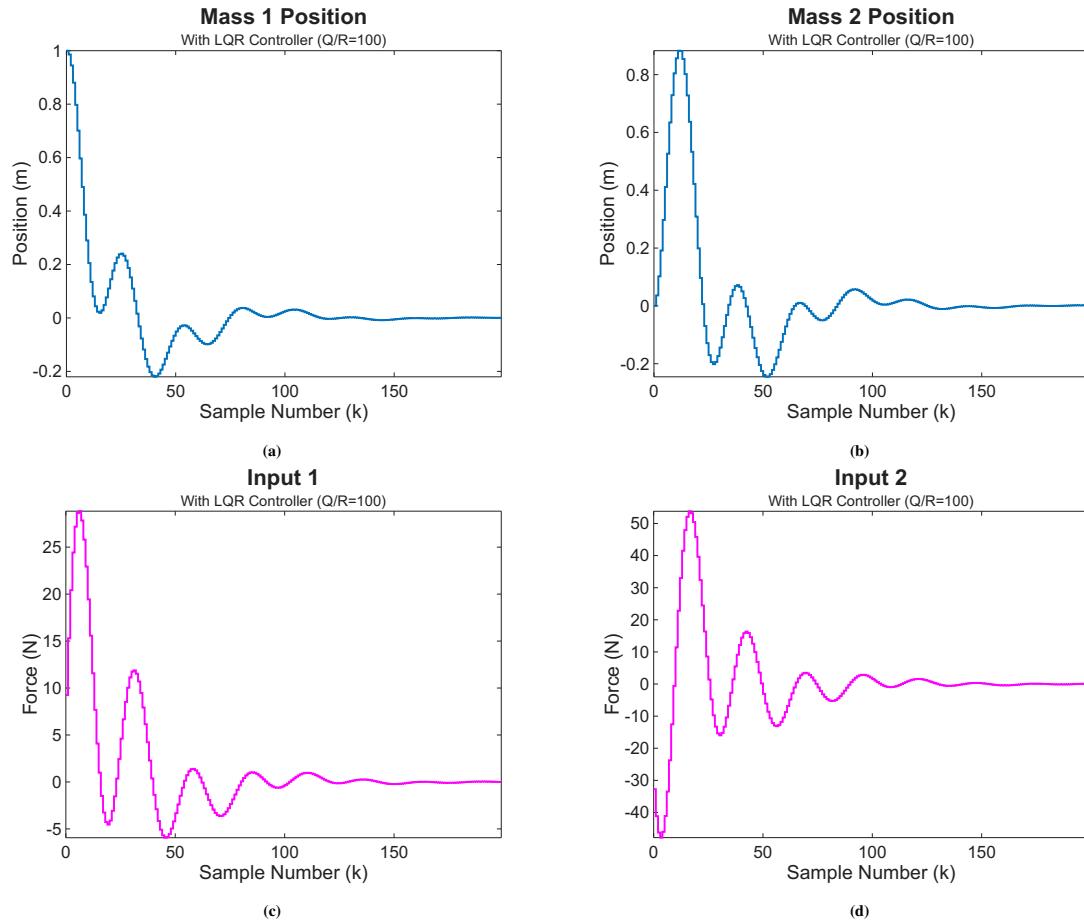


Figure 1.11: Positions of Mass 1 and 2 under a state-feedback controller defined under LQR parameters of $Q/R=100$. Control is achieved within 200 samples, and under maximum input amplitudes of 55N

If we were inclined to tweak the parameters, perhaps the inputs were beyond the capabilities of our actual system, we could easily do so. Scale R by a factor of ten (or Q by a factor of 0.1), then the following controller $F_{big\ R}$ (Eq. 1.46), poles (Figure 1.12), and IO data (Figures 1.13a - 1.13d) would result:

$$F_{big\ R} = \begin{bmatrix} 0.9118 & 0.1270 & -0.3027 & -0.0489 \\ -3.2549 & 3.8474 & -0.1148 & -0.4485 \end{bmatrix} \quad (1.46)$$

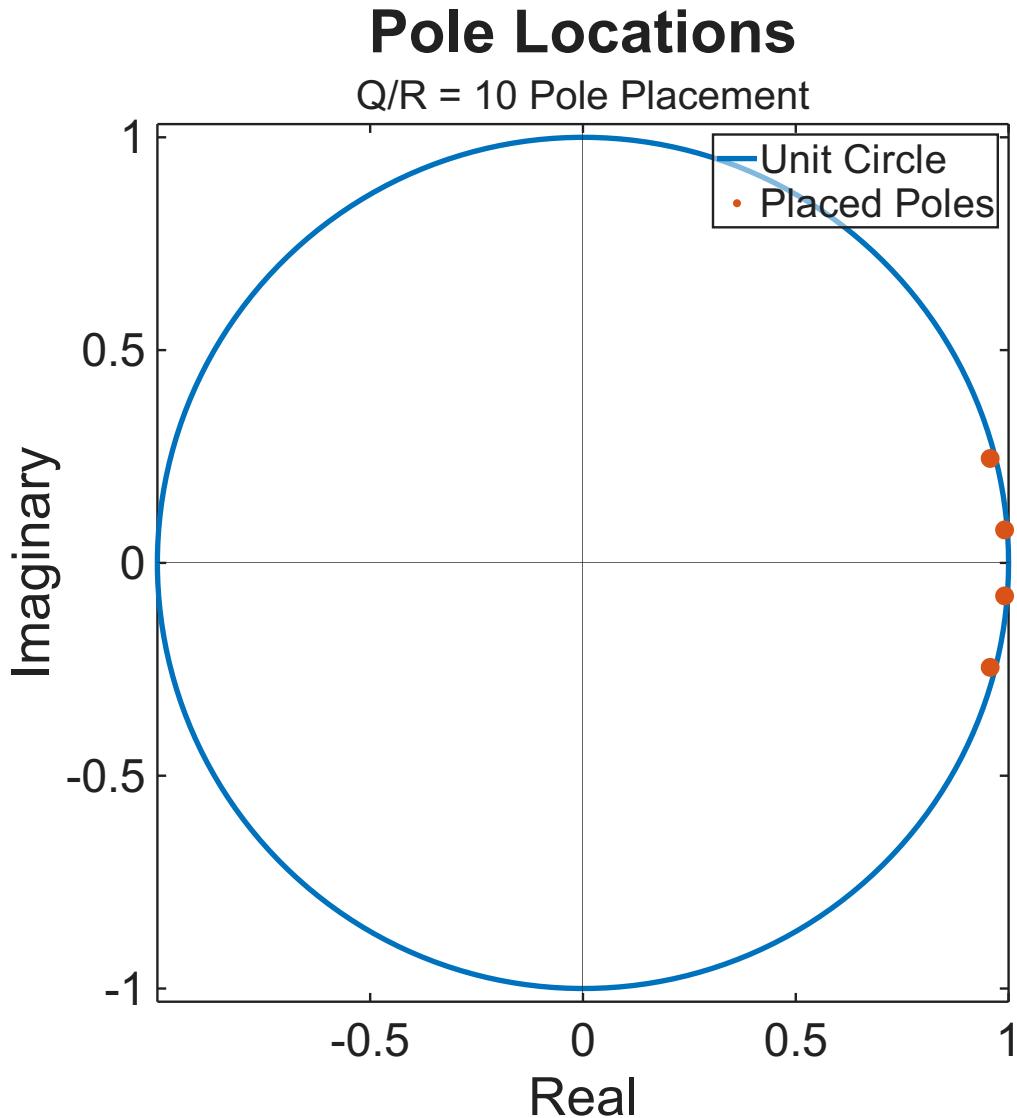


Figure 1.12: Pole Locations of a Q/R = 10 LQR Controller on our Dual-Spring-Mass System

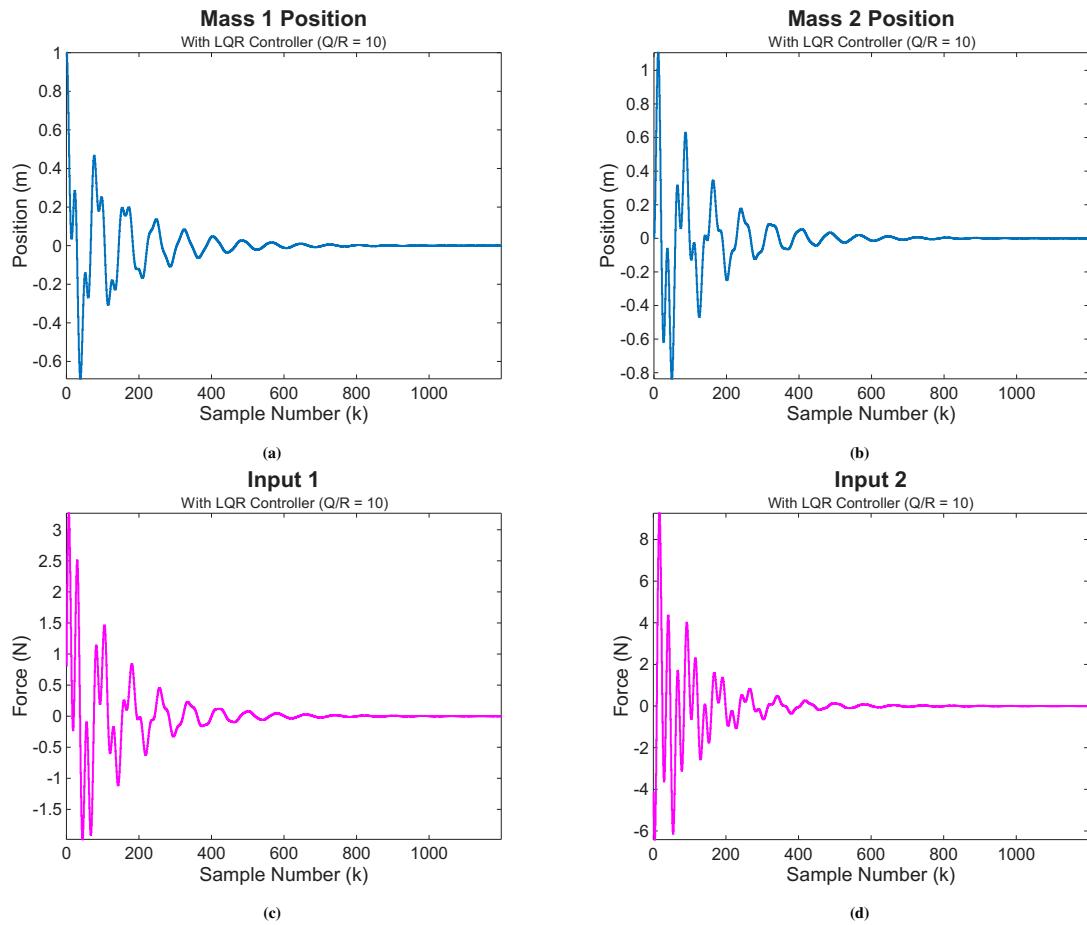


Figure 1.13: Positions of Mass 1 and 2 under a state-feedback controller defined under LQR parameters of $Q/R=10$. Control is achieved within 800 samples, and under maximum input amplitudes of 9N

It can be noted that stabilization takes much longer (around eight seconds), and the poles are much closer to the border of the unit circle. The more we reduce R , the closer our poles get to the origin of the complex plane. It is important to note that $R = 0$ does not result in a deadbeat controller. Given that every state is weighted equally, and velocity is a state, a $R = 0$ controller would seek to balance both displacement and velocity. This is best seen numerically by looking at Mass 1 of our example. It starts at $x = 1$ with no velocity – incurring a cost of 100. A deadbeat controller would attempt to bring shift the mass such that $x = 0$ in one step k , over a period of $\Delta t = 0.01$ seconds. A change in position of 1m over 0.01 seconds implies a velocity, or \dot{x}_1 , of 100. No controller that views position and velocity as equally expensive would make this decision.

1.6 Iterative Learning Control

Switching gears, we will introduce now another form of system. The previous models have all been iterative in time (i.e. adjust next input based on last samples state), but there is a form of control that focuses on trials. This is logical for / applied in the manufacturing process, where the desired output is not a ‘zero’ state, but rather zero error. ILC is particularly useful for its ability to factor out repeated noise and function to produce machined outputs. This relies on the fact that in repeated tasks the initial conditions can be made to be repeated, even if they are not explicitly known⁷⁸⁹.

Iterative Learning Control employs a system representation that is expanded to factor in the temporal element of control steps. Instead of each step (k) trying to send the states to zero, we now want each trial (j) to send the error on our outputs to zero¹⁰. The first step is

⁷Samer Said Saab et al. (2022). “Iterative Learning Control: Practical Implementation and Automation”. In: *IEEE Transactions on Industrial Electronics* 69.2, pp. 1858–1866. DOI: 10.1109/TIE.2021.3063866.

⁸,

⁹Kevin L. Moore, YangQuan Chen, and Hyo-Sung Ahn (2006). “Iterative Learning Control: A Tutorial and Big Picture View”. In: *Proceedings of the 45th IEEE Conference on Decision and Control*, pp. 2352–2357. DOI: 10.1109/CDC.2006.377582.

¹⁰D.A. Bristow, M. Tharayil, and A.G. Alleyne (2006). “A survey of iterative learning control”. In: *IEEE*

to define our output, denoted as \underline{y} , occurring over p time steps. That is, \underline{y} is a $pm \times 1$ column vector. Similarly, there is then a sequence of inputs, denoted as \underline{u} that when applied, will get us here – it will be $pr \times 1$.

Finally, there exists a matrix P that can be constructed out of A , B , C , and D matrices such that the entire output captured from an input sequence can be represented as

$$\underline{y} = P\underline{u} + \underline{d} \quad (1.47)$$

where \underline{d} captures disturbances and initial conditions. All the above matrices are formulated as such

$$\underline{y} = \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(p) \end{bmatrix} \quad \underline{u} = \begin{bmatrix} u(0) \\ u(1) \\ \vdots \\ u(p-1) \end{bmatrix} \quad (1.48)$$

$$P = \begin{bmatrix} CB & D & 0 & 0 & 0 \\ CAB & CB & D & 0 & 0 \\ CA^2B & CAB & CB & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & D \\ CA^{p-1}B & CA^{p-2}B & CA^{p-3}B & \cdots & CB \end{bmatrix} \quad (1.49)$$

$$\underline{d} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{p-1} \end{bmatrix} \quad x(0) + \omega \underline{d} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{p-1} \end{bmatrix} \quad x(0) + \omega \quad (1.50)$$

See that the \underline{d} vector factors in both how the initial conditions propagate through the system and how any additional noise ω play a role. Also note that in the ILC process we do not try

to control $y(0)$ or even model it. We cannot control initial conditions and therefore do not worry about them. As ILC is the pursuit of a desired output, we will call this goal output \underline{y}^* . From Eq. 1.47, there is a sequence of inputs \underline{u} that gets us our goal output. Call this \underline{u}^* .

$$\underline{y}^* = P \underline{u}^* + \underline{d} \quad (1.51)$$

The next step is to introduce the δ operator, signifying the difference between two value operations – this can be thought of as a discrete derivative.

$$\delta_j x = x_j - x_{j-1} \quad (1.52)$$

Applying this δ operator to Eq. 1.47

$$\delta_j \underline{y} = P \cdot \delta_j \underline{u} + \delta_j \underline{d} \quad (1.53)$$

Recognizing \underline{d} is a constant that does not change between trials allows us to drop it out of the equation to get

$$\delta_j \underline{y} = P \cdot \delta_j \underline{u} \quad (1.54)$$

Next, we define error. Each trial (j) will produce an output \underline{y}_j that will be off from our goal out of \underline{y}^* by an error denoted as

$$e_j = \underline{y}^* - \underline{y}_j \quad (1.55)$$

Applying the δ operator to this equation

$$\delta_j e = \delta_j \underline{y}^* - \delta_j \underline{y} \quad (1.56)$$

Once again we have a constant (y^*) which drops out when the delta operator is applied. So

$$\delta_j e = -\delta_j \underline{y} \quad (1.57)$$

which expands to

$$e_j - e_{j-1} = -\delta_j \underline{y} \quad (1.58)$$

To match our earlier notions of state-space models, we will increment every j value by one (allowable since they are relative indices)

$$e_{j+1} - e_j = -\delta_{j+1} \underline{y} \quad (1.59)$$

Through re-arrangement of Eq. 1.59 and substitution of Eq. 1.54, we arrive upon the ILC Equation

$$e_{j+1} = I e_j - P \delta_{j+1} \underline{u} \quad (1.60)$$

This matches our earlier A , B model except now A is the identity matrix (I), and B is the negative dynamics matrix $-P$. Additionally we now are dealing with ‘ILC States’ (n_{ILC}) and ‘ILC Inputs’ (r_{ILC}) and instead of control over samples, we control over trials. To send e_j to zero as trials go to infinity, it is then desirable to find a controller of the form

$$\delta_{j+1} \underline{u} = \mathcal{L} e_j \quad (1.61)$$

Where \mathcal{L} is $pr \times pm$ (or $r_{ILC} \times n_{ILC}$). As we have already explored the ideas of controllers, it logically follows there are an infinite number of these controllers, all facing tradeoffs.

1.6.1 Example — ILC

The first step in setting up an ILC problem is to establish the goal, or y^* . For simplicity, we will work through this example trying to draw a circle. That is, x_1 would ideally trace

out one period of a cosine, and x_2 will follow one period of a sine wave. We can set the resolution of this circle by our choice of p . Supposed we set $p = 100$, meaning we want to draw a circle over p discrete time steps. We will define the goal for our first output (the position of x_1) as y_1^* and the second output (position of x_2) as y_2^* :

$$y_1^* = \cos\left(\frac{2\pi k}{p}\right) \quad y_2^* = \sin\left(\frac{2\pi k}{p}\right) \quad (1.62)$$

Combining those into y^* produces the goal we mark each trial against. It is very important to recognize that y^* is an alternating stack of the component goals

$$\underline{y}^* = \begin{bmatrix} y_1^*(1) \\ y_2^*(1) \\ \vdots \\ y_1^*(p) \\ y_2^*(p) \end{bmatrix} \quad (1.63)$$

With our goal in hand, we now choose a controller. As earlier illustrated, the only requirement of the controller is to place the poles of the system in the unit circle. Now instead of $(A + BF)$ determining the location of our poles however, it is $(I - P\mathcal{L})$. By selecting a controller to be $\mathcal{L} = \alpha P^+$ (where $+$ denotes the pseudo inverse operation and $0 < \alpha < 1$), we can guarantee such pole placement. For the presented system, we select $\alpha = 0.8$, which we will apply for just 10 trials

$$\mathcal{L} = 0.8P^+ \quad (1.64)$$

Plotting the normalization of the error term for each trial, scaled down by the number of outputs in each trial, we can see in Figure 1.14 that the error rapidly drops to zero.

Further showing the progression of the individual outputs and inputs (Figures 1.15a - 1.15d) as well as the shaped output in Figure 1.16 (where x_1 's position is the x-axis and x_2 's position is the y-axis), you can see as the system 'learns' to draw the circle. Trial

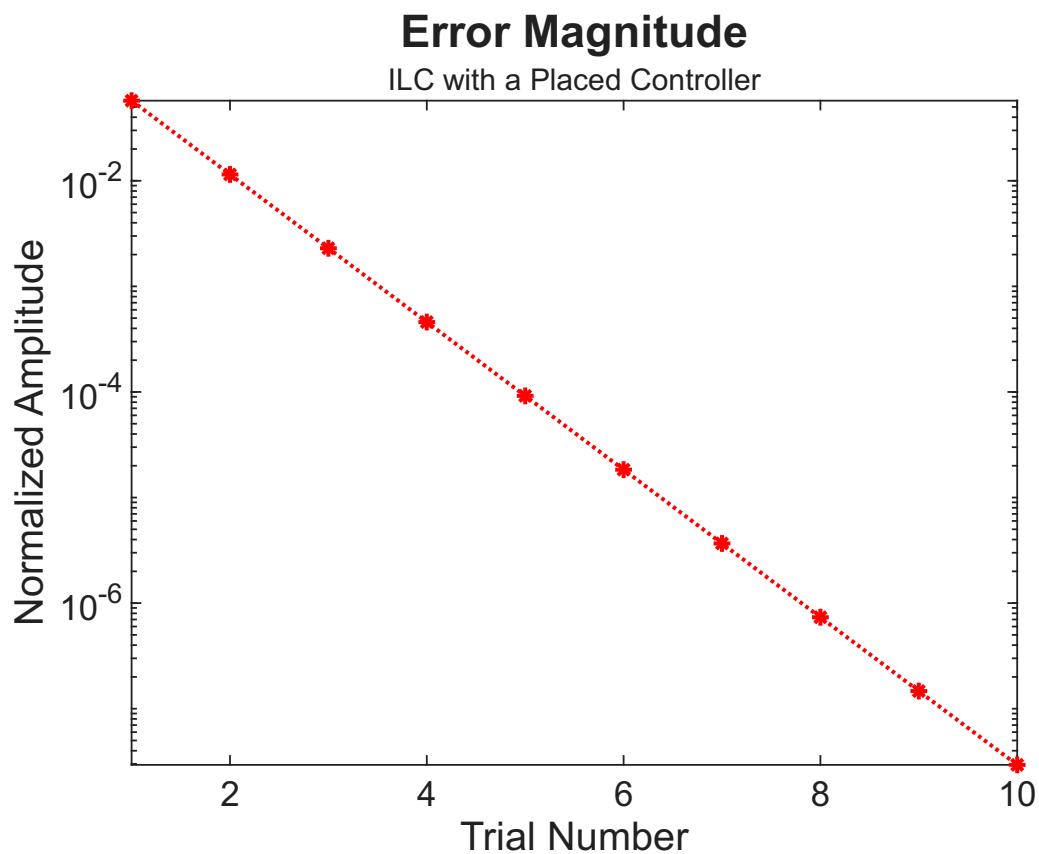


Figure 1.14: Error Progression of an ILC problem when using a perfect knowledge controller $\mathcal{L} = 0.8P^+$ such that the poles of the system under $(I - P\mathcal{L})$ are guaranteed to be within the unit circle and relatively close to the origin for rapid convergence.

1 matches our open-loop response, but even Trial 2 much more closely matches our goal (marked by the dotted red line). By trial 10, we draw our perfect circle. It is convenient here that the initial conditions of the system match those of the initial goal outputs, but we will next show that it is not necessary.

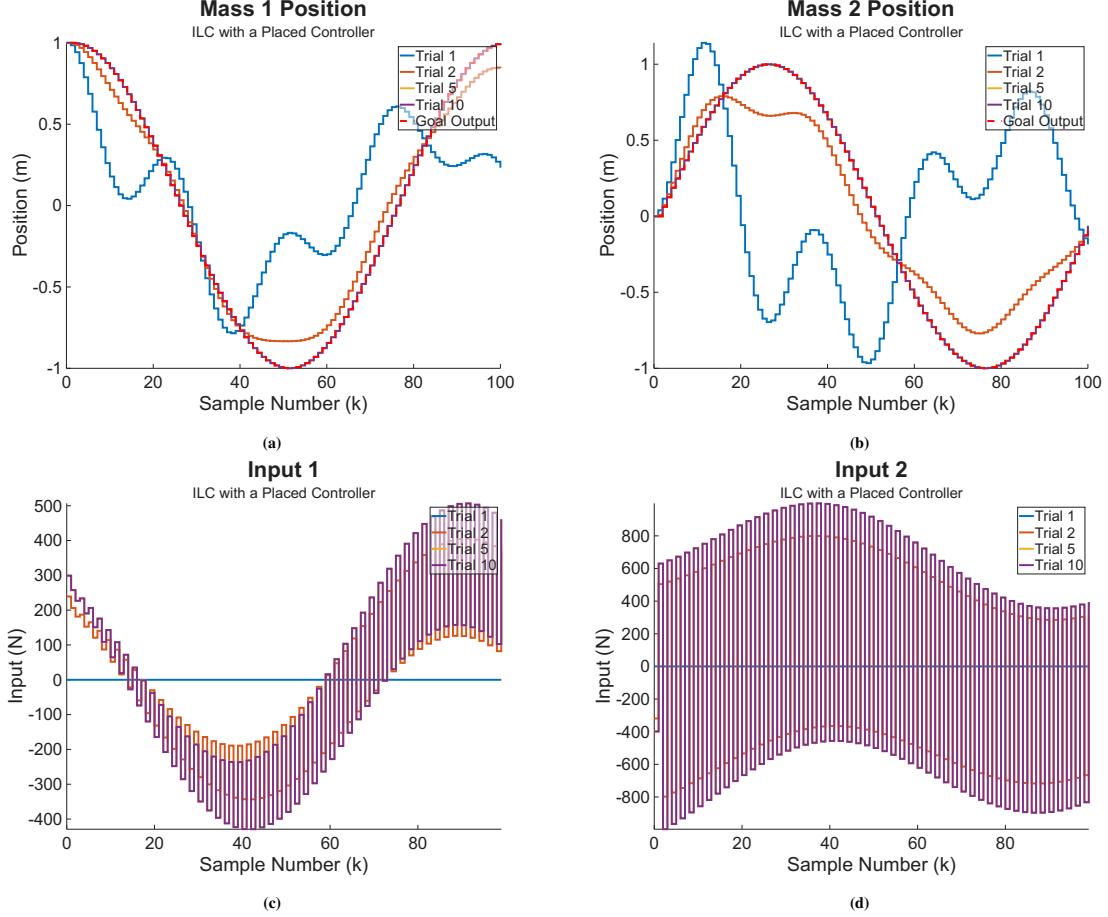


Figure 1.15: The progression of Input-Output trials under our controller of $\mathcal{L} = 0.8P^+$. Trial 1 can be seen to be the open-loop response, and by Trial 10 it can be seen that the output is captured with zero error

Shaped Outputs

ILC with a Placed Controller

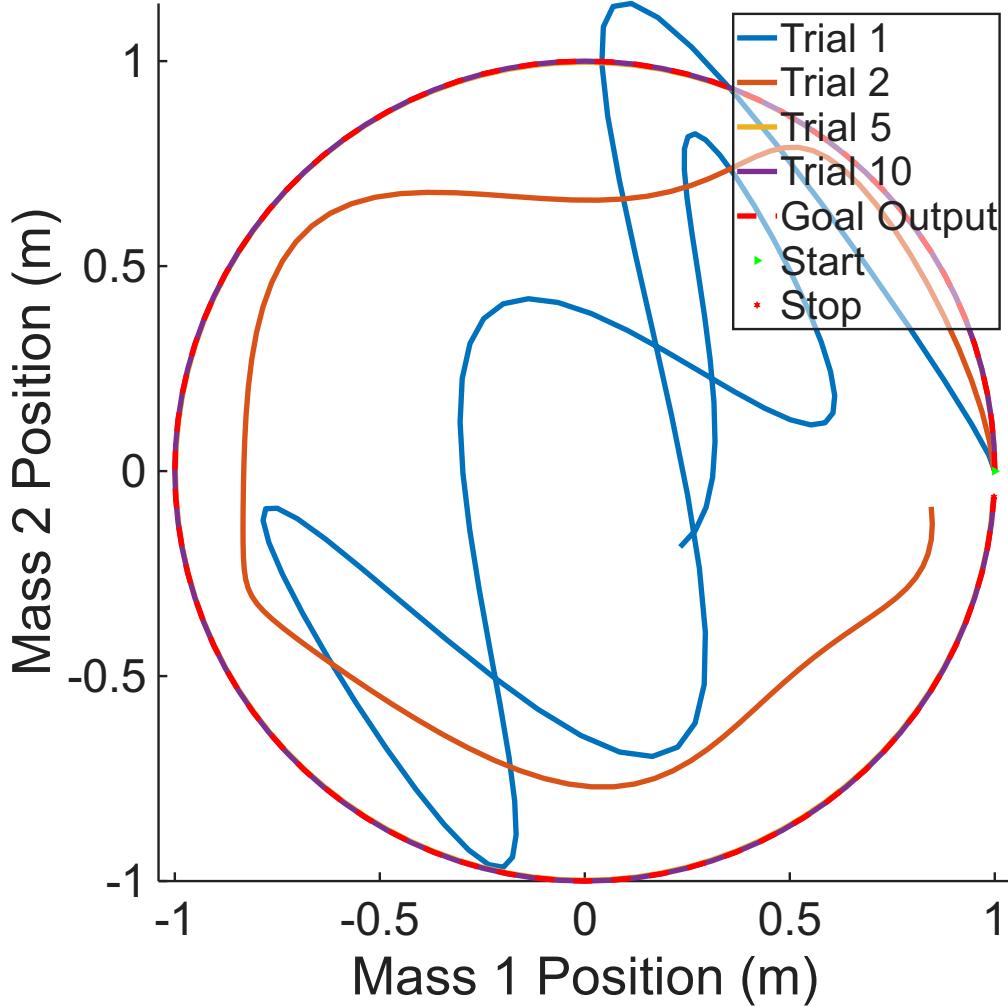


Figure 1.16: Progression of shaped outputs (where Mass 1 Position is the x-coordinate and Mass 2 Position is the y-coordinate) under our ILC controller $\mathcal{L} = 0.8P^+$

To demonstrate ILC's ability to learn arbitrary shapes¹¹, a $p = 500$ point resolution, mouse-drawn ‘Dartmouth’ is introduced as the y^* . Utilizing the same controller shown in Eq. 1.64, we can similarly learn the exact inputs required to generate our desired output. The disregard for initial conditions can be seen in the vertical line that appears - starting from $(1, 0)$ and going for the green play/triangle symbol at the top of the ‘D’ where the goal output begins.

Shaped Outputs

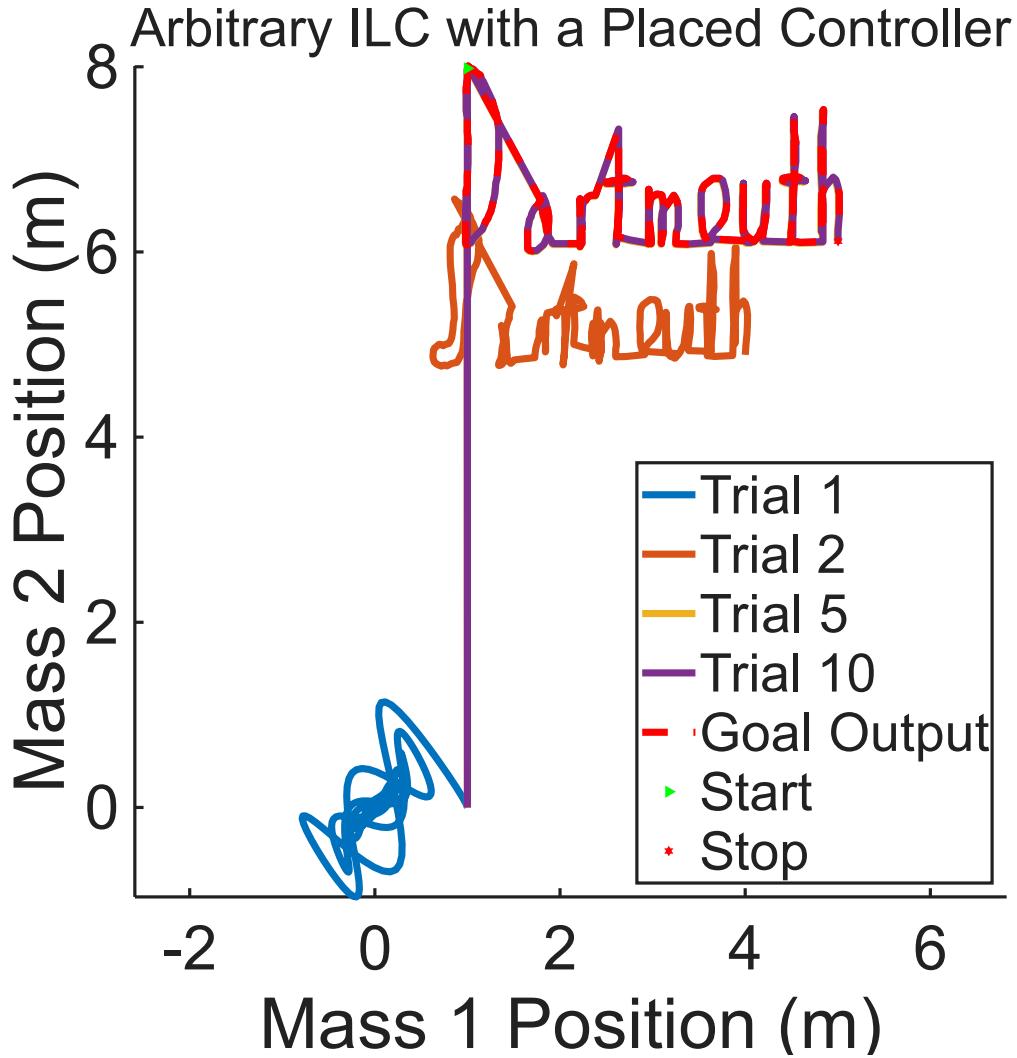


Figure 1.17: Application of ILC Controller $\mathcal{L} = 0.8P^+$ on our Dual-Spring-Mass system to learn the output ‘Dartmouth’. It can be seen that initial conditions and arbitrariness of different goals has no impact on the efficacy of ILC

¹¹To ILC, every shape is as arbitrary as the last.

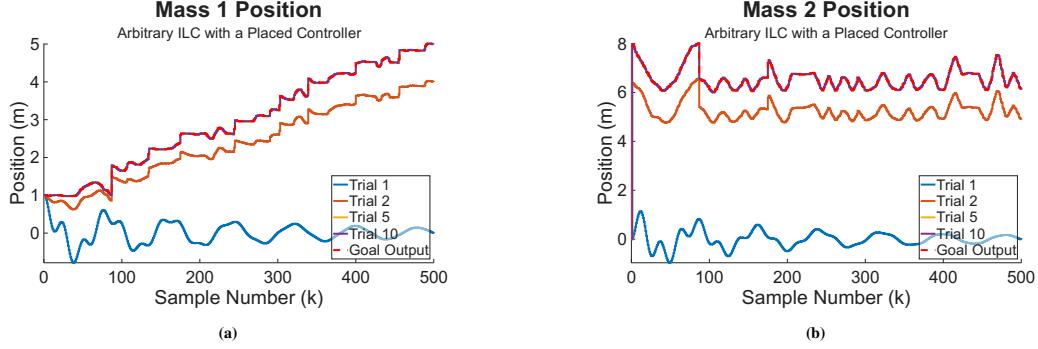


Figure 1.18: Progression of Mass 1 and 2 Positions under controller $\mathcal{L} = 0.8P^+$, learning ‘Dartmouth’. It can be seen that Mass 1, the x-position, gradually increases throughout the each trial whereas Mass 2, the y-position, simply moves back-and-forth / up-and-down

1.7 Reinforcement Learning

Until now we have been assuming that we always know the A , B , C , and D matrices. This is a bold assumption, not often matched in reality. It is then desirable to be able to construct a model-free controller for a given system. Recall our earlier cost function seen in Eq. 1.41. We can choose to restrict our time horizon down from infinity, and now look s steps ahead. We create a cost-to-go function $V(k)$:

$$V(k) = \sum_{i=0}^{s-1} \gamma^i U(k+1) = U(k) + \gamma U(k+1) + \cdots + \gamma^{s-1} U(k+s-1) \quad (1.65)$$

As from our principle of optimality, it similarly follows that whatever controller minimizes $V(k)$ will then also minimize J . This agrees with an important feature of the cost-to-go function, and that is recurrence. If we multiply Eq. 1.65 by γ and increment k by 1, we get

$$\gamma V(k+1) = \gamma U(k+1) + \cdots + \gamma^s U(k+s) \quad (1.66)$$

Substituting Eq. 1.66 into 1.65, the relationship between $V(k)$ and $V(k+1)$ can be shown

$$V(k) = \gamma V(k+1) + U(k) - \gamma^s U(k+s) \quad (1.67)$$

This is known as the recurrence equation. So long as $\gamma < 1$ and s is sufficiently large¹²,

$$V(k) = \gamma V(k+1) + U(k) \quad (1.68)$$

It can then be useful to express $V(k)$ in a supervector format for later descriptions and derivations. Substituting the utility function described in Eq. 1.39 into Eq. 1.67

$$\begin{aligned} V(k) &= u^T(k) R u(k) + x^T(k) Q x(k) \\ &\quad + \gamma u^T(k+1) R u(k+1) + \gamma x^T(k+1) Q x(k+1) \\ &\quad + \cdots + \\ &\quad + \gamma^{s-1} u^T(k+s-1) R u(k+s-1) \\ &\quad + \gamma^{s-1} x^T(k+s-1) Q x(k+s-1) \end{aligned} \quad (1.69)$$

And we can define supervectors for state and input histories

$$x_s(k) = \begin{bmatrix} x(k) \\ x(k+1) \\ \vdots \\ x(k+s-1) \end{bmatrix} \quad u_s(k) = \begin{bmatrix} u(k) \\ u(k+1) \\ \vdots \\ u(k+s-1) \end{bmatrix} \quad (1.70)$$

It is further beneficial to define matrices $\mathbf{Q}_\gamma = \Lambda_n \mathbf{Q} \Lambda_n$ and $\mathbf{R}_\gamma = \Lambda_r \mathbf{R} \Lambda_r$, where \mathbf{Q} and \mathbf{R} are $ns \times ns$ and $rs \times rs$ block-diagonal matrices comprised of s of the cost-defining matrices Q and R (respectively)

$$\mathbf{Q} = \begin{bmatrix} Q_{n \times n} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & Q_{n \times n} \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} R_{r \times r} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & R_{r \times r} \end{bmatrix} \quad (1.71)$$

¹²Or $U(k+s)$ is stable.

and the Λ matrices are defined as

$$\Lambda_n = \begin{bmatrix} I_{n \times n} & & & \\ & \sqrt{\gamma} I_{n \times n} & & \\ & & \ddots & \\ & & & (\sqrt{\gamma})^{s-1} I_{n \times n} \end{bmatrix} \quad (1.72)$$

$$\Lambda_r = \begin{bmatrix} I_{r \times r} & & & \\ & \sqrt{\gamma} I_{r \times r} & & \\ & & \ddots & \\ & & & (\sqrt{\gamma})^{s-1} I_{r \times r} \end{bmatrix} \quad (1.73)$$

Now Eq. 1.69 can be re-written as

$$V(k) = u_s^T(k) \mathbf{R}_\gamma u_s(k) + x_s^T(k) \mathbf{Q}_\gamma x_s(k) \quad (1.74)$$

The above formulations make it easier to represent our Q-function. The Q-function is defined by the current state and input of a system and defined with respect to a controller F . Its logic is as follows: suppose we are at state $x(k)$ and have just made input $u(k)$ – both can be arbitrary. However, from time $k+1$ to infinity, our next state ($x(k+1)$, $x(k+2)$, \dots) will be a function of our previous state and input, as described in Eq. 1.27. And our inputs ($u(k+1)$, $u(k+2)$, \dots) will follow the control law described in Eq. 1.31. Thus the vector $x_s(k)$ can be expressed in terms of the current state $x(k)$, the current input $u(k)$, and all future inputs to $u(k+s-1)$ as

$$x_s(k) = P_1 x(k) + P_2 u_s(k) \quad (1.75)$$

where

$$P_1 = \begin{bmatrix} I \\ A \\ \vdots \\ A^{s-1} \end{bmatrix} \quad P_2 = \begin{bmatrix} 0 & & & \\ B & 0 & & \\ \vdots & \ddots & \ddots & \\ A^{s-2}B & \cdots & B & 0 \end{bmatrix} \quad (1.76)$$

Substituting these into our cost-to-go expression of Eq. 1.74, we get a cost-to-go defined only in terms of present state and all inputs from now until s

$$V(k) = u_s^T(k) \mathbf{R}_\gamma u_s(k) + [P_1 x(k) + P_2 u_s(k)]^T \mathbf{Q}_\gamma [P_1 x(k) + P_2 u_s(k)] \quad (1.77)$$

If we define the symmetric matrix \mathbf{S} as

$$\mathbf{S} = \begin{bmatrix} P_1^T \mathbf{Q}_\gamma P_1 & P_1^T \mathbf{Q}_\gamma P_2 \\ P_2^T \mathbf{Q}_\gamma P_1 & \mathbf{R}_\gamma + P_2^T \mathbf{Q}_\gamma P_2 \end{bmatrix} \quad (1.78)$$

We can once again reduce the cost-to-go function to

$$V(k) = \begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix}^T \mathbf{S} \begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix} \quad (1.79)$$

As previously mentioned, all inputs after k will follow the control law $u(k) = Fx(k)$, and so it must be possible to further reduce our representation. Start with

$$u_s(k) = \begin{bmatrix} u(k) \\ u_{s-1}(k+1) \end{bmatrix} \quad u_{s-1}(k+1) = \begin{bmatrix} u(k+1) \\ u(k+2) \\ \vdots \\ u(k+s-1) \end{bmatrix} \quad (1.80)$$

Given our system model and control law, all future inputs can be tracked back to $x(k)$ and

$u(k)$. By repeated substitution, it can be shown

$$\begin{aligned} u(k+1) &= Fx(k+1) = F[Ax(k) + Bu(k)] \\ u(k+2) &= Fx(k+2) = F(A+BF)[Ax(k) + Bu(k)] \\ &\vdots \\ u(k+s-1) &= F(A+BF)^{s-2}[Ax(k) + Bu(k)] \end{aligned} \quad (1.81)$$

We can now rewrite 1.80 as

$$u_{s-1}(k+1) = F_{xx}(k) + F_{uu}(k) \quad (1.82)$$

where

$$F_x = \begin{bmatrix} FA \\ F(A+BF)A \\ \vdots \\ F(A+BF)^{s-2}A \end{bmatrix} \quad F_u = \begin{bmatrix} FB \\ F(A+BF)B \\ \vdots \\ F(A+BF)^{s-2}B \end{bmatrix} \quad (1.83)$$

Then

$$\begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix} = \begin{bmatrix} x(k) \\ u(k) \\ u_{s-1}(k+1) \end{bmatrix} = \begin{bmatrix} I_{n \times n} & 0_{n \times r} \\ 0_{r \times n} & I_{r \times r} \\ F_x & F_u \end{bmatrix} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \quad (1.84)$$

Defining \mathbf{R}

$$\mathbf{R} = \begin{bmatrix} I_{n \times n} & 0_{n \times r} \\ 0_{r \times n} & I_{r \times r} \\ F_x & F_u \end{bmatrix} \quad (1.85)$$

We can now write

$$\begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix} = \mathbf{R} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \quad (1.86)$$

If we create matrix¹³ $\mathbf{P} = \mathbf{R}^T \mathbf{S} \mathbf{R}$, we can substitute Eq. 1.86 into Eq. 1.79 to get a new cost-to-go function

$$Q(x(k), u(k)) = \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \mathbf{P} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \quad (1.87)$$

We may now utilize this formulation, along with the recurrence equation, to find the optimal Q-function. As the Q-function is defined for a given controller F , and we constructed $Q(x(k), u(k))$ in our observance with the cost function that defined our LQR controller, the Q-function will be optimized by the LQR controller. The optimal controller F will produce $u(k+1)$ that minimizes our cost-to-go, now defined in Eq. 1.87, starting from $x(k)$. That is, it will give us input $u(k+1)$ defined as:

$$u(k+1) = \arg \min_{u(k+1)} Q(x(k+1), u(k+1)) \quad (1.88)$$

A comment on the ‘argmin’ function: this operator will give us the value of the specified argument that will minimize the given function. So in our case, the $u(k+1)$ that will minimize the cost-to-go function $Q(x(k+1), u(k+1))$ is returned. Therefore it follows that the Q-function that utilizes this minimizing input will equal the minimum possible value for the Q-function

$$Q(x(k+1), \arg \min_{u(k+1)} Q(x(k+1), u(k+1))) = \min_{u(k+1)} Q(x(k+1), u(k+1)) \quad (1.89)$$

So we can now write the recurrence equation for the deterministic Q-Learning-based RL method. By taking Eq. 1.68, substituting in our new cost-to-go defined in Eq. 1.87 and the logic in Eq. 1.89, we arrive upon the relationship that the optimal Q-function (as defined by the optimal controller) must satisfy:

$$Q(x(k), u(k)) = \gamma \min_{u(k+1)} Q(x(k+1), u(k+1)) + U(k) \quad (1.90)$$

¹³Note this is not the same P matrix defined in the ILC problem

Any Q-function will satisfy

$$Q(x(k), u(k)) = \gamma Q(x(k+1), u(k+1)) + U(k) \quad (1.91)$$

but only Eq. 1.90 is satisfied by the optimal Q-function, as defined by the optimal controller F .

To extract the controller from a given Q-function, we return to Eq. 1.87. Recall matrix \mathbf{P} is symmetric. We can re-write it as

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xu} \\ \mathbf{P}_{xu}^T & \mathbf{P}_{uu} \end{bmatrix} \quad (1.92)$$

Where the x and u subscript indicate the size of each \mathbf{P} component. \mathbf{P} is $(n+r) \times (n+r)$, so \mathbf{P}_{xx} refers to the top-left $n \times n$ portion, and the same logic follows for the other components. We can extract controller F as

$$F = -(\mathbf{P}_{uu})^{-1} (\mathbf{P}_{xu}^T) \quad (1.93)$$

Thus we have shown from a Q-function we can extract its controller, and given that we have an equation of recurrence that defines the optimal Q-function, we can begin to solve for the optimal Q-function purely from system input-output data.

1.7.1 Policy Iteration

The first method which we will demonstrate is that of Policy Iteration. To do this, we need data triplets of $x(k), u(k)$ and $x(k+1)$. By manipulating enough of these triplets, we can episodically solve for the \mathbf{P} that parametrizes a Q-function, while simultaneously

optimizing the Q-function. We begin with Eq. 1.91, but re-arrange it to

$$Q(x(k), u(k)) - \gamma Q(x(k+1), u(k+1)) = U(k) \quad (1.94)$$

It is next necessary to find a way to re-write $Q(x(k), u(k))$. We will start by defining the stack operator for a matrix. Given an arbitrary matrix H that is $v \times w$, the stack operator creates H^s that is a single column, making our matrix $vw \times 1$. So if

$$H = \begin{bmatrix} h_1 & h_2 & \cdots & h_w \end{bmatrix} \quad (1.95)$$

Where each h_i is $v \times 1$, then

$$H^s = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_w \end{bmatrix} \quad (1.96)$$

It is important to recognize that the stack operator is not the transpose operation, as each h_i is a vector, not a scalar. Our next operator is the Kronecker product, marked \otimes . This operator is used to multiply two matrices in a way such that each component of one is used to scale the entirety of the second. So for an $m \times n$ matrix A , and a $p \times q$ matrix B

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} \quad (1.97)$$

The Kronecker product between the two will create an $mp \times nq$ matrix

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \quad (1.98)$$

Now to see how these operators will be useful, we return to Eq. 1.87, re-written below as

$$Q(x(k), u(k)) = \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \mathbf{P} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \quad (1.87)$$

To better demonstrate the process, we will assume $x(k)$ and $u(k)$ are scalar, and \mathbf{P} is thus a 2×2 matrix.

$$x(k) = x_1 \quad u(k) = u_1 \quad \mathbf{P} = \begin{bmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{21} & \mathbf{P}_{22} \end{bmatrix} \quad (1.99)$$

Manually working out Eq. 1.87, we can re-write it as

$$\begin{aligned} Q(x(k), u(k)) &= \begin{bmatrix} x_1 & u_1 \end{bmatrix} \begin{bmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{21} & \mathbf{P}_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ u_1 \end{bmatrix} \\ &= \begin{bmatrix} x_1 \mathbf{P}_{11} + u_1 \mathbf{P}_{21} & x_1 \mathbf{P}_{12} + u_1 \mathbf{P}_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ u_1 \end{bmatrix} \\ &= x_1^2 \mathbf{P}_{11} + x_1 u_1 \mathbf{P}_{21} + x_1 u_1 \mathbf{P}_{12} + u_1^2 \mathbf{P}_{22} \end{aligned} \quad (1.100)$$

It can be shown that the results of Eq. 1.100 can then be expressed by stacking \mathbf{P} as

$$Q(x(k), u(k)) = \begin{bmatrix} x_1^2 & x_1 u_1 & x_1 u_1 & u_1^2 \end{bmatrix} \begin{bmatrix} \mathbf{P}_{11} \\ \mathbf{P}_{21} \\ \mathbf{P}_{12} \\ \mathbf{P}_{22} \end{bmatrix} \quad (1.101)$$

It is easy to now see where the stack operator will come into play in the second matrix. Thus it comes down to reducing the first matrix. We can see

$$\begin{aligned} \begin{bmatrix} x_1^2 & x_1 u_1 & x_1 u_1 & u_1^2 \end{bmatrix} &= \begin{bmatrix} x_1 & u_1 \end{bmatrix} \otimes \begin{bmatrix} x_1 & u_1 \end{bmatrix} \\ &= \begin{bmatrix} x_1 \\ u_1 \end{bmatrix}^T \otimes \begin{bmatrix} x_1 \\ u_1 \end{bmatrix}^T \end{aligned} \quad (1.102)$$

Putting it all together, we can re-write Eq. 1.100 as

$$Q(x(k), u(k)) = \left[\begin{bmatrix} x_1 \\ u_1 \end{bmatrix}^T \otimes \begin{bmatrix} x_1 \\ u_1 \end{bmatrix}^T \right] \mathbf{P}^S \quad (1.103)$$

This was just demonstrated in the scalar case, but can be similarly proven for when $x(k)$ is $n \times 1$ and $u(k)$ is $r \times 1$. Thus we can write our Q-Function as

$$Q(x(k), u(k)) = \left[\begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \otimes \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \right] \mathbf{P}^S \quad (1.104)$$

For controller F_j , we have a given Q-function parametrized by \mathbf{P}_j . Recall that $x(k)$ and $u(k)$ can be arbitrary¹⁴, $x(k+1)$ will be produced by nature/the system, and all inputs from $k+1$ onward are defined by our control law. Thus we can re-write Eq. 1.94 as

$$\left[\begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \otimes \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T - \gamma \begin{bmatrix} x(k+1) \\ F_j x(k+1) \end{bmatrix}^T \otimes \begin{bmatrix} x(k+1) \\ F_j x(k+1) \end{bmatrix}^T \right] \mathbf{P}_j^S = U(k) \quad (1.105)$$

¹⁴For learning, $u(k)$ is best when randomized. If following a control-law process, added exploration terms are needed (see example)

To simplify equations, write $X_j(k)$

$$X_j(k) = \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \otimes \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T - \gamma \begin{bmatrix} x(k+1) \\ F_j x(k+1) \end{bmatrix}^T \otimes \begin{bmatrix} x(k+1) \\ F_j x(k+1) \end{bmatrix}^T \quad (1.106)$$

$X_j(k)$ will then be dimensions $1 \times (n+r)^2$. From each $x(k), u(k)$, and $x(k+1)$ triplet, we can produce an $X_j(k)$ and $U(k)$ pair. Stacking those we can construct

$$\begin{bmatrix} X_j(k) \\ X_j(k') \\ X_j(k'') \\ \vdots \end{bmatrix} \mathbf{P}_j^S = \begin{bmatrix} U(k) \\ U(k') \\ U(k'') \\ \vdots \end{bmatrix} \quad (1.107)$$

Where k, k', k'', \dots do not need to be consecutive (but logically will be). With sufficient data samples, we can then solve for \mathbf{P}_j^S as

$$\mathbf{P}_j^S = \begin{bmatrix} X_j(k) \\ X_j(k') \\ X_j(k'') \\ \vdots \end{bmatrix}^+ \begin{bmatrix} U(k) \\ U(k') \\ U(k'') \\ \vdots \end{bmatrix} \quad (1.108)$$

In the noise free scenario, we need $(n+r)^2$ collections to solve for \mathbf{P}_j^S . Anything less, our matrix will be poorly conditioned and the pseudo-inverse operator will have more than one solution – unlikely to be optimal. By unstacking \mathbf{P}_j^S , we can update controller F_j using Eq. 1.93 such that iteratively

$$F_{j+1} = -(\mathbf{P}_{uu})^{-1} (\mathbf{P}_{xu}^T) \quad (1.109)$$

Since Eq. 1.105 is derived from the optimal condition outlined in Eq. 1.90, the controller we derived must also be optimal. It may take several iterations, but this process will alternate updating \mathbf{P} and F until the optimal controller is found.

Example — Policy Iteration

We now return to our earlier spring-mass system shown in Figure 1.1. Operating off the same parameters outlined in Eqs. 1.44 which produced the F_{LQR}^γ shown in Eq. 1.45 as

$$F_{LQR}^\gamma = \begin{bmatrix} 9.9546 & -1.8952 & -2.6156 & -0.3501 \\ -25.2185 & 29.3267 & -0.8026 & -3.767 \end{bmatrix} \quad (1.45)$$

Our system has four states ($n = 4$) and two inputs ($r = 2$). As such, each $X_j(k)$ will be 1×36 ($(n+r)^2 = (4+2)^2$). That means that each controller ‘update’ marked by the process shown in Eq. 1.108 requires 36 triplets of state, input, and next state data $(x(k), u(k), x(k+1))$. Knowing this, we will set parameters dictating the number of controllers we will try and the number of data points we will collect per controller. We must also define a controller to start with, which we will set to be all zeros. For every sample k , we first compute our $u(k)$. When learning, it is not enough to just use our classic $u(k) = Fx(k)$; we must add some random excitation term for exploration. The mathematical reason for this is to ensure that we construct sufficient linear-independent X_j s to allow for a proper solving of \mathbf{P}_j . Intuitively - how can one expect to learn by not trying something new every now and again?

$$u(k) = Fx(k) + v(k) \quad (1.110)$$

Where $v(k)$ is some exploration term intentionally added to the classic $u(k)$. This is not noise, as we do not know noise – this is generated, known, and added by us to learn. Due to the arbitrary nature of the triplets, it is also possible to make the input purely random

and not based in any way on the current state. The input under that approach would be

$$u(k) = v(k) \quad (1.111)$$

It is important to choose an exploration magnitude relative to the impact of inputs. For this system where inputs map relatively directly to a change of states, we set the range of values to be from $[-1, 1]$.

Whichever input approach we chose, we then apply it to the system in state $x(k)$. Nature then produces $x(k+1)$ for us. We now have our $x(k)$, $u(k)$, $x(k+1)$ triplet. Following Eq. 1.106 we formulate $X_j(k)$. Note that the $F_j x(k+1)$ term does not include an exploration term. At the same time, we will compute the utility $U(k)$ as defined in Eq. 1.39

For our system, we will repeat this process 35 more times before we can update the controller once. We compute \mathbf{P}_j^S from Eq. 1.108, and undo the stack operator by reshaping it into a 6×6 matrix. The way in which we do this does not matter (rows to column or column to rows) as \mathbf{P}_j is symmetric. Numerical operators are not exact, however, and we can accelerate the learning process by imposing symmetry. That is, after computing a P_j which is semi-symmetric, we set \mathbf{P}_j as

$$\mathbf{P}_j = \frac{1}{2} (P_j + P_j^T) \quad (1.112)$$

Now we refer to Eq. 1.93 to extract the components to solve for our next controller as shown in Eq. 1.109. In this example, we grab the bottom right 2×2 block of \mathbf{P}_j as \mathbf{P}_{uu} , and the bottom left 2×4 block as \mathbf{P}_{xu}^T . We then use those parameters to update our controller, and repeat the process. After you have iterated through all the controllers you wish to learn, it is often useful to run out a few trials without an exploration term on the input to verify to yourself that your controller does indeed work. The system will stabilize as you go under this approach, but only so much when the input is distorted. In this case,

after five controllers (of 36 trials each) we produce the controller F_{policy}

$$F_{policy} = \begin{bmatrix} 9.9546 & -1.8952 & -2.6156 & -0.3501 \\ -25.2185 & 29.3267 & -0.8026 & -3.767 \end{bmatrix} \quad (1.113)$$

which matches our F_{LQR}^γ exactly to at least 4 decimal places. Its application can be seen in Figures 1.19a - 1.19d Notice how even for the first 36 trials there is variation on the input

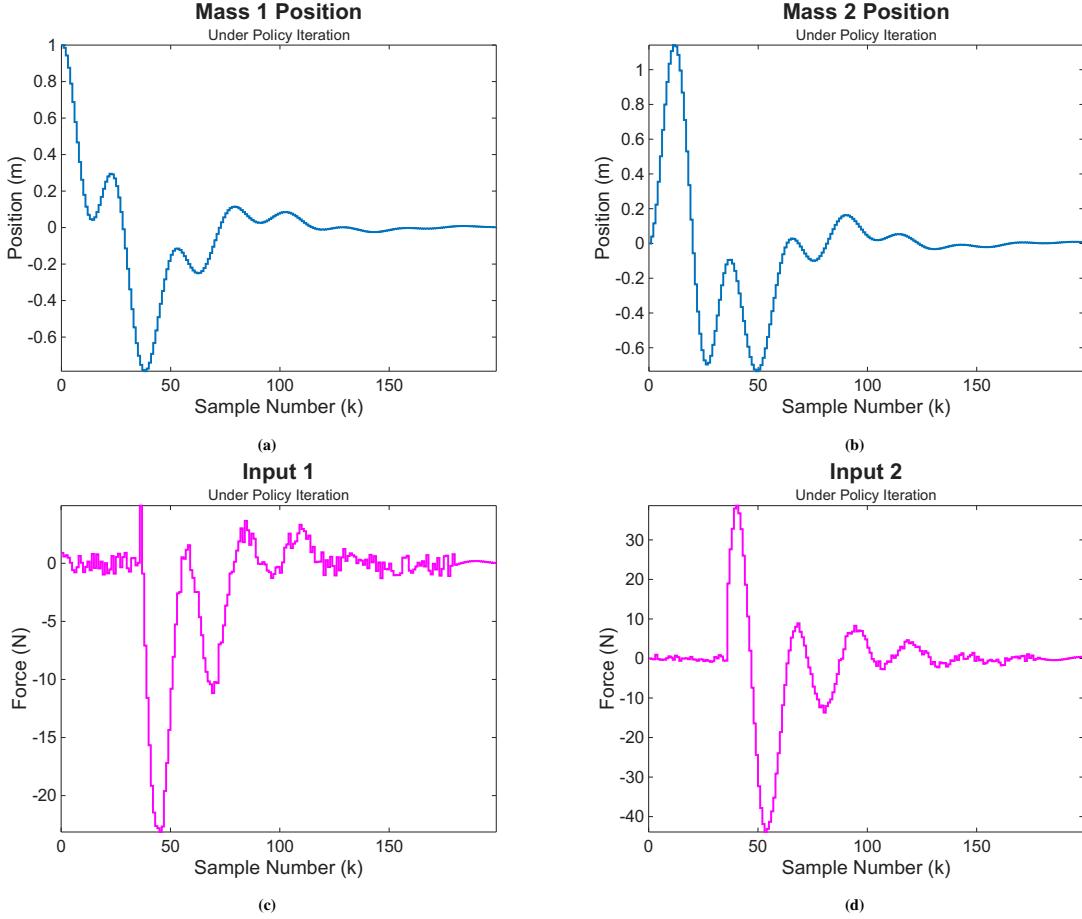


Figure 1.19: Input-Output Data of our Dual-Spring-Mass system under 5 Policy Iteration Trials of 36 samples/steps each. Learning parameters of $Q/R = 100$ and $\gamma = 0.8$. After 5 controllers, the learning stops and exploration $v(k)$ is no longer applied to the input for the final 20 trials.

due to the exploration term, and the final 20 trials are much smoother as they follow a strict control law without exploration. In Figures 1.20a and 1.20b we can see how the various parameters converged through the learning process. Each figure corresponds to a different input / controller row, and different lines are the impact that each state has on the input.

After just two trials we see we almost perfectly capture our end controller, which ends up matching the F_{LQR}^γ .

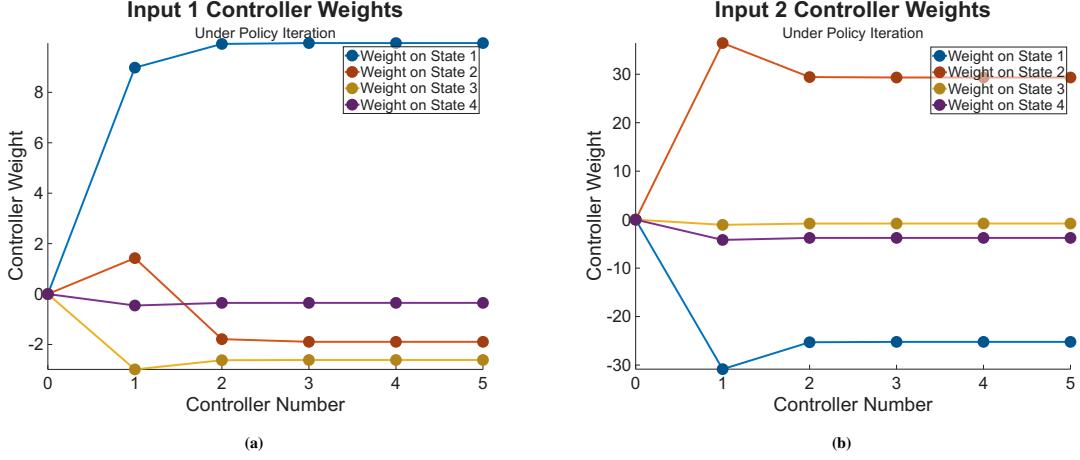


Figure 1.20: Progression of controller weights through Policy Iteration Trials. For two inputs, there are two rows of the controller F to describe how to weight their respective inputs from the associated samples collected states.

1.7.2 Input Decoupling

In Policy Iteration, we learn every input at the same time. Input Decoupling allows us to learn one input at a time, reducing the number of collections per controller from $(n + r)^2$ to $(n + 1)^2$, but at the cost of needing to complete r times as many learning trials. It can be easily shown that when $n^2 \geq r$, Policy Iteration learns faster / in less trials. Input Decoupling will always reduce the number of trials needed for one input to learn, but rarely the whole controller. However, no learning-optimality is lost, and it is often that control on one input sends all states to zero (though at a sub-optimal rate). Recall our cost-to-go function (as defined in Eq. 1.87)

$$Q(x(k), u(k)) = \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \mathbf{P} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \quad (1.87)$$

For our r -input problem, we can express $u(k)$ as a stack of each individual input

$$u(k) = \begin{bmatrix} u_1(k) \\ u_2(k) \\ \vdots \\ u_r(k) \end{bmatrix} \quad (1.114)$$

Which can be produced by our similarly-represented stacked controller

$$F = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_r \end{bmatrix} \quad (1.115)$$

Where each F_i is a $1 \times n$ vector. In general, for input $u_i(k)$, we can re-write Eq. 1.114 and 1.115 as

$$u(k) = \begin{bmatrix} u_{Ti}(k) \\ u_i(k) \\ u_{Bi}(k) \end{bmatrix} \quad (1.116)$$

$$F = \begin{bmatrix} F_{Ti} \\ F_i \\ F_{Bi} \end{bmatrix} \quad (1.117)$$

Where subscripts T and B represent the top t elements and bottom b elements of $u(k)$ and F . $t + 1 + b = r$, by definition. Defining matrix \mathbf{G}_i

$$\mathbf{G}_i = \begin{bmatrix} I_{n \times n} & 0_{n \times 1} \\ F_{Ti} & 0_{t \times 1} \\ 0 & 1 \\ F_{Bi} & 0_{b \times 1} \end{bmatrix} \quad (1.118)$$

We can write the state-input stack as:

$$\begin{aligned} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} &= \begin{bmatrix} x(k) \\ u_{Ti}(k) \\ u_i(k) \\ u_{Bi}(k) \end{bmatrix} \\ &= \begin{bmatrix} I_{n \times n} & 0_{n \times 1} \\ F_{Ti} & 0_{t \times 1} \\ 0 & 1 \\ F_{Bi} & 0_{b \times 1} \end{bmatrix} \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix} \\ &= \mathbf{G}_i \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix} \end{aligned} \quad (1.119)$$

By defining a \mathbf{P} akin to the one from Eq. 1.92

$$\mathbf{P}_i = \mathbf{G}_i^T \mathbf{P} \mathbf{G}_i \quad (1.120)$$

We can write an input-decoupled Q-function for $u_i(k)$ as

$$Q_i(x(k), u_i(k)) = \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix}^T \mathbf{P}_i \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix} \quad (1.121)$$

Once again we have a function of current state $x(k)$, but now the only concern is a single input variable $u_i(k)$. In our previous Q-function the controller that was built in was $r \times n$; now we are looking for a $1 \times n$ controller F_i . Just as we did for Policy Iteration, we now define a recurrence equation like that seen in Eq. 1.91.

$$Q_i(x(k), u_i(k)) = \gamma Q_i(x(k+1), u_i(k+1)) + U(k) \quad (1.122)$$

Where the exact same logic of optimality and cost minimization applies as it did in the Policy Iteration example. It can be shown that each optimal input-decoupled Q-function satisfies its own recurrence equation¹⁵. That is

$$Q_i(x(k), u_i(k)) = \gamma \min_{u_{i(k+1)}} Q_i(x(k+1), u_i(k+1)) + U_i(k) \quad (1.123)$$

The analogies to Policy Iteration continue where instead of Eq. 1.87 we now have

$$Q_i(x(k), u_i(k)) = \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix}^T \mathbf{P}_i \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix} \quad (1.124)$$

and instead of Eq. 1.92

$$\mathbf{P}_i = \begin{bmatrix} \mathbf{P}_{ixx} & \mathbf{P}_{ixu} \\ \mathbf{P}_{ixu}^T & \mathbf{P}_{iyy} \end{bmatrix} \quad (1.125)$$

Where the F_i associated with Q_i is captured as it is in Eq. 1.93

$$F_i = -(\mathbf{P}_{iyy})^{-1} (\mathbf{P}_{ixu}^T) \quad (1.126)$$

¹⁵M.Q. Phan (n.d.). “Optimal State-Space System Identification and Control”. Writing in Process.

A similar stacking computation as seen in Eq. 1.107 can be done, except now we use X_{i_j} , defined as

$$X_{i_j}(k) = \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix}^T \otimes \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix}^T - \gamma \begin{bmatrix} x(k+1) \\ F_{i_j}x(k+1) \end{bmatrix}^T \otimes \begin{bmatrix} x(k+1) \\ F_{i_j}x(k+1) \end{bmatrix}^T \quad (1.127)$$

Note that we still compute $U(k)$ in the complete form, using all the inputs on the system not just the current one of interest (u_i). We can solve for \mathbf{P}_i in the exact same manner as we do in Eq. 1.109, with the iterative equation

$$F_{i_{j+1}} = -(\mathbf{P}_{i_{j_{uu}}})^{-1} \left(\mathbf{P}_{i_{j_{xx}}}^T \right) \quad (1.128)$$

Example — Input Decoupling

Once again we turn to the system in Figure 1.1, Eqs. 1.44 which produced the F_{LQR}^γ shown in Eq. 1.45

$$F_{LQR}^\gamma = \begin{bmatrix} 9.9546 & -1.8952 & -2.6156 & -0.3501 \\ -25.2185 & 29.3267 & -0.8026 & -3.767 \end{bmatrix} \quad (1.45)$$

As before, our system has four states ($n = 4$) and two inputs ($r = 2$) but we only learn one input at a time now. So each $X_j(k)$ will be 1×25 ($n + 1)^2 = (4 + 1)^2$ versus the Policy Iteration's 36. We once again set the number of controllers to learn, the number of inputs per controller, and an initial controller. We still compute $u(k)$, but only learn on $u_i(k)$. So $u(k) = Fx(k)$, but then we modify input i as

$$u_i(k) = F_i x(k) + v(k) \quad (1.129)$$

Where $v(k)$ is some random value. We could also purely randomize our input as

$$u_i(k) = v(k) \quad (1.130)$$

We then apply it to the system to produce $x(k+1)$. With our $u_i(k), x(k), x(k+1)$ triplet, following Eq. 1.127 we formulate $X_{ij}(k)$. At the same time, we will compute the utility $U(k)$ as defined in Eq. 1.39. For our system, we will repeat this process 24 more times before we can update the controller once. We compute \mathbf{P}_{ij}^S from Eq. 1.108 (using $X_{ij}(k)$ in place of $X_j(k)$), and undo the stack operator by reshaping it into a 5×5 matrix. Symmetry is imposed once again.

Refer to Eq. 1.125 to extract the components to solve for our next controller as shown in Eq. 1.128. In input decoupling, we grab the bottom right scalar of \mathbf{P}_{ij} as $\mathbf{P}_{i_{uu}}$, and the bottom left 1×4 block as $\mathbf{P}_{i_{xu}}^T$. We then use those parameters to update our controller and repeat the process. In this case, after five controllers (of 25 trials each) we produce the controllers seen in Eq. 1.131

$$\begin{aligned} F_1 &= \begin{bmatrix} 9.9546 & -1.8952 & -2.6156 & -0.3501 \end{bmatrix} \\ F_2 &= \begin{bmatrix} -25.2185 & 29.3267 & -0.8026 & -3.767 \end{bmatrix} \end{aligned} \quad (1.131)$$

Which once again matches our LQR controller exactly. The learning process and application can be seen in Figures 1.21a - 1.21d

It can be seen how the control processes take longer than the policy iteration approach, and by inspecting the inputs you can see alternating noises indicating the rotations of learning on the different inputs. Additionally, Figures 1.22a and 1.22b show how the various parameters converged through the learning process. Notice how it is only every other controller number that parameters change for each input.

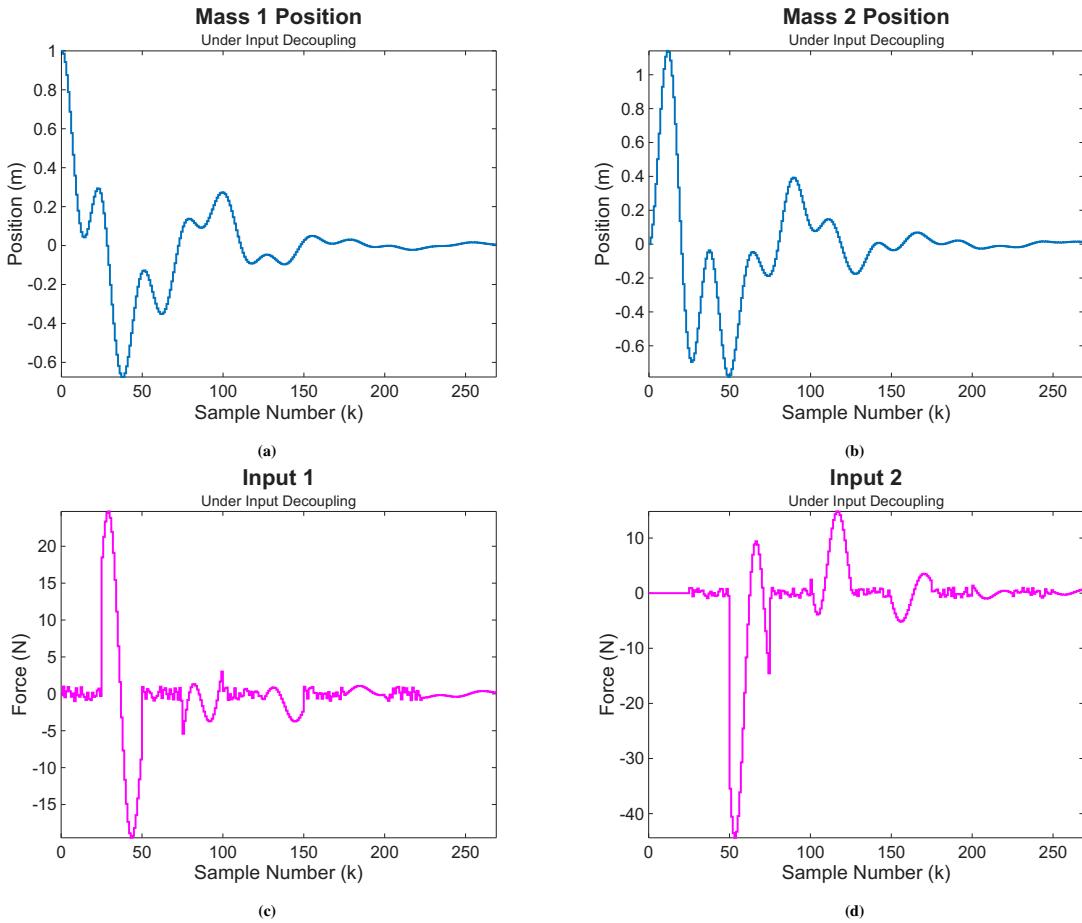


Figure 1.21: Input-Output Data of Dual-Spring-Mass system under Input Decoupled Learning. 5 passes are made on each input, of which we have two, and requires 25 trials each. Learning is halted for the final 20 trials which can be seen by both inputs being smooth at the same time.

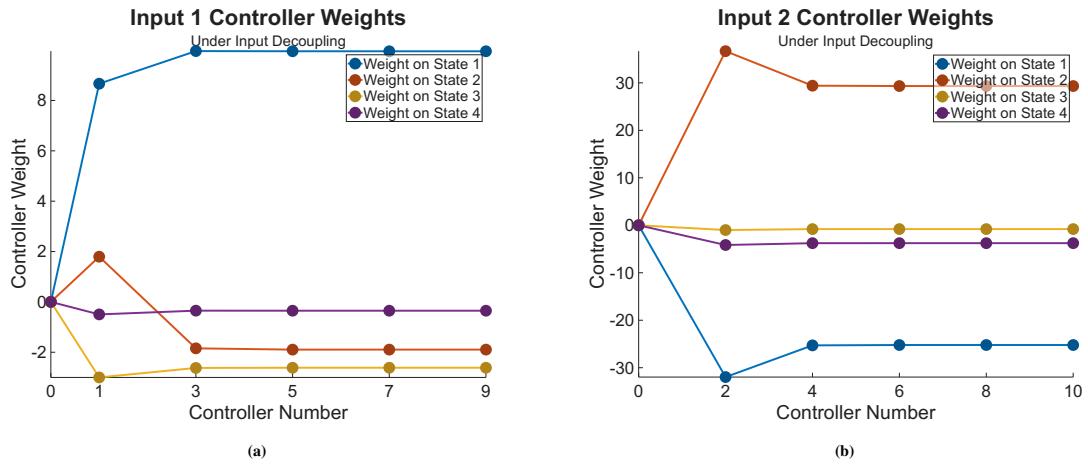


Figure 1.22: Progression of Controller Weights through Input-Decoupled trials. Notice how for the dual-input system, the weights for a given input are only updated every other trial.

1.8 Summary

This concludes our crash course of Modern Control Theory. From Continuous State-Space to Reinforcement Learning, we have covered all the background knowledge known and verified in the field. All information that follows is merely adaptations and re-representations of what you now know.

Chapter 2

Methods and Experimentation

2.1 Reinforcement Learning on Iterative Learning Control

The technique of ILC has been shown to provide low-error tracking to a specific goal under a well-defined controller. To define a controller with respect to a specific cost function, RL has also been shown to provide a very viable approach¹. How do these two methods behave when combined?

Recall our earlier definition of the ILC system:

$$e_{j+1} = Ie_j - P\delta_{j+1}\underline{u} \quad (1.60)$$

And its controller

$$\delta_{j+1}\underline{u} = \mathcal{L}e_j \quad (1.61)$$

It is logical to draw parallels from this formulation to that our ABCD formulation

$$x(k+1) = Ax(k) + Bu(k) \quad (1.27)$$

$$y(k) = Cx(k) + Du(k) \quad (1.28)$$

Where our ‘state’ is now the error e_j and our ‘input’ is the change in inputs $\delta_{j+1}\underline{u}$. We will refer to the error term as our ‘ILC State’ and the change in inputs as our ‘ILC Input’.

As previously shown, our ‘ILC State’ (n_{ILC}) is now a $pm \times 1$ vector, and our ‘ILC Input’ (r_{ILC}) is $pr \times 1$. However, a state is still a state, and an input is an input. So the principals of RL still apply to find a controller, now demarcated \mathcal{L} , that sends our error state to zero. Thus our utility function that defines the cost function that the found controller will

¹Yueqing Zhang, Bing Chu, and Zhan Shu (2019). “A Preliminary Study on the Relationship Between Iterative Learning Control and Reinforcement Learning”. In: *IFAC-PapersOnLine* 52.29. 13th IFAC Workshop on Adaptive and Learning Control Systems ALCOS 2019, pp. 314–319. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2019.12.669>. URL: <https://www.sciencedirect.com/science/article/pii/S2405896319326187>.

minimize is

$$U_j = \delta_j \underline{u}^T R \delta_j \underline{u} + e_j^T Q e_j \quad (2.1)$$

2.1.1 Example — Policy Iteration on ILC

We first begin by setting the number of steps in our process. We will work with a $p = 10$ (why not the same 100 we used earlier will be shown shortly). Given our two-input, two-output system, that means we have 20 effective states and 20 effective inputs. We use this resolution to then set our goal output. We will replicate the earlier goals of

$$y_1^* = \cos\left(\frac{2\pi k}{p}\right) \quad y_2^* = \sin\left(\frac{2\pi k}{p}\right) \quad (1.62)$$

stacked as in Eq. 1.63

Next, as we are operating without a controller, we must define our cost matrices

$$Q = 100 \cdot I_{20 \times 20} \quad R = 1 \cdot I_{20 \times 20} \quad \gamma = 0.8 \quad (2.2)$$

Using the complete knowledge afforded to us in simulation, we can find the 20×20 controller \mathcal{L}_{LQR}^γ

$$\mathcal{L}_{LQR}^\gamma = \begin{bmatrix} 0.0198 & -0.0000 & \cdots & 0.2288 & 0.1583 \\ 0.0001 & 0.0393 & \cdots & 0.1583 & 0.3778 \\ \vdots & \vdots & & \vdots & \vdots \\ -0.0000 & -0.0000 & \cdots & 0.0198 & 0.0001 \\ -0.0000 & -0.0000 & \cdots & -0.0000 & 0.0393 \end{bmatrix} \quad (2.3)$$

Given our ILC state-input dimensions, $X_j(k)$ will be $1 \times 1,600$ ($(n+r)^2 = (20+20)^2$). This does not mean 16 seconds of data (from our systems $\Delta t = 0.01$ seconds), but it is 1,600 trials of 10 steps (p), which means 160 seconds (ideally) and lots of potentially

wasted parts for a single controller update. In this example, we will learn five controllers.

We start with a controller of all zeros, and begin learning just as before. There must be one initial trial outside of the learning to generate our first ‘state’ of error. It is logical to have this be the open loop behavior, in which case the output \underline{y} is equal to the noise and initial conditions parameter \underline{d} (from Eq. 1.47). For each trial, we compute the change in inputs,

$$\delta_j \underline{u} = \mathcal{L} e_{j-1} + v_j \quad (2.4)$$

where v_j is our exploration term. In this case, v_j is normally distributed around 0 and covers ranges [-1, 1]. We then compute and apply the system input as $\underline{u}_j = \underline{u}_{j-1} + \delta_j \underline{u}$. Keep in mind the distinction between the input to the system, \underline{u}_j , and the input to the ILC learning of $\delta_j \underline{u}$.

Upon applying \underline{u}_j , our system will produce a sequence of outputs. Recall that as we cannot control $y(0)$, we exclude it from our model such that

$$\underline{y} = \begin{bmatrix} y(1) \\ \vdots \\ y(p) \end{bmatrix} \quad (2.5)$$

Computing e_j from $\underline{y}^* - \underline{y}_j$, we have all the information we need to proceed with learning and the next ILC trial.

For the purposes of learning, observe the following analogies, where we translate the

RL formatting to one in-line with the ILC system

$$x(k) \rightarrow e_{j-1} \quad (2.6)$$

$$u(k) \rightarrow \delta_j \underline{u} \quad (2.7)$$

$$x(k+1) \rightarrow e_j \quad (2.8)$$

$$Fx(k) \rightarrow \mathcal{L}e_j \quad (2.9)$$

$$(2.10)$$

So our $1 \times 1,600 X_j$ is

$$X_j = \begin{bmatrix} e_{j-1} \\ \delta_j \underline{u} \end{bmatrix}^T \otimes \begin{bmatrix} e_{j-1} \\ \delta_j \underline{u} \end{bmatrix}^T - \gamma \begin{bmatrix} e_j \\ \mathcal{L}e_j \end{bmatrix}^T \otimes \begin{bmatrix} e_j \\ \mathcal{L}e_j \end{bmatrix}^T \quad (2.11)$$

We repeat the ILC process until we have enough trials to solve for our \mathbf{P}_j^S , unstack it, impose symmetry, and update our controller as

$$\mathcal{L} = -(\mathbf{P}_{uu})^{-1} (\mathbf{P}_{xu}^T) \quad (2.12)$$

Where \mathbf{P}_{uu} is now the bottom-right 20×20 matrix in P_j , and \mathbf{P}_{xu}^T the bottom-left 20×20 .

For the listed example, we repeat this process 4 more times (for a total of 5 controllers).

Our end controller is

$$\mathcal{L}_{policy} = \begin{bmatrix} 0.0198 & -0.0000 & \cdots & 0.2288 & 0.1583 \\ 0.0001 & 0.0393 & \cdots & 0.1583 & 0.3778 \\ \vdots & \vdots & & \vdots & \vdots \\ -0.0000 & -0.0000 & \cdots & 0.0198 & 0.0001 \\ -0.0000 & -0.0000 & \cdots & -0.0000 & 0.0393 \end{bmatrix} \quad (2.13)$$

With an error magnitude of 8.22×10^{-11} , computed as

$$\frac{|\mathcal{L}_{policy} - \mathcal{L}_{LQR}^\gamma|}{p^2 mr} \quad (2.14)$$

Where we use the ‘norm’ operator in the numerator to compute the absolute magnitude of the differences, then scale down by the number of elements in each controller.

So via RL we can still extract the exact LQR controller. The controller learning process can be seen in Figures 2.1a- 2.1d.

Each input number refers to the step along the process in which it is being applied, and the state refers to the errors of the previous trial.

Note that we only are showing a few select weights and inputs, as each of the twenty inputs are informed by twenty errors, which would make for a very cluttered set of plots.

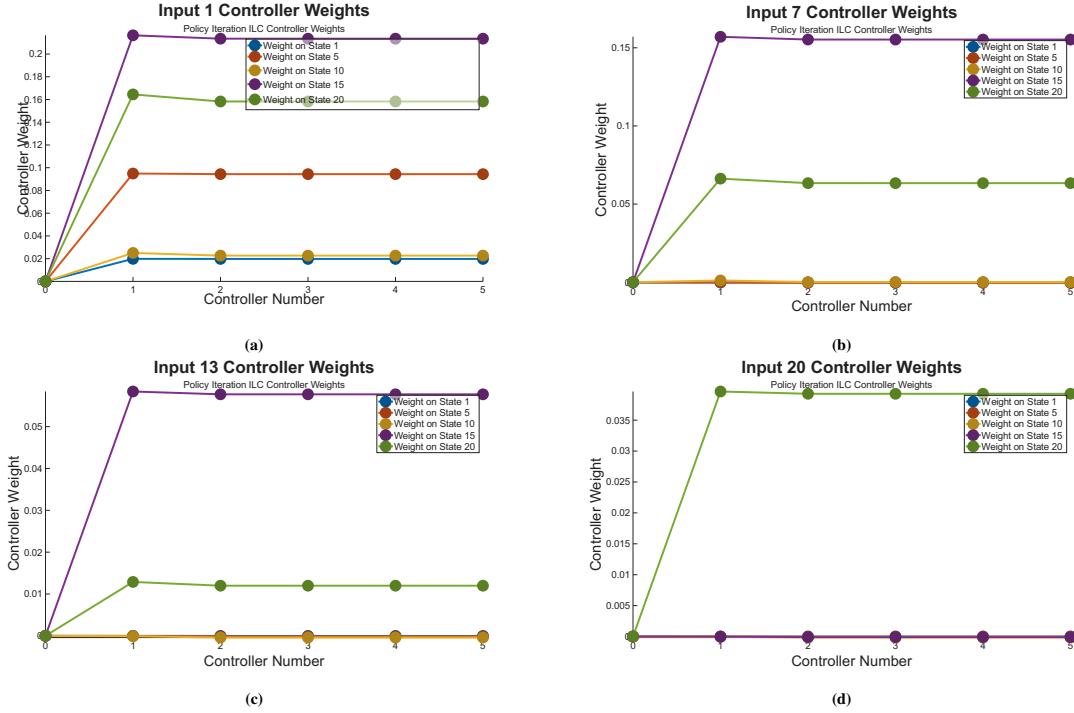


Figure 2.1: Select Controller Weights on Select Inputs for an ILC problem through Policy Iteration Trials. 5 Controllers are learned, for an ILC system on the Dual-Spring-Mass system of trial length $p = 10$ – each controller update requires 1600 ILC trials

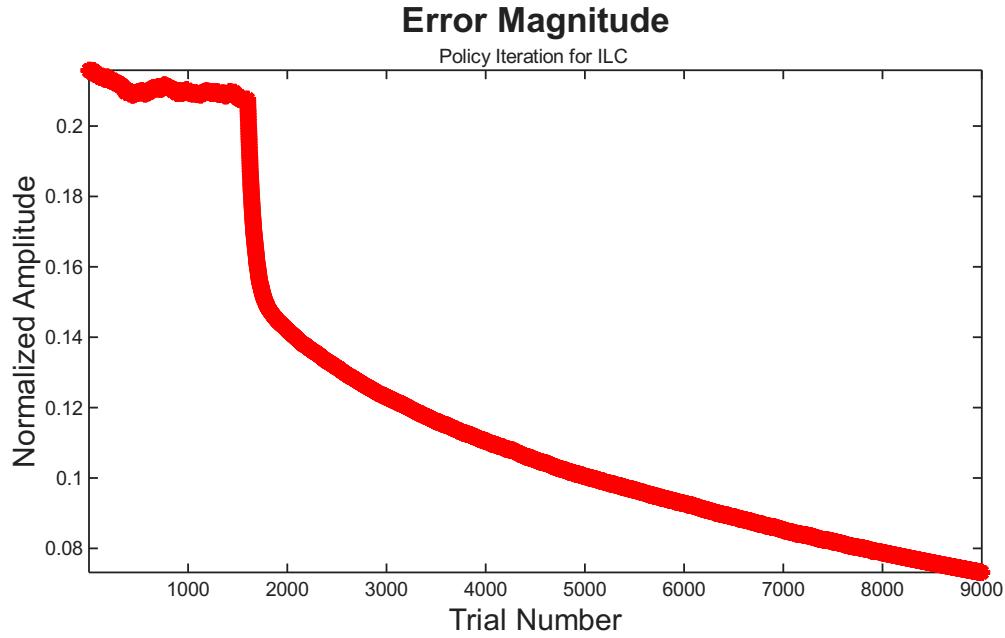


Figure 2.2: Error Magnitude of Output through Policy Iteration Trials, where $Q/R = 100$. Observe that after the first controller is learned and applied, there is a sharp reduction in error but due to the relatively high R in its definition, subsequent reductions in error are much slower.

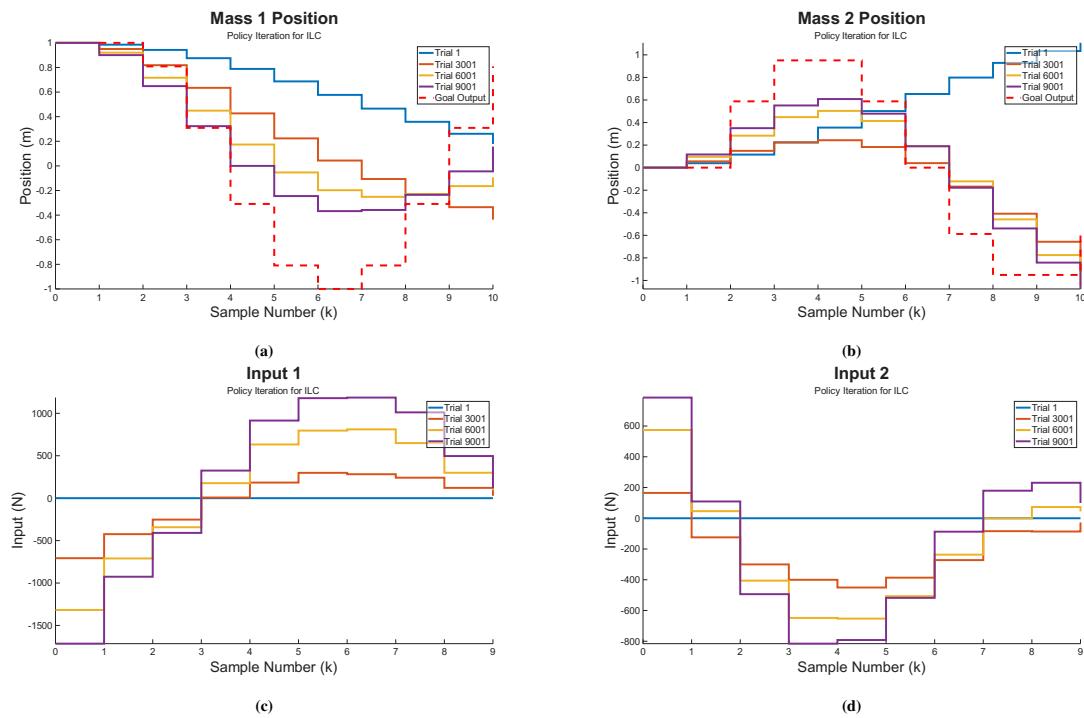


Figure 2.3: Input-Output Progressions through Policy Iteration Trials of $Q/R = 100$. Observe Trial 1 to be the open-loop response, and subsequent trials to be progressing towards the goal output.

Shaped Outputs

Policy Iteration for ILC

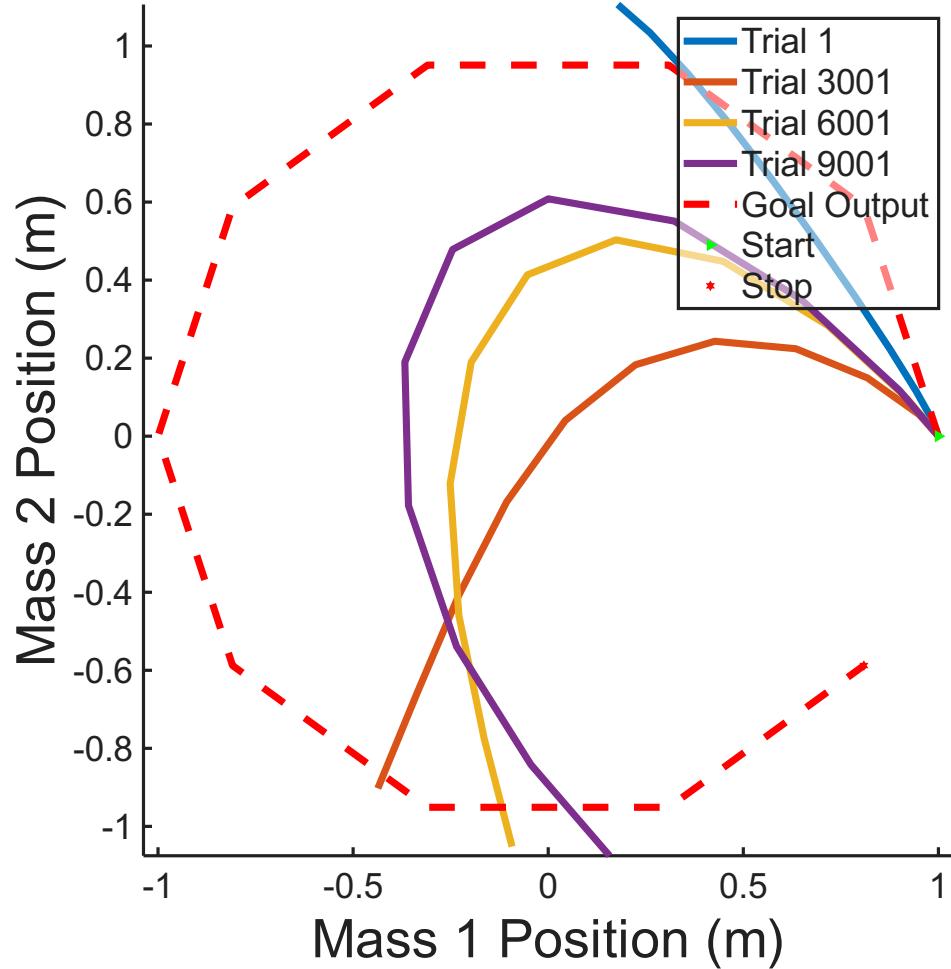


Figure 2.4: Shaped Output through Policy Iteration Trials of $Q/R = 100$. Observe the progression towards the desired output, but also the extreme number of trials that would be needed to properly further the learning.

It is clear here that even over our nearly 10,000 trials, we did not perfectly capture our output. This is since our R weighting was relatively large. Further reductions of R does not remove the existence of an LQR controller but introduces issues when computing the pseudo-inverse of X_j . It is a requirement for X_j to be well-conditioned and composed of sufficient linearly independent equations (full rank). When learning in ILC with a small R , we approach a point where both e_j and $\delta_{j+1}u$ are near-zero and we lose the ability to even generate linearly-independent X_j s, if we are learning with sequential trials. This is because we tell the system it is ok to make big changes between trials, and in doing so it too-quickly sends our error e_j to zero, meaning it no longer needs to generate a change in inputs $\delta_j\underline{u}$ – our X_j is then essentially all zeros.

If one wishes to reduce R , there are multiple approaches. The first is to collect more trials per input controller. This will obviously increase the total number of trials needed to learn, but by increasing our collections, we improve our odds of having enough linearly-independent samples. The next approach would be to completely randomize both ‘state’ and ‘input’, to ensure truly arbitrary e_{j-1} , $\delta_j\underline{u}$, e_j triplets – this can also be accomplished with purely randomized inputs. The final option, and the one shown below, is to increase the magnitude of your exploration term. Whereas previous v_j was normally distributed around 0, ranging from [-1, 1], we will now explore in the range of [-1000, 1000]. Keeping the same goal as defined in Eq. 1.62, but redefining our cost-matrices as

$$Q = 100 \cdot I_{20 \times 20} \quad R = 10^{-6} \cdot I_{20 \times 20} \quad \gamma = 0.8 \quad (2.15)$$

We have the new LQR Controller

$$\mathcal{L}_{LQR}^\gamma = \begin{bmatrix} 2,095.3 & -44.6 & \cdots & 1.8 & -0.5 \\ -22.4 & 2,050.8 & \cdots & -0.5 & 1.4 \\ \vdots & \vdots & & \vdots & \vdots \\ -0.9 & 0.1 & \cdots & 2,095.3 & -22.4 \\ 0.1 & -6.7 & \cdots & -44.6 & 2,050.8 \end{bmatrix} \quad (2.16)$$

Repeating our Policy Iteration learning process with our new R and v_j amplitude, we see we can once again extract the LQR controller

$$\mathcal{L}_{LQR}^\gamma = \begin{bmatrix} 2,095.3 & -44.6 & \cdots & 1.8 & -0.5 \\ -22.4 & 2,050.8 & \cdots & -0.5 & 1.4 \\ \vdots & \vdots & & \vdots & \vdots \\ -0.9 & 0.1 & \cdots & 2,095.3 & -22.4 \\ 0.1 & -6.7 & \cdots & -44.6 & 2,050.8 \end{bmatrix} \quad (2.17)$$

With an error magnitude of 8.55×10^{-6} , computed as in Eq. 2.14. We see a much sharper error progression in Figure 2.6 that is to be expected when imposing such a small cost on the change of inputs. We see this occur right after the first learned controller is applied. The progression of controller weights, system outputs, and inputs can also be shown to have sharp drop upon application of the first learned controller. Due to the reduced R , we only run 100 trials without the exploration term, and it only takes less than a handful of those for the error to drop to zero.

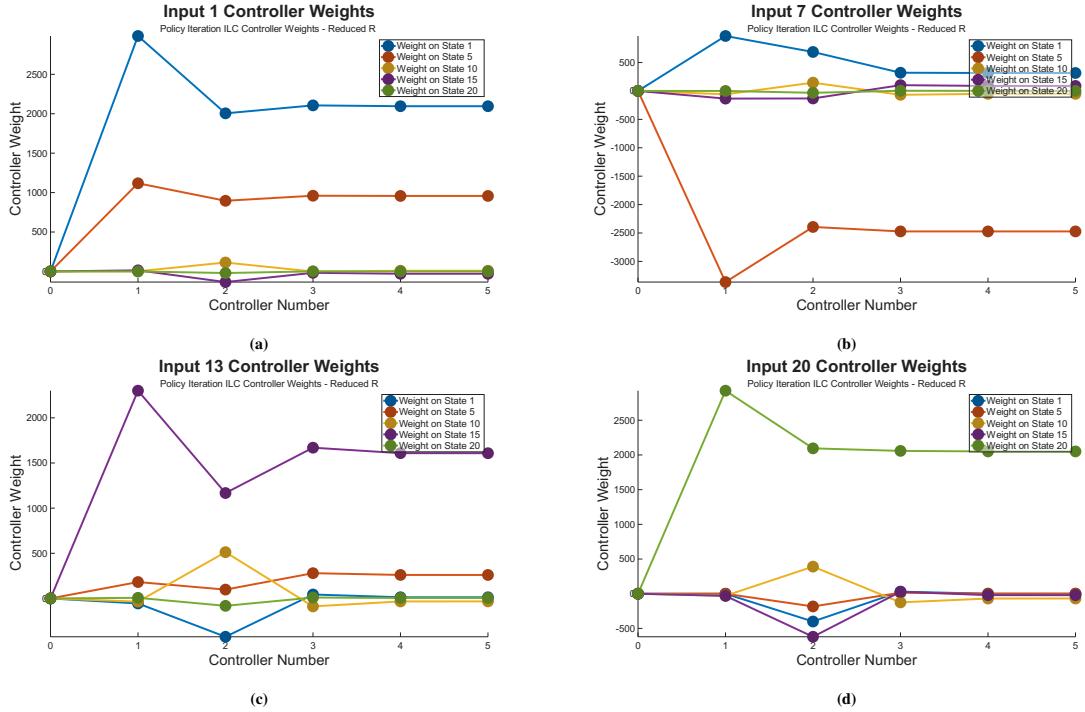


Figure 2.5: Select Controller Weights on Select Inputs progression through Policy Iteration Trials when $Q/R = 1 \times 10^8$. Due to the necessary amplified exploration $v(k)$, observe the tendency for weights to converge much less smoothly than before, displaying behaviors of overshoot and lag.

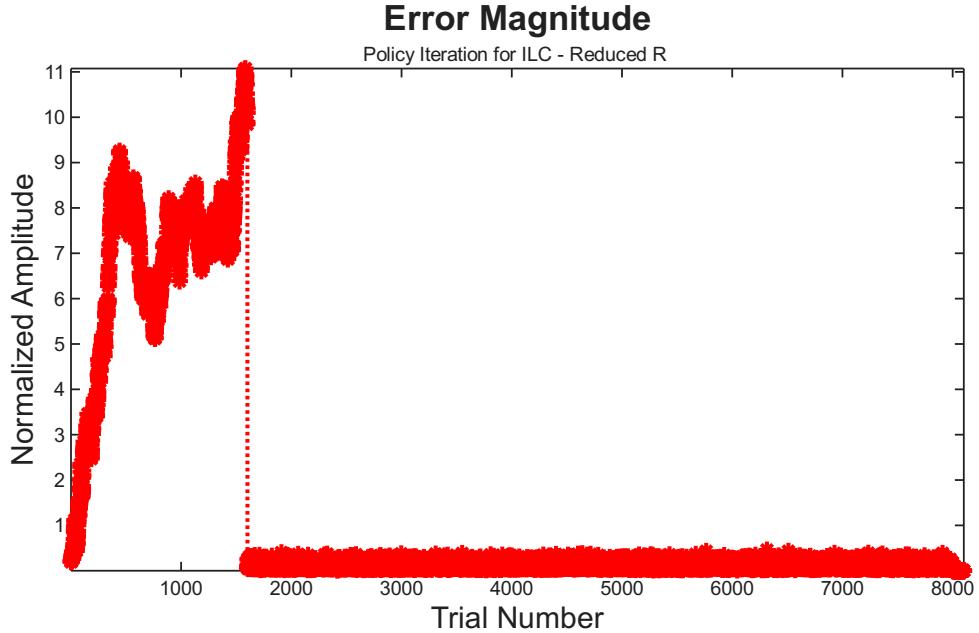


Figure 2.6: Error Magnitude of Output through Policy Iteration Trials with $Q/R = 1 \times 10^8$. Observe how initial errors creep up much further than the earlier shown $Q/R = 100$ trial, but the application of the first trial drastically reduces error , such that it would be zero were it not for the extreme input exploration terms.

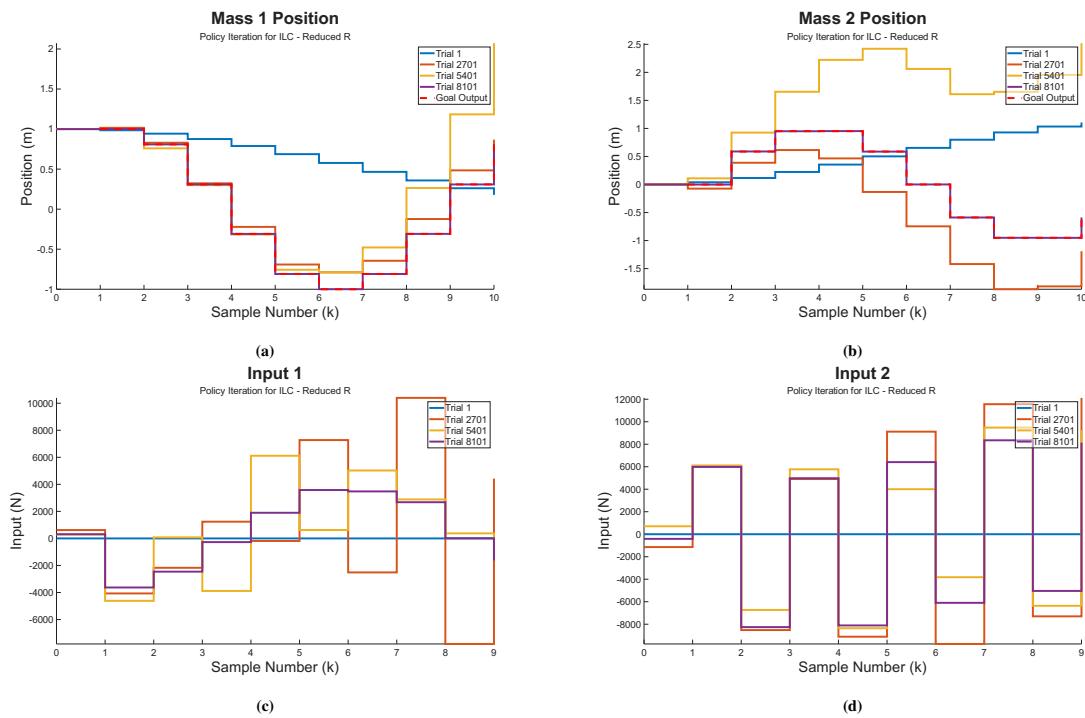


Figure 2.7: Progression of Input-Output Data in a Dual-Spring-Mass system under ILC derived from RL when $Q/R = 1 \times 10^8$

Shaped Outputs

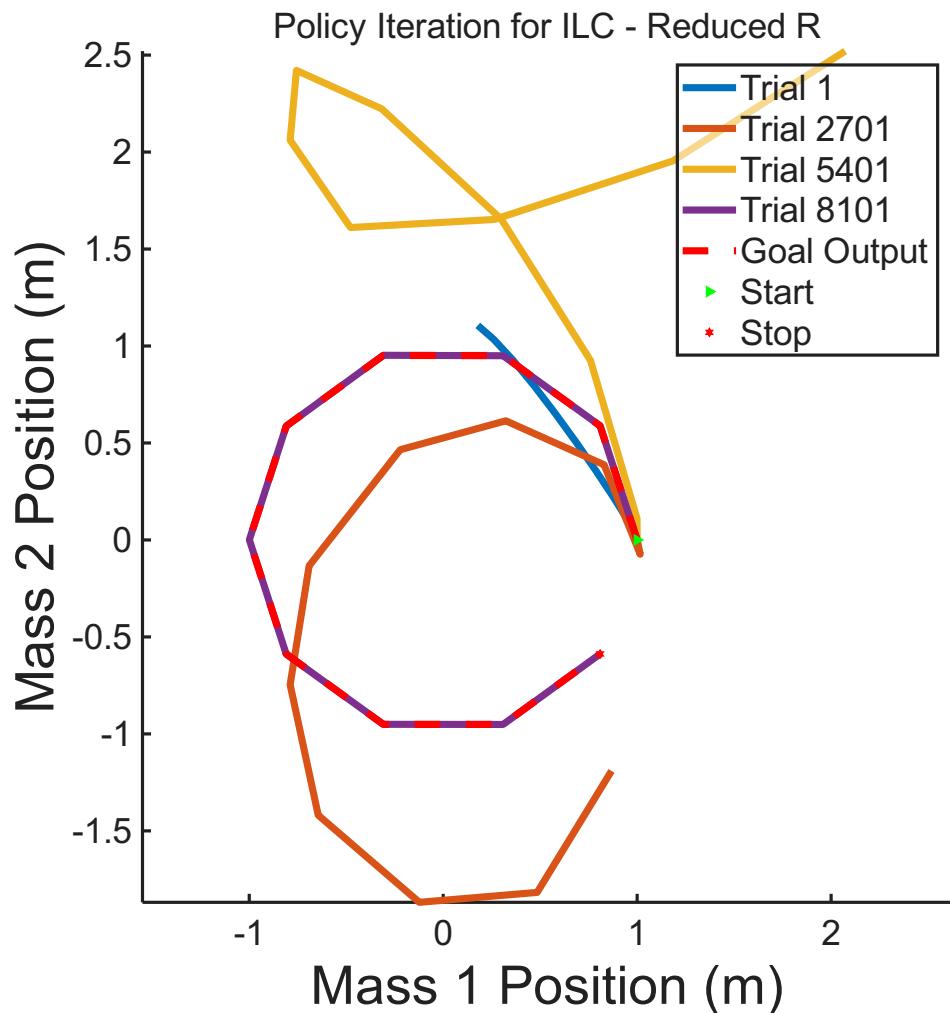


Figure 2.8: Shaped Output through Policy Iteration Trials when $Q/R = 1 \times 10^8$

The trade-offs of this approach are obvious. With such a large exploration term, while we are learning we produce no outputs like our goal, as shown in 2.8. Additionally our controller progression is a lot jumpier - displaying characteristics of overshoot. Yet once we are done learning, we can apply this controller and within a very limited number of trials have zero error.

As mentioned at the beginning of the example, this is a relatively low-resolution ILC problem, with $p = 10$. What if we were to increase this to $p = 100$ as with our basic ILC introduction problem? Our number of ILC states would climb from 20 to 200, as would our ILC inputs. Which would mean the dimensions of our X_j would go from $1 \times 1,600$ to $1 \times 160,000$. \mathbf{P}_j just went from having 2.56×10^6 elements - already a lot - to 2.56×10^{10} . Monetarily and computationally this makes the RL approach very costly. In fact, if we set $p = 100$, MATLAB will not even begin to try and solve the problem. When we go to pre-allocate the array that holds all the stacks of X_j s, we are met with the following error:

```
Error using zeros
Requested 160000x160000 (190.7GB) array exceeds maximum
array size preference (31.6GB). This might cause MATLAB
to become unresponsive.
```

so if we want to learn even the most common of ILC problems, we have to somehow reduce our dimensions.

2.1.2 Example — Input Decoupling on ILC

We have already shown that Input Decoupling can be used to reduce the dimensions of X_j from $1 \times (n + r)$ to $1 \times (n + 1)$, so it is a logical place to turn to when considering the costs of learning. In our ILC context, it will now take only 441 trials to update a single controller (though this must be done 20 times for a total of 8,820 trials for a complete attempt at \mathcal{L}_{LQR}^γ).

However while it will take iterating through all r inputs to arrive at the LQR controller, individual controllers that we learn will help control the system regardless of whether it is yet our complete LQR. Repeating the parameters and goal from earlier example of Policy Iteration (Eqs. 1.62 and 2.2), we predictably find the same 20×20 controller \mathcal{L}_{LQR}^γ (see Eq. 2.3).

The learning process follows just as is the policy iteration example. The only difference being that we are only learning only one controller step at a time, and following input decoupling logic. So when we compute $\delta_j \underline{u} = \mathcal{L}_i e_{j-1}$, we modify only one of the terms with our exploration, such that

$$\delta_j \underline{u}_i = \mathcal{L}_i e_{j-1} + v_i \quad (2.18)$$

Now our ILC to Input Decoupling analogies are

$$x(k) \rightarrow e_{j-1} \quad (2.19)$$

$$u_i(k) \rightarrow \delta_j \underline{u}_i \quad (2.20)$$

$$x(k+1) \rightarrow e_j \quad (2.21)$$

$$F_i x(k) \rightarrow \mathcal{L}_i e_j \quad (2.22)$$

So our $1 \times 441 X_j$ is

$$X_j = \begin{bmatrix} e_{j-1} \\ \delta_j \underline{u}_i \end{bmatrix}^T \otimes \begin{bmatrix} e_{j-1} \\ \delta_j \underline{u}_i \end{bmatrix}^T - \gamma \begin{bmatrix} e_j \\ \mathcal{L}_i e_j \end{bmatrix}^T \otimes \begin{bmatrix} e_j \\ \mathcal{L}_i e_j \end{bmatrix}^T \quad (2.23)$$

We repeat the ILC process until we have enough trials to solve for our \mathbf{P}_j^S , unstack it, impose symmetry, and update our controller as

$$\mathcal{L}_i = -(\mathbf{P}_{uu})^{-1} (\mathbf{P}_{xu}^T) \quad (2.24)$$

Where \mathbf{P}_{uu} is now the bottom-right scalar in P_j , and \mathbf{P}_{xu}^T the bottom-left 20×20 . For the listed example, we repeat this process 19 more times to cover all the ‘inputs’, then that process 4 more times to generate 5 ‘complete’ controllers. Our end controller is

$$\mathcal{L}_{decoupled} = \begin{bmatrix} 0.0199 & -0.0001 & \cdots & 0.2312 & 0.1620 \\ 0.0001 & 0.0392 & \cdots & 0.1613 & 0.3887 \\ \vdots & \vdots & & \vdots & \vdots \\ -0.0003 & -0.0000 & \cdots & 0.0198 & -0.0001 \\ -0.0003 & -0.0003 & \cdots & -0.0007 & 0.0396 \end{bmatrix} \quad (2.25)$$

Which has an error of 1.15×10^{-4} , as computed in Eq. 2.14. We show the learning process below, once again only showing a few select weights and inputs of the controller.

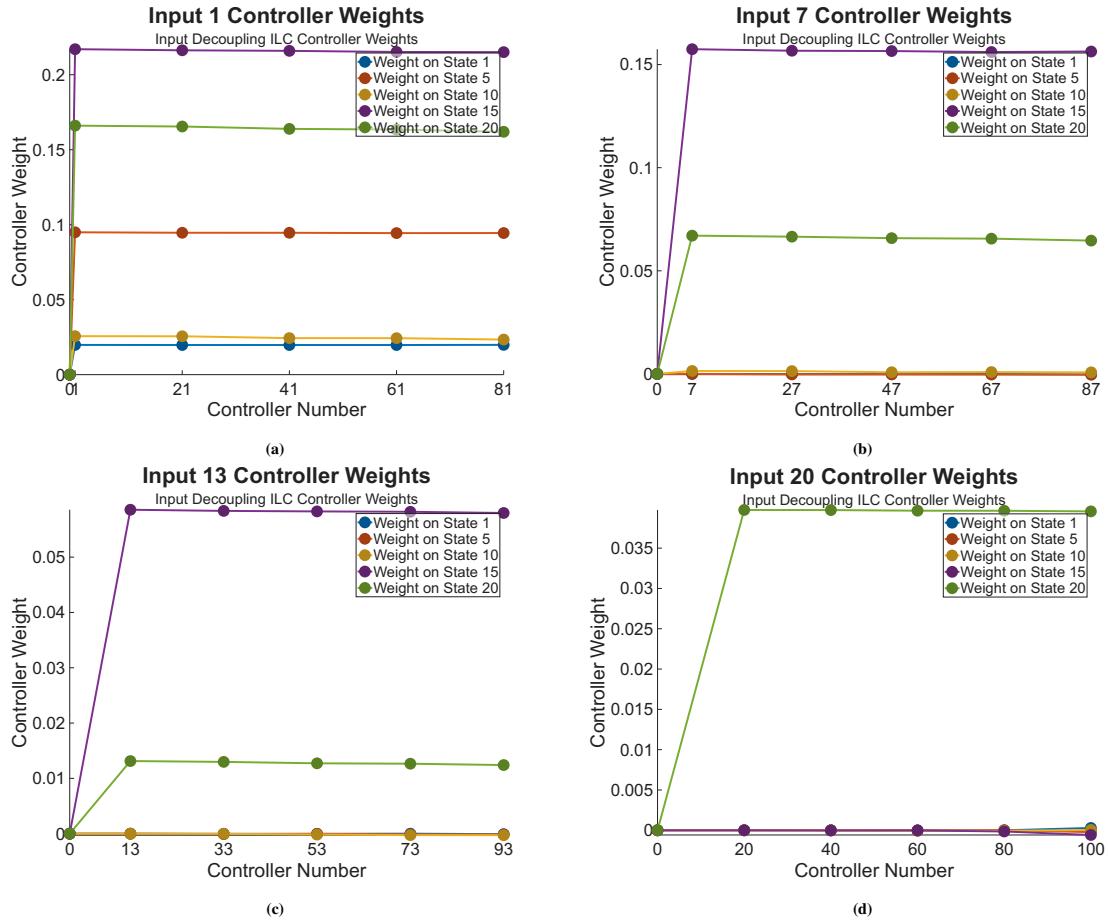


Figure 2.9: Select Controller Weights on Select Inputs through Input Decoupling Trials to learn the ILC Controller when $Q/R = 100$. Notice how each controller update takes 441 ILC trials, and each input controller is only updated at that rate.

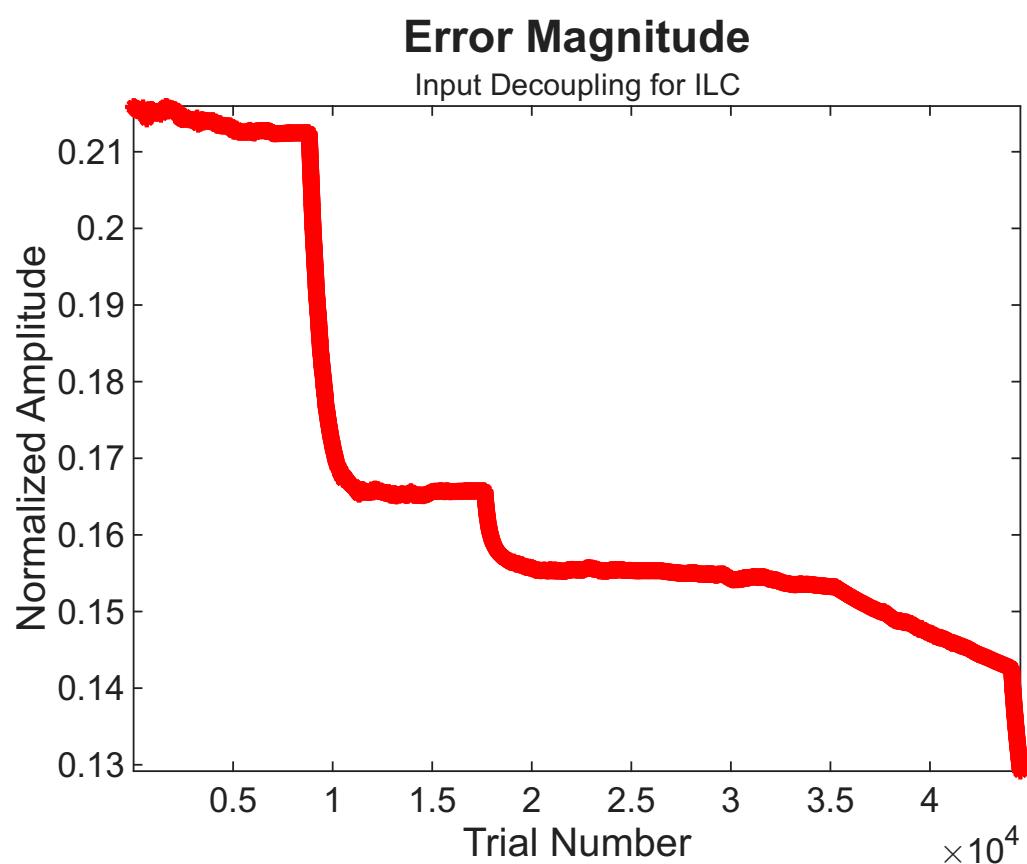


Figure 2.10: Error Magnitude of Output through Input Decoupling Trials

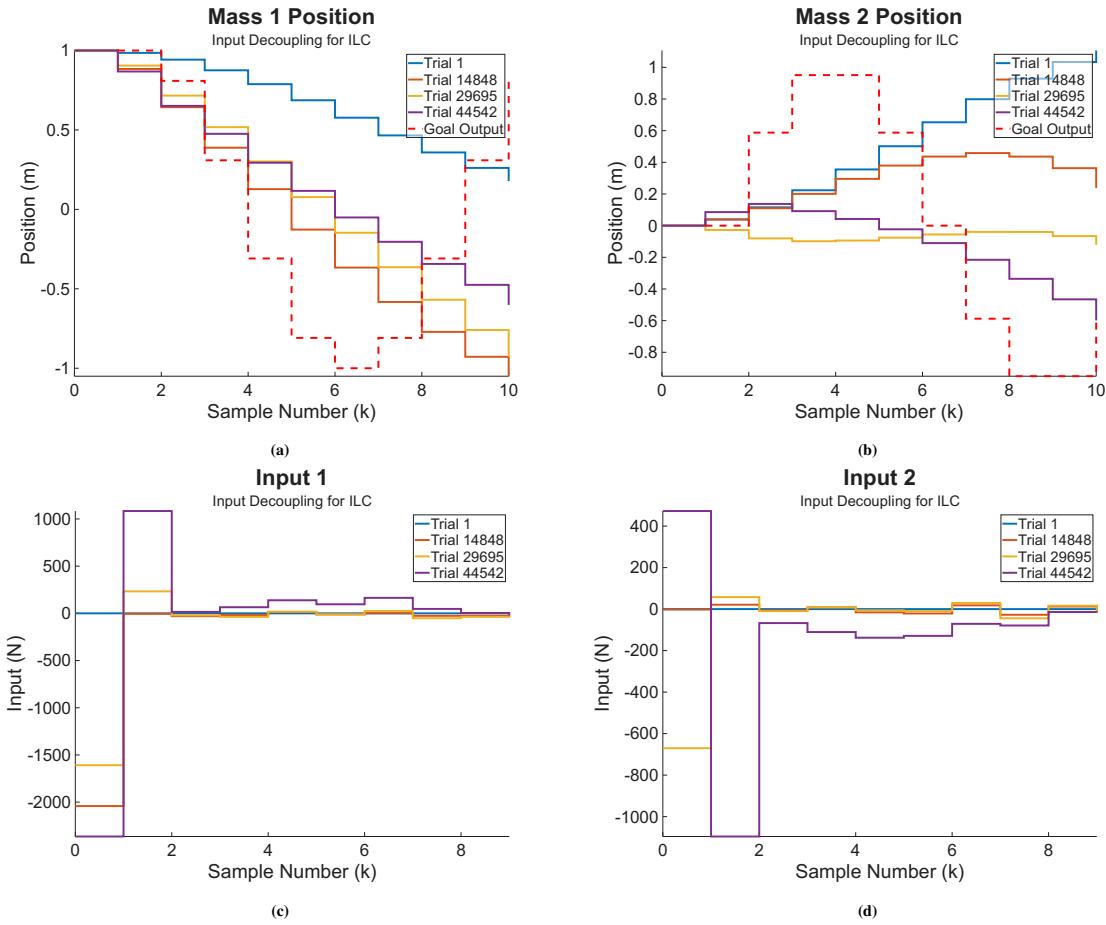


Figure 2.11: Input-Output Data progression through Input Decoupled Learning trials when $Q/R = 100$

Shaped Outputs

Input Decoupling for ILC

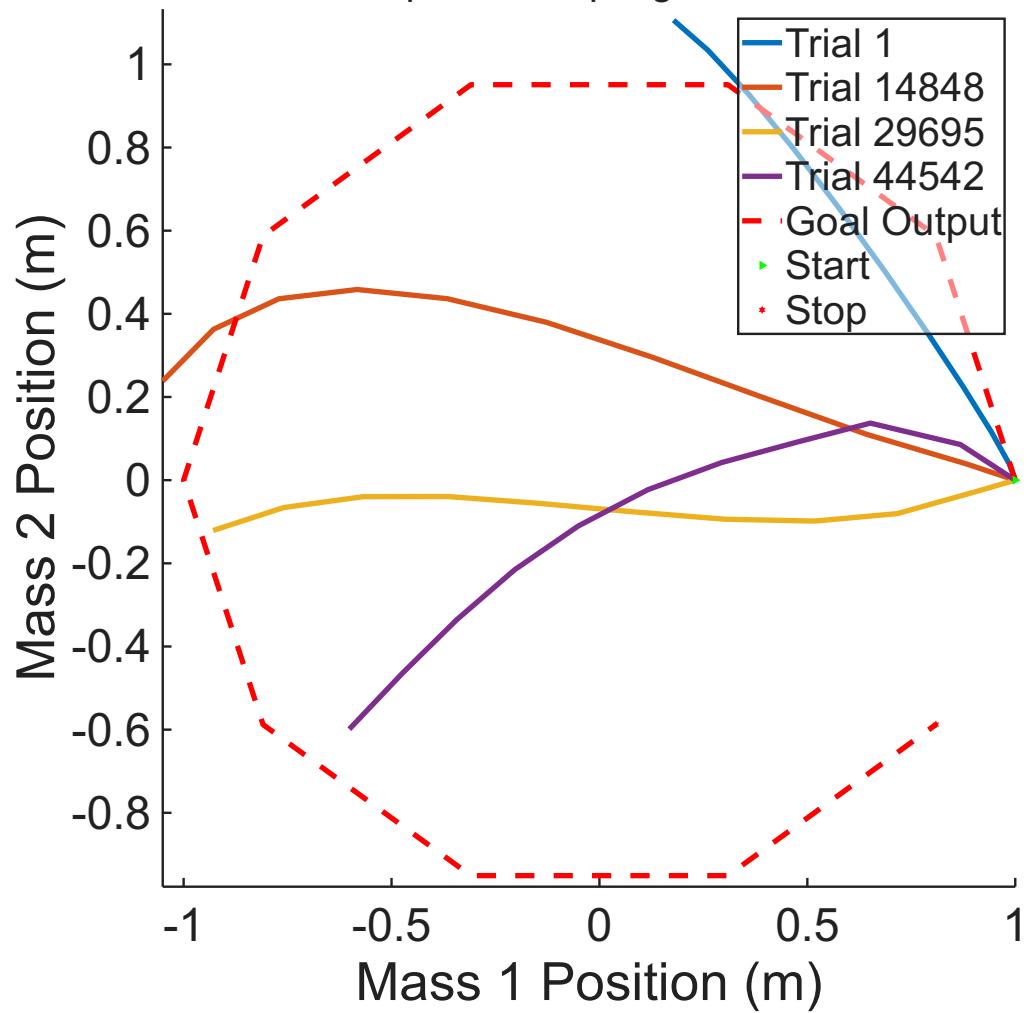


Figure 2.12: Shaped Output through Input Decoupling Trials when $Q/R = 100$.

The input decoupling approach ends up suffering from the same issues as the policy iteration approach, and then some. We could solve the same R issue the same way (more exploration), but we face the issue of rotating controllers.

While each controller update takes less collections than the Policy approach, it takes 20 controllers to arrive upon a complete controller like the one Policy produces. To generate 5 ‘complete’ controllers (comprised of fresh controllers for all 20 inputs) it takes 44,100 trials via Input Decoupling versus Policy Iteration’s 8,000.

2.1.3 Summary of RL on ILC

We have shown that the RL can be used to address the ILC problem. However the dimensions pose a significant limitation on complexity and resolution of any given goal. Additionally, the nature of the ILCs problem to want to send the ‘input’ and ‘state’ to zero make it such that we must add extra exploration terms when constructing such large matrices, otherwise we end up with ill-conditioned matrices from which a controller cannot be extracted. It is thus desirable to find a way to reduce our input-outputs signals into a lower dimensions for the learning process.

2.2 Basis Functions

While we have shown that the RL process still works for the ILC problem, the dimensional limits present a significant problem. Signal dimension reduction is not a new challenge, and can be seen from Fourier Transforms to Linear Regressions. The method which we will explore are Basis Functions. Phan and Frueh 1996 - Learning Control for Trajectory Tracking using Basis Functions - provides the logic from which we launch our approach.

2.2.1 What are Basis Functions

Basis Functions offer the ability to represent a signal as a weighting of composite signals.

Suppose you had the vector

$$\begin{bmatrix} -2 & 8 & 13 & -4 & 9 \end{bmatrix}^T \quad (2.26)$$

You could express this as a sequence of five numbers, or if you had already pre-defined some vector ϕ as

$$\phi = \begin{bmatrix} 1 & -4 & -6.5 & 2 & -4.5 \end{bmatrix}^T \quad (2.27)$$

you could capture Eq. 2.26 exactly as -2ϕ .

This is the premise behind basis functions. For any signal of length (or ‘resolution’) ℓ , we can describe it as a composite of η functions that are defined for ℓ points. We then create a $\ell \times \eta$ ‘basis space’ Φ out of basis functions ϕ_i , as shown in Eq. 2.28

$$\Phi = \begin{bmatrix} | & | & & | \\ \phi_1 & \phi_2 & \cdots & \phi_\eta \\ | & | & & | \end{bmatrix} \quad (2.28)$$

where each ϕ_i is a $(\ell \times 1)$ vector. The only condition on each basis functions is that it is

independent of / orthogonal to any other basis functions.

$$\phi_i \cdot \phi_j = 0, \text{ for } i \neq j \quad (2.29)$$

where \cdot is the dot operator. Put another way, Φ must be full rank.

To represent a signal in terms of basis functions, we then use a $\eta \times 1$ weighting vector β .

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_\eta \end{bmatrix} \quad (2.30)$$

In the ideal scenario, a signal can be captured with a single β . This happens when the signal we are representing is a scalar of the chosen basis function, as shown in Eq. 2.26. The worst case scenario we have also already seen – you just may not have realized it. Whenever a basis space is not specified, and a signal still represented, a $\ell \times \ell$ identity matrix is implicitly being used, and thus ℓ basis functions. If we were to re-express Eq. 2.26 in the worst case, then it would at most take five basis weights

$$\begin{bmatrix} -2 \\ 8 \\ 13 \\ -4 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -2 \\ 8 \\ 13 \\ -4 \\ 9 \end{bmatrix} \quad (2.31)$$

The Basis Space also does not necessarily need to be $I_{\ell \times \ell}$. So long as it is full rank, a basis space of ℓ basis functions can perfectly capture any signal. An added feature of the basis space as we have defined it, is its width will never exceed its height. That is, it will always be a tall matrix (square at most), as $\eta \leq \ell$. This fact, coupled with the full rank

nature, ensures that the a left-side product between the space's pseudo-inverse and itself will always be the identity matrix.

$$I_{\eta \times \eta} = \Phi^+ \Phi \quad (2.32)$$

To summarize, any $\ell \times 1$ signal ω can be expressed exactly as a product of a basis space Φ and basis coefficients β , so long as ω exists in the basis space.

$$\omega = \Phi \beta \quad (2.33)$$

The trick then comes down to picking the right basis functions, and ensuring ω is in the basis space. With luck, it would be possible to get a signal with a single function and weight, but luck is never a good plan.

2.2.2 Chebyshev Polynomials

Chebyshev Polynomials offer a logical foundation and algorithm for constructing basis functions². Chebyshev Polynomials of the first kind, T_n are defined as

$$T_n(\cos \theta) = \cos(n\theta) \quad (2.34)$$

and have a useful property that they are all orthogonal with respect to one another, defined over the space of [-1, 1]. They can also be generated iteratively in recurrence equation

²Nhan Nguyen (2013). “Least-Squares Model-Reference Adaptive Control with Chebyshev Orthogonal Polynomial Approximation”. In: *Journal of Aerospace Information Systems* 10.6, pp. 268–286. doi: 10.2514/1.I010037. eprint: <https://doi.org/10.2514/1.I010037>. URL: <https://doi.org/10.2514/1.I010037>.

shown in Eq. 2.37

$$T_0(x) = 1 \quad (2.35)$$

$$T_1(x) = x \quad (2.36)$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad (2.37)$$

Earlier functions, due to their lower frequency, help capture macro behaviors in a signal. As one progresses further in the process, the higher frequency signals capture the finer details. To those familiar with Fourier Analysis, you will recognize this behavior of marginal enhancements with each additional, higher frequency. Or it is not unlike in a Taylor Expansion, where we similarly see diminishing returns on accuracy from each additional component added.

Example — Matlab Creation of Chebyshev Polynomials

To utilize the features of Chebyshev Polynomials, we must first create them. Since they are only orthogonal over the domain of $x \in [-1, 1]$, that is where we will restrict ourselves for their generation. Recalling we wish to capture a signal of ℓ points, we will then define our x from $-1 \rightarrow 1$ in steps of $\frac{2}{\ell}$. This can be accomplished in Matlab by setting our x to $x = \text{linspace}(-1, 1, \ell)$ ³ – this is our T_1 (Eq. 2.36). We must similarly create a $\ell \times 1$ of all 1s to serve as our T_0 (Eq. 2.35). To generate η functions, we then iterate as described in Eq. 2.37. The process can also be seen in Code Appendix D.7

In the presented example, we set $\ell = 100$ and $\eta = 20$, generating a Φ that is 100×20 . Figure 2.13 shows our T_0 , T_1 , and some select iteratively generated functions. You will see that the x-axis units are ‘Chebyshev Steps’ that go from $0 \rightarrow 99$, even though we numerically said they had to go from $-1 \rightarrow 1$. Numerically the functions must be defined in this range, but for application the can be viewed purely as data points along a signal of

³Note the ' to transpose the vector into the column format

any length, and the real values of -1 and 1 no longer hold any meaning.

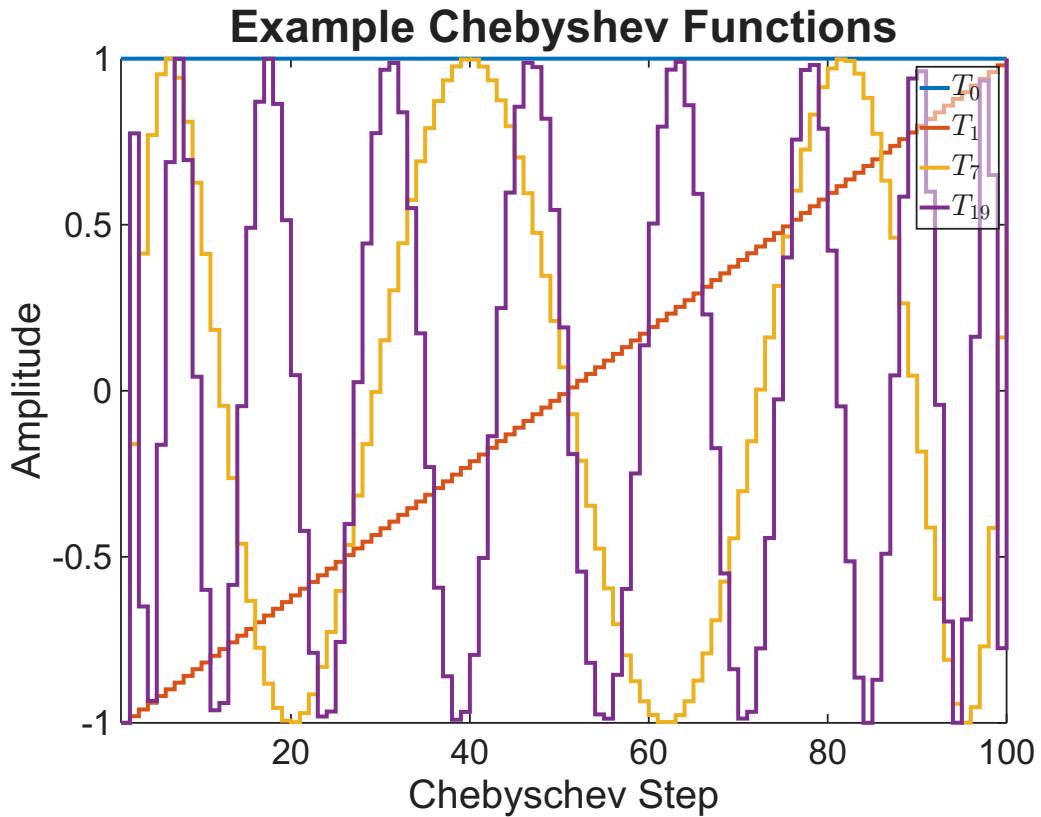


Figure 2.13: Select Chebyshev Polynomials

We can now utilize these functions to generate a wide array of complex signals. To demonstrate the arbitrary capabilities, refer to Figure 2.14. This is a 100 time step signal created with just 20 chebyshev polynomials, weighted as shown in Figure 2.15

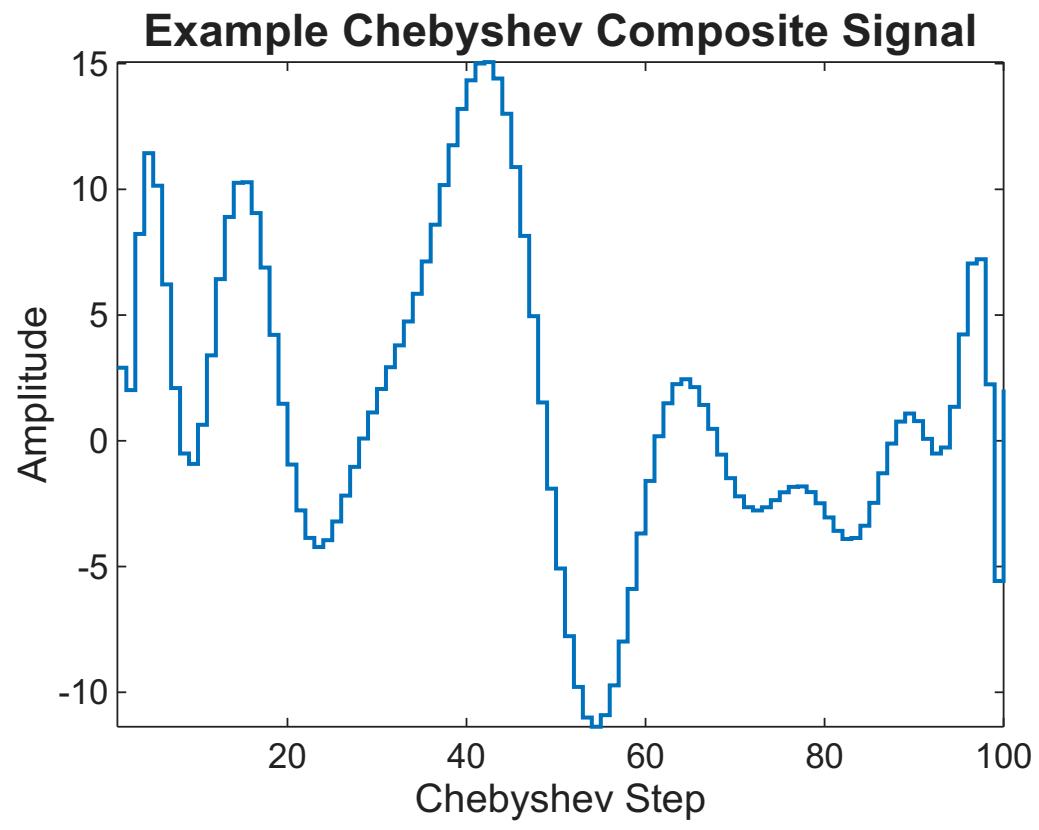


Figure 2.14: Example Signal constructed from $\eta = 20$ Chebyshev Polynomials of length $\ell = 100$

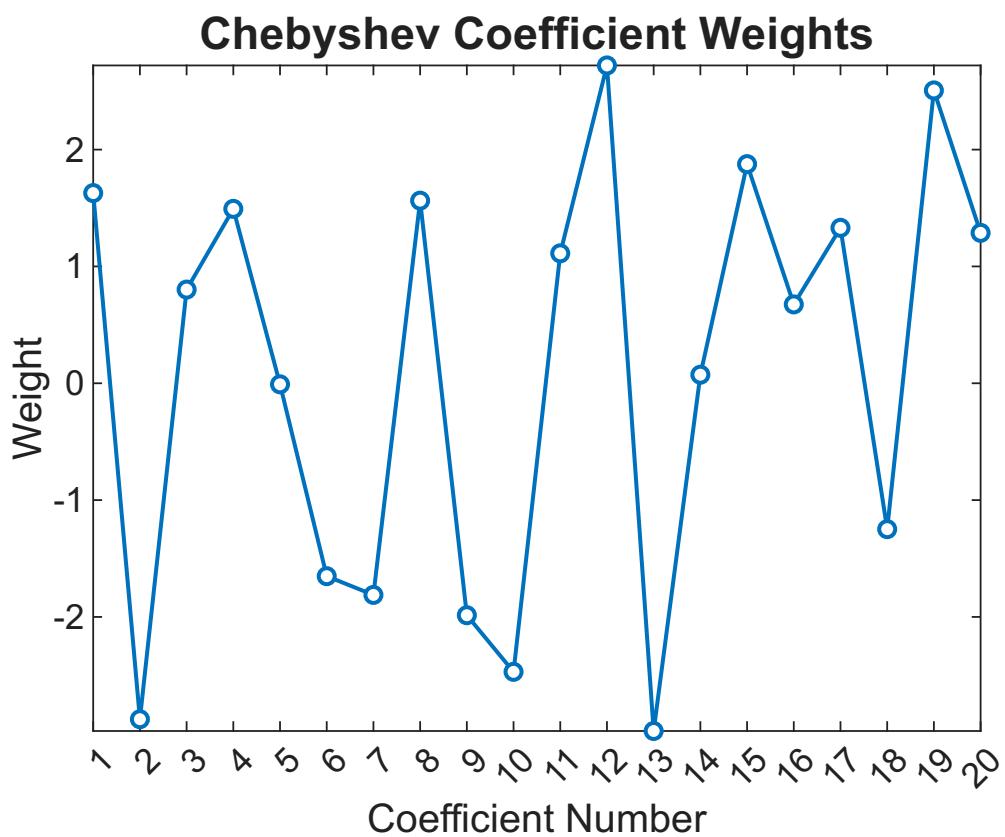


Figure 2.15: Example Chebyshev Weights to Generate Signal in Fig. 2.14

2.2.3 Basis Requirements in Reinforcement Learning

Clearly Chebyshev Polynomials are a powerful tool, but how precise must we be to apply them to our RL problem? Our goal is to reduce the number of dimensions for both the states and the inputs, so let us start by redefining the ILC problem in a basis space. Recall our model:

$$\underline{y} = P\underline{u} + \underline{d} \quad (1.47)$$

For complete flexibility, we will assign separate basis spaces to the output (\underline{y}) and our inputs (\underline{u}). The \underline{y} basis space will be Φ_y and the \underline{u} will be Φ_u . The \underline{u} will get to keep the basis coefficients convention of β shown in Eq. 2.33, but the \underline{y} space will be expressed in terms of α weights. We will have η_y basis coefficients on the output, and η_u basis coefficients on the input. So Φ_y will be $n_{ILC} \times \eta_y$ and Φ_u will be $r_{ILC} \times \eta_u$.

Assuming our basis spaces to be well defined, we then have the exact relationships

$$\underline{y} = \Phi_y \alpha \quad (2.38)$$

$$\underline{u} = \Phi_u \beta \quad (2.39)$$

which, due to the tall nature of the basis spaces, can be reversed as

$$\alpha = \Phi_y^+ \underline{y} \quad (2.40)$$

$$\beta = \Phi_u^+ \underline{u} \quad (2.41)$$

It is important to note that when our basis spaces are not well-defined, these reverse operations become approximations. $\Phi_y^+ \underline{y}$ gives us the projection of \underline{y} into the basis space defined by Φ_y – the same goes for \underline{u} and Φ_u . The projection can be thought of ‘how much is captured’ when going from one space to another. The most intuitive interpretation is to call it a shadow. Just as the shadow of a tree is telling us how much of its 3-D representation can be expressed in 2-D, the projection of \underline{y} onto Φ_y tells us how much of \underline{y} can be described

in Φ_y . However this ‘shadow’ does not tell us if there is some data behind our proverbial tree. This allows multiple \underline{y} s and \underline{u} s to produce the same α and β , so long as the differences in the \underline{y} s and \underline{u} s occur in the null space (outside) of their respective basis spaces.

Substituting these identities into Eq. 1.47, we can re-write it as

$$\Phi_y \alpha = P \Phi_u \beta + \underline{d} \quad (2.42)$$

We then multiply both sides on the left by Φ_y^+ , and given the property in Eq. 2.32, we isolate α as

$$\alpha = \Phi_y^+ P \Phi_u \beta + \Phi_y^+ \underline{d} \quad (2.43)$$

To simplify matters moving forward, we introduce a new $\eta_y \times \eta_u$ matrix system dynamics H which, just as P captured the impact of inputs on output, will describe the impacts of β s on α s

$$H = \Phi_y^+ P \Phi_u \quad (2.44)$$

We then apply the δ operator and recognizing $\Phi_y^+ \underline{d}$ as a constant, drop it out as done in Eq. 1.54

$$\delta_j \alpha = \delta_j H \beta \quad (2.45)$$

Referring back to our section on ILC (1.6), this will all look very similar and one can properly assume our next step is to define our goal output. Recall our goal output \underline{y}^* ; given the identity of Eq. 2.40 we can write a new goal of α^*

$$\alpha^* = \Phi_y^+ \underline{y}^* \quad (2.46)$$

meaning that each trials α_j , now marked with the j subscript to indicate trials, has an

associated error. Calling this error e_{α_j} , we have

$$e_{\alpha_j} = \alpha^* - \alpha_j \quad (2.47)$$

$$= \Phi_y^+ (\underline{y}^* - \underline{y}_j) \quad (2.48)$$

Once again applying the δ operator and following the same logical steps shown in the pure-form ILC derivation, we can write our new ILC Equation in the basis spaces Φ_y and Φ_u

$$e_{\alpha_{j+1}} = I e_{\alpha_j} - H \delta_{j+1} \beta \quad (2.49)$$

Our new model, while still adhering to the $ABCD$ format of Eq. 1.27 and the ILC format of Eq. 1.60, now has controllable dimensions. Just as going from state-space to ILC took our number of states from $n \rightarrow n_{ILC} = pn$, in the transition to basis space we have taken our state count from $n_{ILC} \rightarrow \eta_y$, where $\eta_y \leq n_{ILC}$. The same can be shown for our inputs. It can also be shown at every step of the derivation process that if Φ_y and Φ_u are set to the identity matrix, we perfectly match the earlier full-dimension ILC.

The only thing left now is to define our control law in our basis space. Recall Eq. 1.61

$$\delta_{j+1} \underline{u} = \mathcal{L} e_j \quad (1.61)$$

Substituting $\delta_{j+1} \underline{u} = \delta_{j+1} \Phi_u \beta$ and $e_j = \Phi_y e_{\alpha_j}$

$$\delta_{j+1} \Phi_u \beta = \mathcal{L} \Phi_y e_{\alpha_j} \quad (2.50)$$

we can move the Φ_u outside the δ operator, and left-multiply both side by Φ_u^+ to write

$$\delta_{j+1} \beta = \Phi_u^+ \mathcal{L} \Phi_y e_{\alpha_j} \quad (2.51)$$

Similar to how we defined H , we can now define the $\eta_u \times \eta_y$ controller \mathcal{L}_β

$$\mathcal{L}_\beta = \Phi_u^+ \mathcal{L} \Phi_y \quad (2.52)$$

So that

$$\delta_{j+1} \beta = \mathcal{L}_\beta e_{\alpha_j} \quad (2.53)$$

Armed with this exact model and controller, we can now explore the the relationships between η_y , η_u , Φ_y , Φ_y , \mathcal{L}_β , \underline{y}^* and any resultant error.

Demonstration of Requirements

Recall the assumptions (Eqs. 2.38 and 2.39) we made when deriving our basis space formation. When we called those equations ‘identities’, that was built on the notion that the basis functions were capable of fully capturing the inputs and outputs. Here we wish to explore how strictly we must adhere to these assumptions.

All the following trials will employ the following parameters over 20 trials

$$\mathcal{L}_\beta = 0.5H^+ \quad (2.54)$$

$$p = 100 \quad (2.55)$$

We will use our perfect knowledge to ensure our controller works to control our system within a reasonable number of trials.

An important note moving forward is that some of the shaped outputs look like arbitrary nonsense. That is because they are. In examples where we define our \underline{u}^* to set the \underline{y}^* it is much harder to pick pleasant and recognizable images while still preserving the understandability of our theory. The presented goal ‘shapes’ are no more arbitrary than the circle goal in Figure 1.16 or the word ‘Dartmouth’ in Figure 1.17.

FIPO The first assumption we will relax is that the output basis space captures \underline{y}^* . That is our \underline{u}^* will fully be described in Φ_u , but \underline{y}^* will only partially be in Φ_y . In other words, we will have Full Input, Partial Output Basis Functions (FIPO) describing their respective spaces.

We can always construct a basis space to capture \underline{y}^* by setting one of the functions of Φ_y equal to \underline{y}^* , so this scenario could always be avoided with minimal dimensions. However we must ensure our input can be captured in our input basis space, so we construct our input basis space out of the first 10 chebyshev polynomials⁴, using the method shown in Eq. 2.37.

$$\Phi_u = \begin{bmatrix} T_0 & T_1 & \dots & T_8 & T_9 \end{bmatrix} \quad (2.56)$$

and we will construct \underline{u}^* as done in Eq. 2.39, using a β^* defined as

$$\beta^* = \begin{bmatrix} 1 & 0.2 & -0.3 & 4 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}^T \quad (2.57)$$

which produces the input signals shown in Figures 2.16a and 2.16b. Recall that in our ILC problem, \underline{u} and \underline{y} are stacks of input/output data, rotating through the different components of each (see Eq. 1.63), so we must unstack them for logical interpretation. As a consequence of this ‘stack-to-components’ action, both of the goals end up looking very similar, since they are drawn as alternating components of the same parent signal. This has no impact on the following results, as the inputs are arbitrary regardless.

We apply this \underline{u}^* to our system to create \underline{y}^* . For convenience, we will set $\Phi_y = \Phi_u$. It can be demonstrated that our output basis space does not capture our \underline{y}^* by computing our

⁴Each function must be 200 points in resolution, as our number of ILC inputs is 200 ($p \times r$)

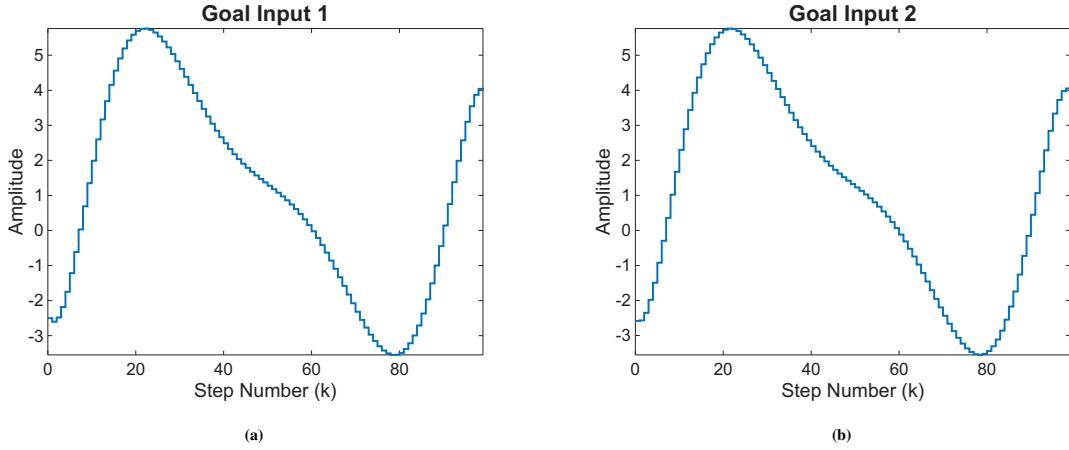


Figure 2.16: Goal Inputs deconstructed from an \underline{u} explicitly constructed from basis functions to be in Φ_u

α^* and attempting to go back to \underline{y}^* . For our system and given input,

$$\alpha^* = \Phi_y^+ \underline{y}^* \quad (2.40)$$

$$= \begin{bmatrix} 0.1364 \\ -0.2043 \\ 0.2136 \\ -0.2672 \\ -0.1991 \\ 0.1067 \\ -0.0212 \\ -0.0241 \\ -0.0025 \\ -0.0320 \end{bmatrix} \quad (2.58)$$

If \underline{y}^* were in Φ_y , then $\underline{y}^* - \Phi_y \alpha^*$ should be a zero vector⁵. However, we see both numerically in Eq. 2.59 and visually in Figures 2.17a and 2.17b that this is not the case.

$$|\underline{y}^* - \Phi_y \alpha^*| = 3.7798 \quad (2.59)$$

⁵Check this yourself using \underline{u}^* and $\Phi_u \beta^*$

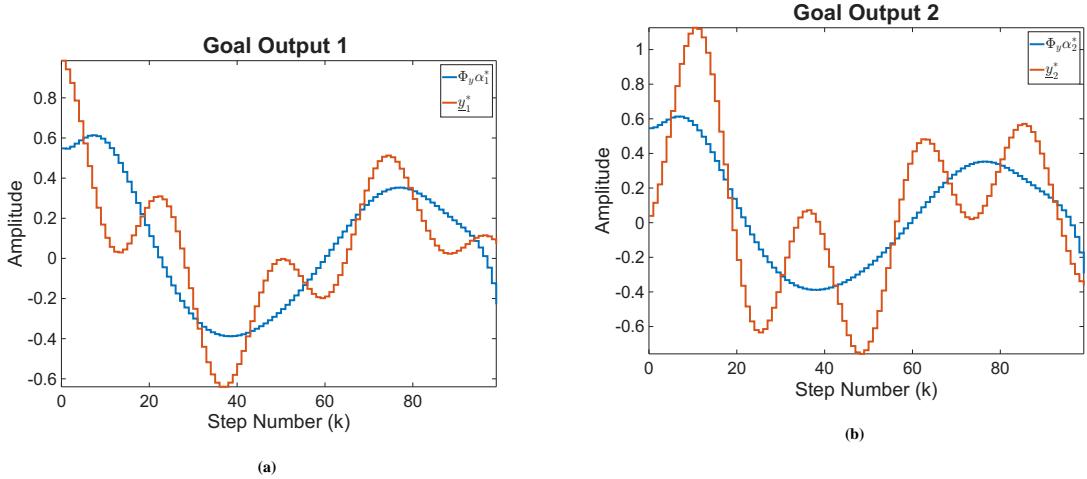


Figure 2.17: Deconstructed y^* 's vs the goal constructed from α^* . α^* is found by inverting the the y^* in the space Φ_y . By then attempting to go back, this highlights the inability of Φ_y to perfectly capture y^* .

Now we have a Φ_u that meets earlier assumptions (input is fully described in its space) and we are correctly violating the assumption that the output is fully defined, we now see how our system performs.

Having defined our goal and our controller – the first steps for any ILC problem, we can proceed with our test. As before we must conduct at least one trial to generate e_0 . If we set $\beta_0 = 0_{\eta_u \times 1}$, then $u_0 = 0_{r_{ILC} \times 1}$ and $\underline{y}_0 = \underline{d}$. Just as before, we can compute $e_0 = \underline{y}^* - \underline{y}_0$ – but now we must convert into the output basis space Φ_y before we can proceed onto the next trial, by $e_{\alpha_0} = \Phi_y^+ e_0$ ⁶.

Armed with our first error, we now iteratively apply our controller. Using the previous trials error in the η_y space and controller \mathcal{L}_β , we compute our change in betas $\delta_j \beta$. We use that to compute our new β_j , convert it from the Φ_u space to get \underline{u} . Applying our new sequence of inputs, our system will produce new outputs to calculate an error with and calculate our e_{α_j} . Repeat this process as needed.

Through these trials, an amazing property emerges. Even though we cannot capture \underline{y}^* perfectly in Φ_y , we are still able to inform our controller with error data that it enables it to

⁶It also works to convert the output into the output basis space and compute $e_{\alpha_j} = \alpha^* - \alpha_j$

send the error to zero. Figures 2.18a and 2.18b show that through trials, the error on both α and β are sent to zero.

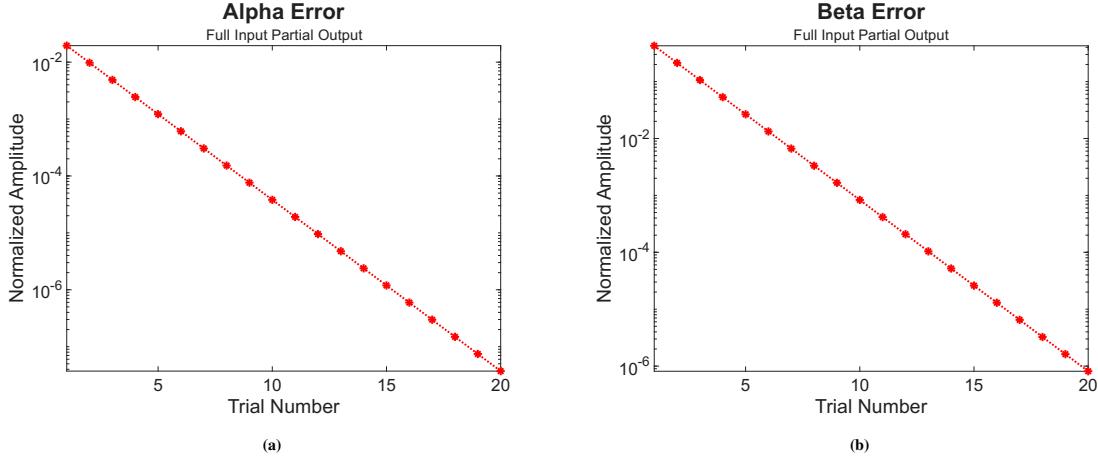


Figure 2.18: Progression of Errors on coefficients through perfect-knowledge controller trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$. Even though $\underline{y}^* \notin \Phi_y$, the associated coefficient errors can still go to zero.

We can additionally see the progression of the coefficients through trials in Figures 2.19a and 2.19b. We intentionally only plot a few of the coefficients to prevent a cluttered plot.

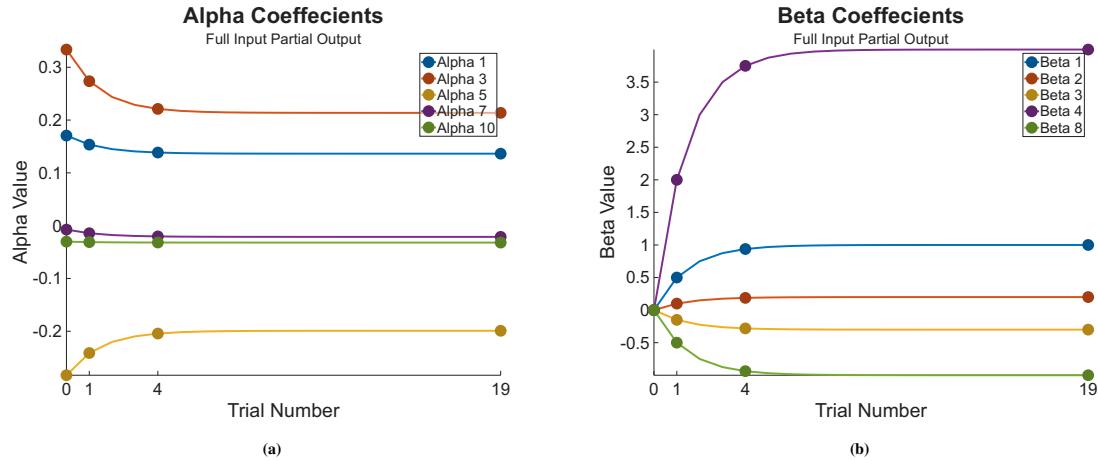


Figure 2.19: Progression of Coefficients through perfect-knowledge controller trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$

Figures 2.20a through 2.20c show that the zero-error in the basis space translates to zero-error in the full-dimension space.

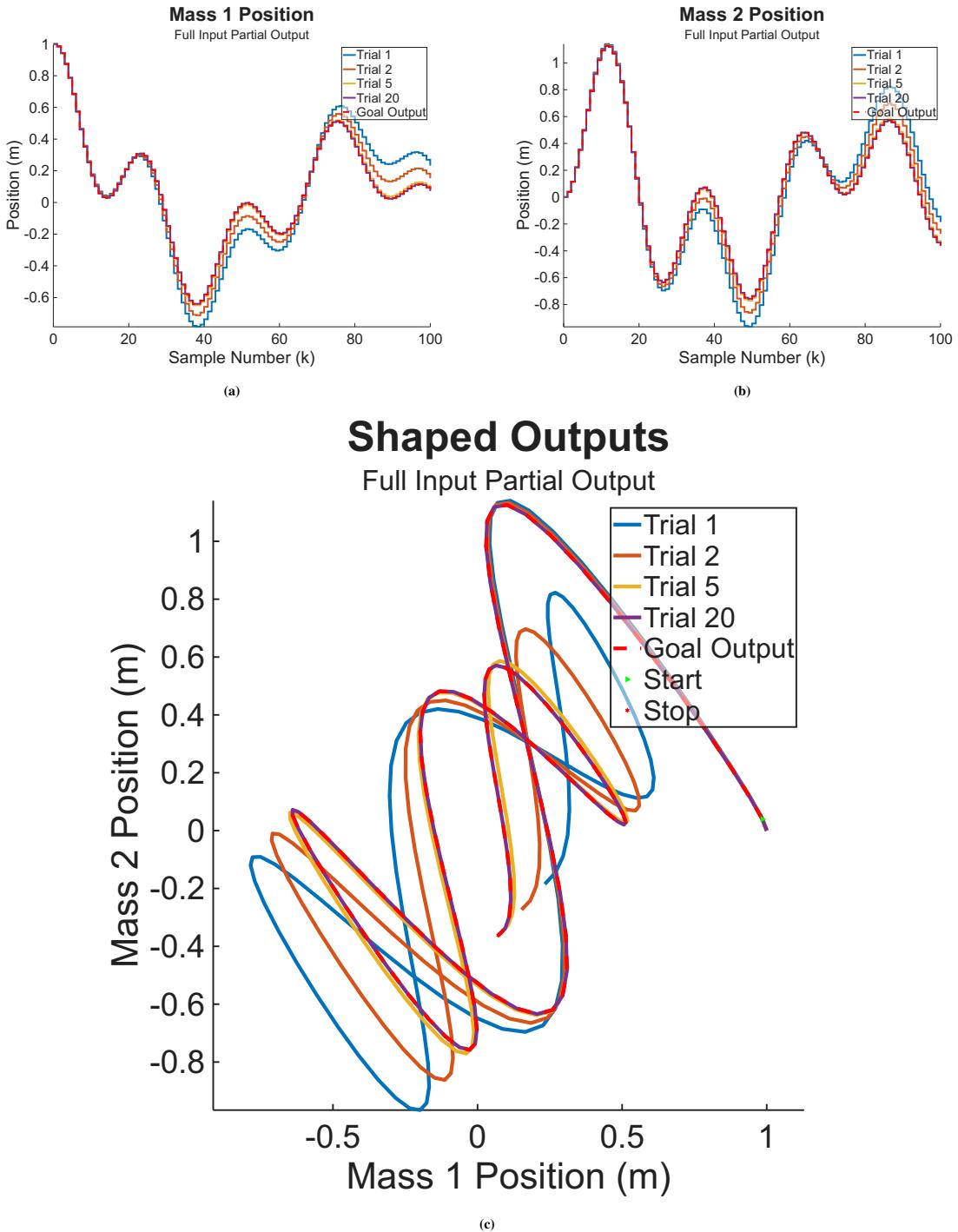


Figure 2.20: Progression of outputs through ILC trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$

PIFO Obviously the next thing to check is what if \underline{y}^* is fully defined in Φ_y but \underline{u}^* is not in Φ_u ? Or in our terminology, there is a set of Partial Input, Full Output Basis Functions (PIFO) describing those spaces. It is incredibly easy to ensure our \underline{y}^* is captured because, if you recall our ideal basis space condition, we can simply set one of our basis functions $\phi = \underline{y}^*$ at the same time that we set \underline{y}^* .

So now our \underline{u}^* will only partially be described in Φ_u , but \underline{y}^* will be fully in Φ_y .

Ensuring we capture \underline{u}^* is understandably much harder without perfect system knowledge. With our current understanding, the only way to guarantee \underline{u}^* is in Φ_u is to expand it to be $r_{ILC} \times r_{ILC}$ and full rank. Obviously this high dimension hurts us, So let us test if it is completely necessary for $\underline{u}^* \in \Phi_u$.

We will repeat a similar process as the above example, except now we define \underline{y}^* first. For consistency, we will re-use the same parameters as those in Eq. 2.57, except now for α^*

$$\alpha^* = \begin{bmatrix} 1 & 0.2 & -0.3 & 4 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}^T \quad (2.60)$$

Remembering to split \underline{y}^* and plotting the goal outputs separately, we see our goals in Figures 2.21a and 2.21b. It should be no surprise that they look exactly the same as the goal inputs in the previous example; they are defined the exact same.

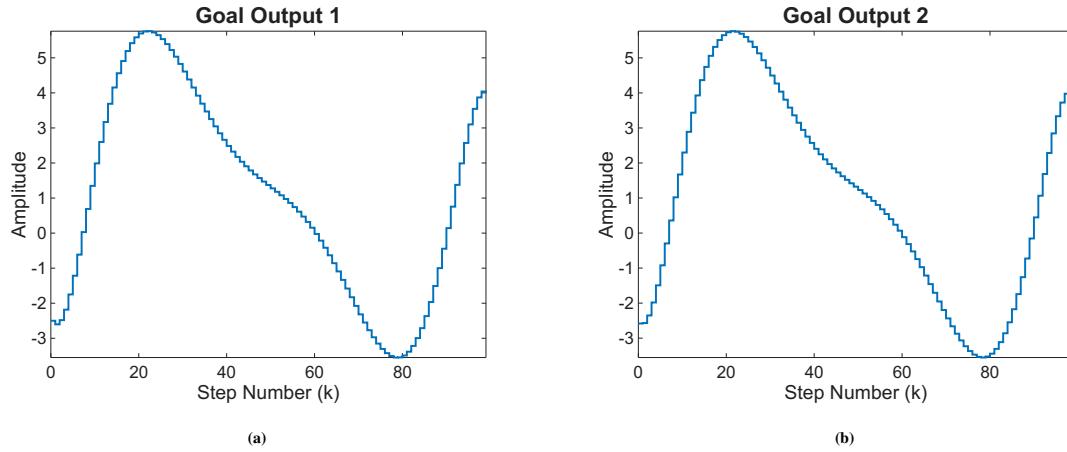


Figure 2.21: Goal Outputs deconstructed from a \underline{y}^* explicitly constructed from basis functions to be in Φ_y

We can back out our input

$$\underline{u}^* = P^+(\underline{y}^* - \underline{d}) \quad (2.61)$$

From there we calculate

$$\beta^* = \Phi_u^+ \underline{u}^* \quad (2.41)$$

$$= \begin{bmatrix} 1,555.4 \\ 4,385.8 \\ 2,871.3 \\ 4,453.6 \\ 2,478.4 \\ 4,508.5 \\ 1,871.0 \\ 4,618.8 \\ 1,124.2 \\ 3,643.1 \end{bmatrix} \quad (2.62)$$

This can be numerically confirmed to not capture our input in Eq. 2.63 and visually in Figures 2.22a and 2.22b

$$|\underline{u}^* - \Phi_u \beta^*| = 1.2514 \times 10^6 \quad (2.63)$$

Having confirmed we have a Φ_u that meets earlier assumptions (input is fully described in its space) and we are correctly violating the assumption that the output is fully defined, we now see how our assumptions perform.

Once again, now that we have our goal and our controller we can proceed to generate e_0 . Unsurprisingly, we follow the exact same steps as before⁷.

Here we begin to see the limits of our basis functions. While figure 2.23a shows that

⁷See Appendix D.10 for a more explicit showing of this

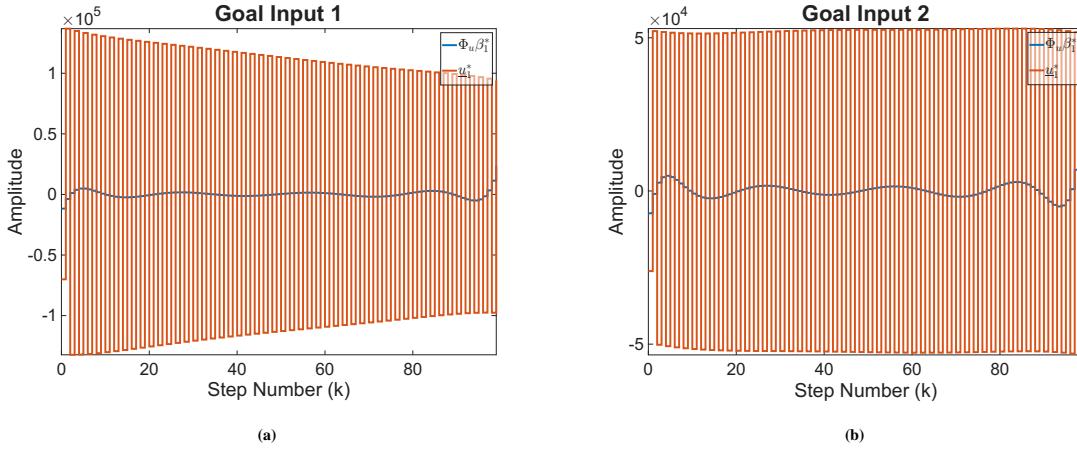


Figure 2.22: Deconstructed Goal \underline{u}^* 's from $\Phi_u \beta^*$, where β^* was backed out of \underline{y}^* with perfect knowledge.

we can send e_α to zero, no other parameter does so. Figure 2.23b shows that through trials, we reach a steady state error. Recall earlier from Eq. 2.40 that multiple \underline{y} s can produce the same α . We are able to generate \underline{y}^* , but there is some additional \underline{y}_{null} that exists outside of Φ_y . So we can capture \underline{y}^* fully in our basis, but cannot see the error that exists outside of it in the null space. Our input is not producing our \underline{y}^* , since our e_α goes to zero, our betas stop learning. The progression of the coefficients through trials can be seen Figures 2.24a

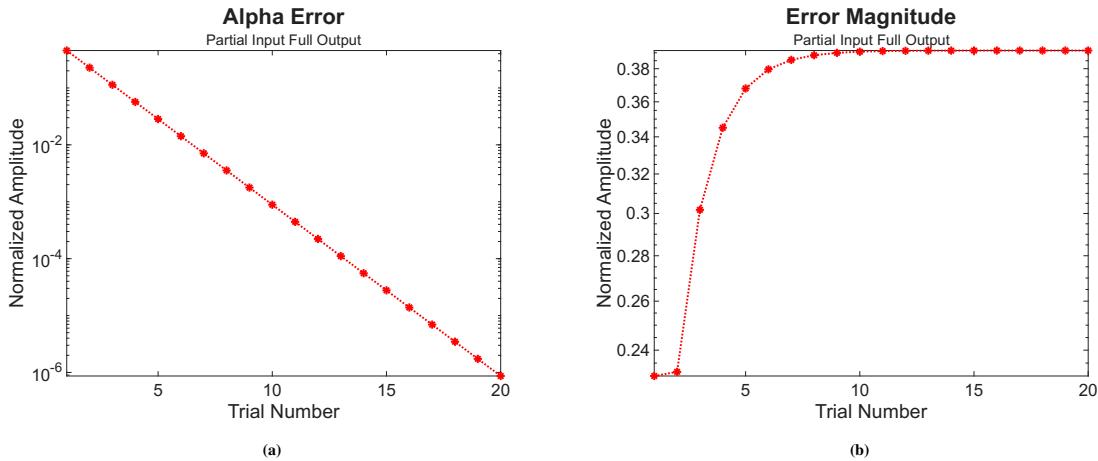


Figure 2.23: Progression of coefficient errors through trials when $\underline{u}^* \notin \Phi_u$ but $\underline{y}^* \in \Phi_y$. Contrast this with the earlier example where $\underline{u}^* \in \Phi_u$, we see now that the error of α can still go to zero, but the error on β reaches a non-zero steady state.

and 2.24b. Once again, only select coefficients are plotted for cleanliness.

Just how drastically off we are can be seen in Figures 2.25a through 2.25c

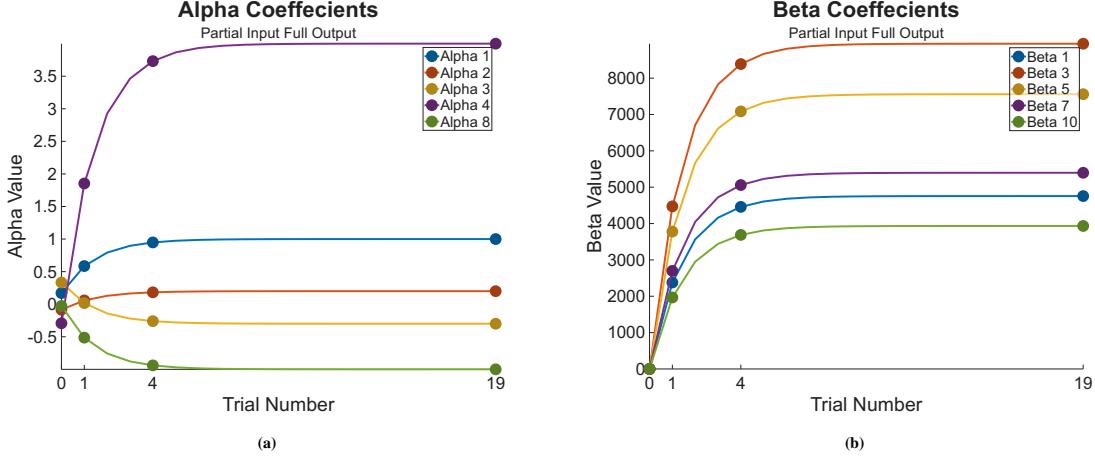


Figure 2.24: Progression of Coefficients through trials when $\underline{u}^* \notin \Phi_u$ but $\underline{y}^* \in \Phi_y$

FIFO vs PIFO Even if the output is not fully captured in a basis space, you may recall that in the ILC problem that is not what we are concerned with. In ILC we deal with the error; more specifically sending the error to zero. No matter our basis space, $e_j = \Phi_y 0_{\eta_y \times 1} = 0_{\ell \times 1}$ due to the zero-matrix identity (just how multiplying by 0 always equals 0). So no matter our goal output, our goal error is always enclosed in the basis space.

It would be natural then to think the same applies for the input. After all, in a properly converged ILC problem $\delta_j \underline{u}$ should also be all zeros. However as shown in our PIFO example, this hope is disproven. Remember that the ILC problem is trying to ‘learn’ the \underline{u}^* that produces \underline{y}^* (Eq. 1.51), and does this by changing inputs between trials, so while the goal $\delta_j \underline{u}$ may be within Φ_u , we need to be able to get there.

This can be readily seen by falling back to the logic of deadbeat controllers. Imagine a scenario where we have a deadbeat \mathcal{L}_β . We start with $\beta_0 = 0_{\eta_u \times 1}$, apply it to produce $\underline{y}_0 = \underline{d}$, and compute $e_{\alpha_0} = \Phi_y (\underline{y}^* - \underline{d})$. We follow our control law of Eq. 2.53 to compute $\delta_1 \beta = \mathcal{L}_\beta e_{\alpha_0}$, such that $\beta_1 = \beta_0 + \delta_1 \beta$. Given that β_0 was a zero-vector, we can ignore it, and since we are saying that \mathcal{L}_β is deadbeat, we can say that β_1 is our best attempt at β^* , meaning our best attempt at \underline{u}^* must be in Φ_u .

If one were to repeat this example with a non-deadbeat controller, they would find that any β found would always produce a \underline{u} in the space Φ_u . So while it is possible to always

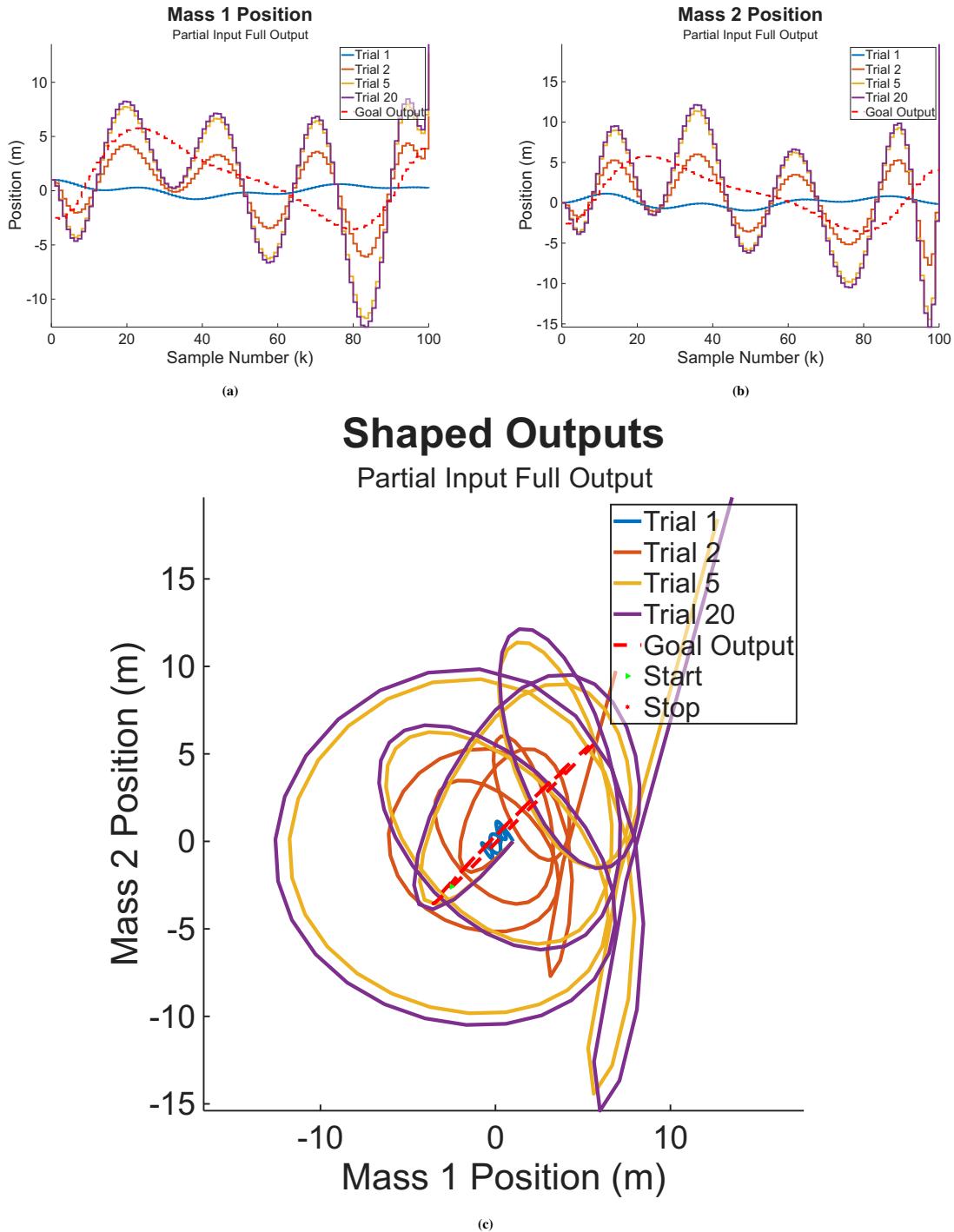


Figure 2.25: Progression of Outputs through ILC trials when $\underline{u}^* \in \Phi_u$ but $\underline{y}^* \notin \Phi_y$

achieve a Φ_y that captures our goal of zero error, the space defining the inputs must be selected carefully to include \underline{u}^* . It is also important to note that the learned input is **not** the projection of \underline{u}^* onto Φ_u ⁸.

So our Φ_u matters, but Φ_y does not⁹. Given that, why not define Φ_y such that $\eta_y = 1$. Even if we then leave Φ_u as the identity matrix, then thinking ahead to our RL problem, we would have the ability to infinitely reduce our ‘state’ dimensions down to 1. Then input-decoupling could be employed such that each controller could be updated in just 4 trials, and that would only then need to be done r_{ILC} times.

FISO To test this theory, we will setup our problem very much like we did the FIPO example. β^* will once again be defined as it is in Eq. 2.57, with our Φ_u similarly following Eq. 2.56. We will define $\underline{y}^* = P\underline{u}^* + \underline{d}$. So we have a set of Full Input, Single Output Basis Functions (FISO) describing those spaces.

We have already seen this input sequence in Figures 2.16a and 2.16b, and show the outputs in Figures 2.26a and 2.26b. However, the learning history is quite different. Even

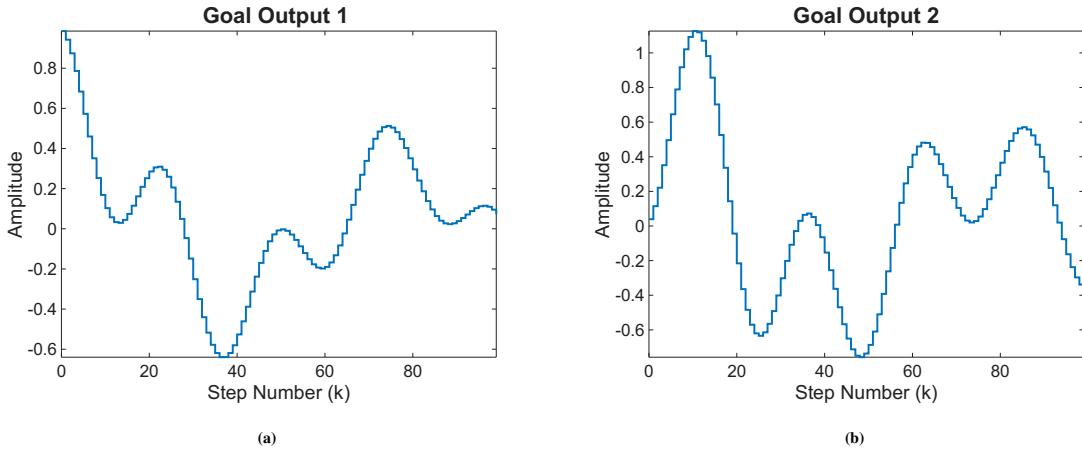


Figure 2.26: Deconstructed Goal Outputs for $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \Phi_y$

though we appear to meet all the criteria of capturing our signals in our basis space, we are unable to send the error to zero. It may look ok, but the average error on the output of each

⁸Unless $\underline{u}^* \in \Phi_u$ of course

⁹This is consistent with the findings of Phan and Frueh 1996

point¹⁰ is 0.9823

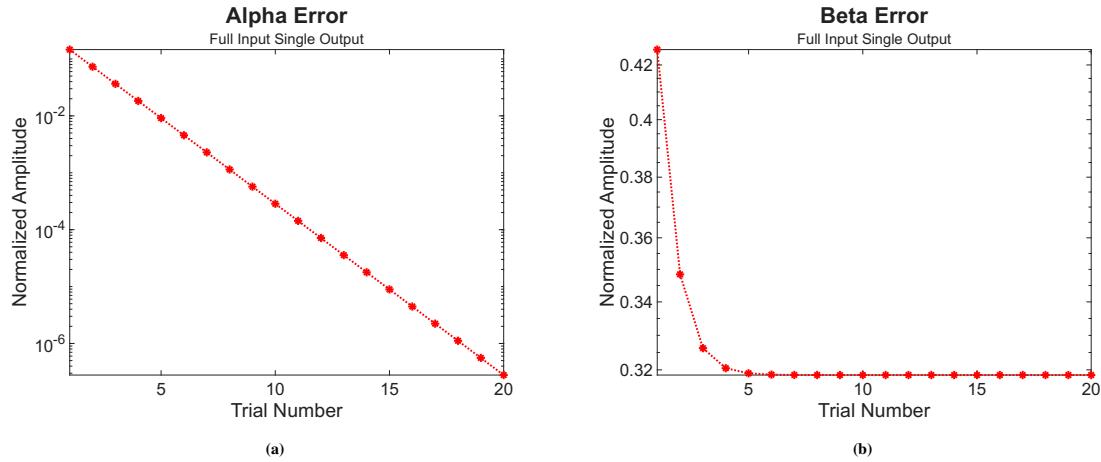


Figure 2.27: Progression of coefficient errors through trials when $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \Phi_y$

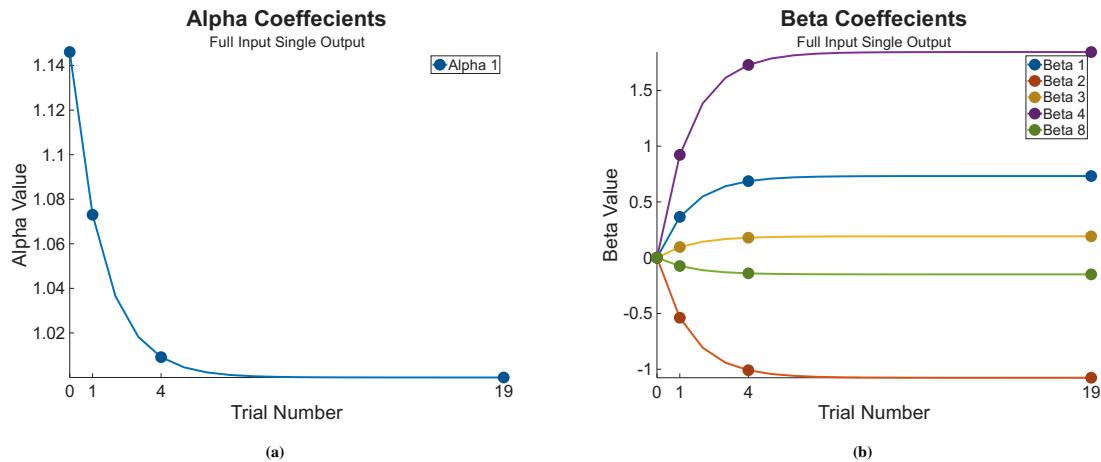


Figure 2.28: Progression of coefficients through trials when $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \Phi_y$

We see that we are still able to send $e_\alpha \rightarrow 0$, or $\alpha \rightarrow 1$, but the other components do not follow. Once again we are faced with this steady-state error on the beta.

¹⁰Computed as $\frac{|e|}{n_{ILC}}$

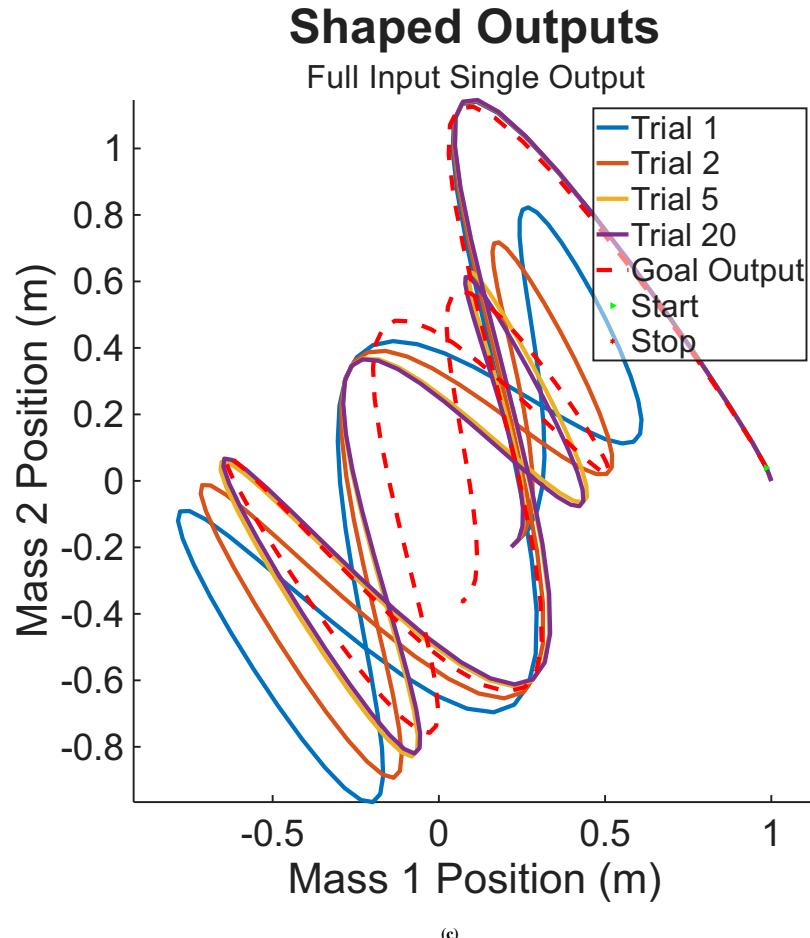
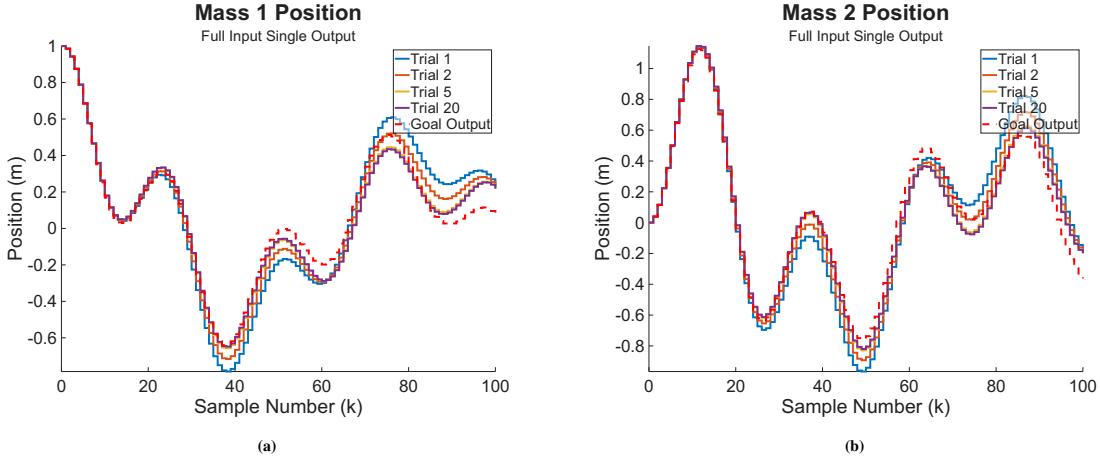


Figure 2.29: Progression of Outputs through ILC trials when $\underline{u}^* \in \Phi_u$ and $\underline{y}^* = \Phi_y$. Even though \underline{u}^* could in theory be fully described by Φ_u , the controller is unable to capture \underline{y}^*

SIFO We now check the other scenario, where the spaces can be described with a set of Single Input, Full Output Basis Functions (SIFO). This is an extreme version of our POFI example, so we set it up the same. The difference being we extract \underline{u}^* in Eq. 2.61, we also use that to set our input basis space such that $\underline{u}^* = \Phi_u$. The goals can be seen in Figures 2.21a and 2.21b, and the inputs that get us there in Figures 2.30a and 2.30b – nothing new here.

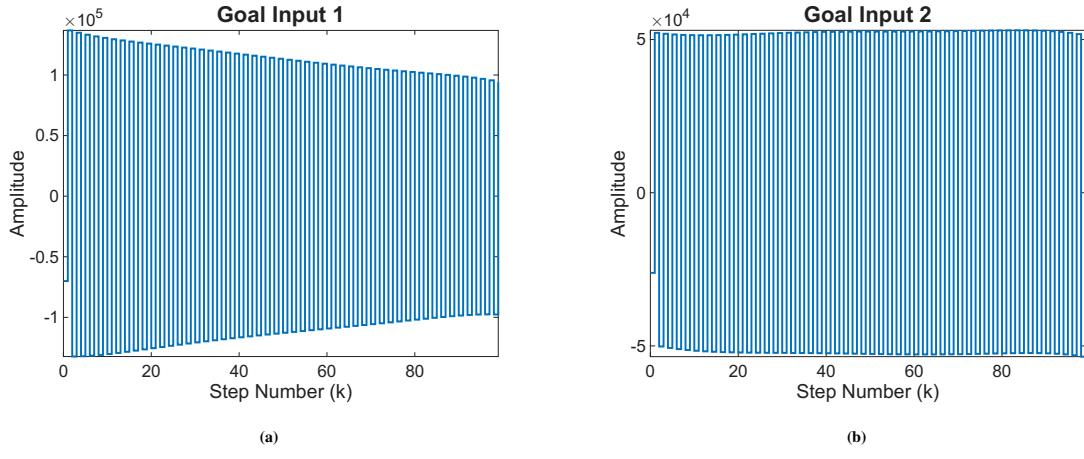


Figure 2.30: Deconstructed Goal Inputs for $\underline{u}^* = \Phi_u$ and $\underline{y}^* \in \Phi_y$

What is new, however, is the learning process. We see we are able to actually find our output, with the average error on each point being 8.85×10^{-5} - within our ‘zero’ range.

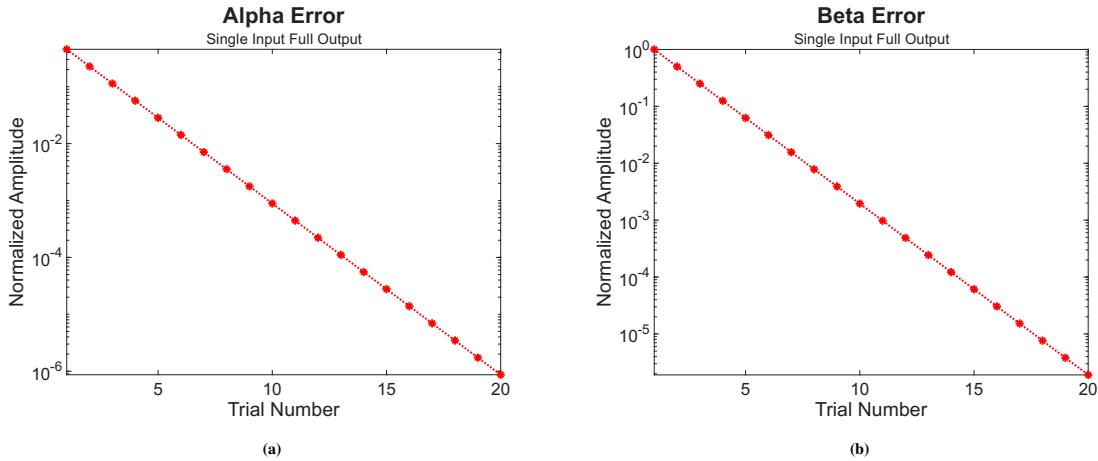


Figure 2.31: Progression of coefficient errors through trials when $\underline{u}^* = \Phi_u$ and $\underline{y}^* \in \Phi_y$

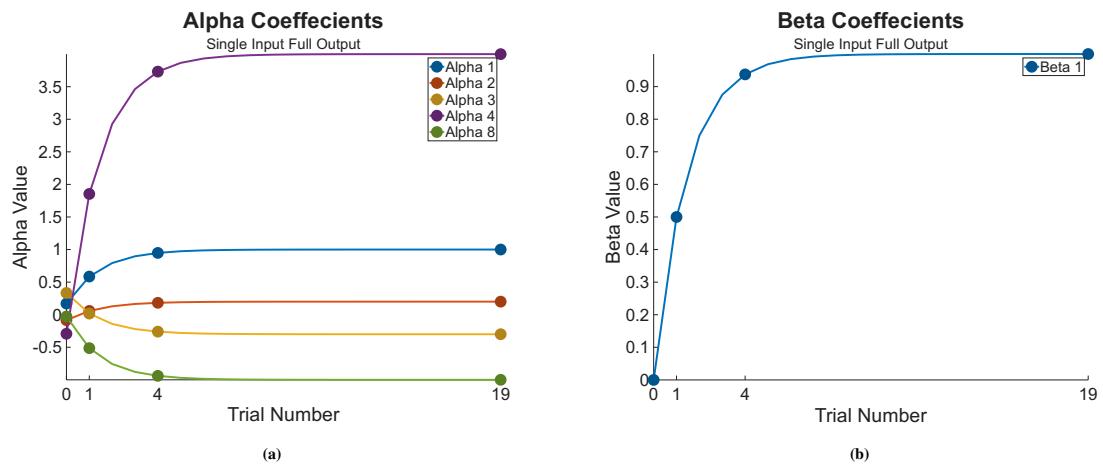
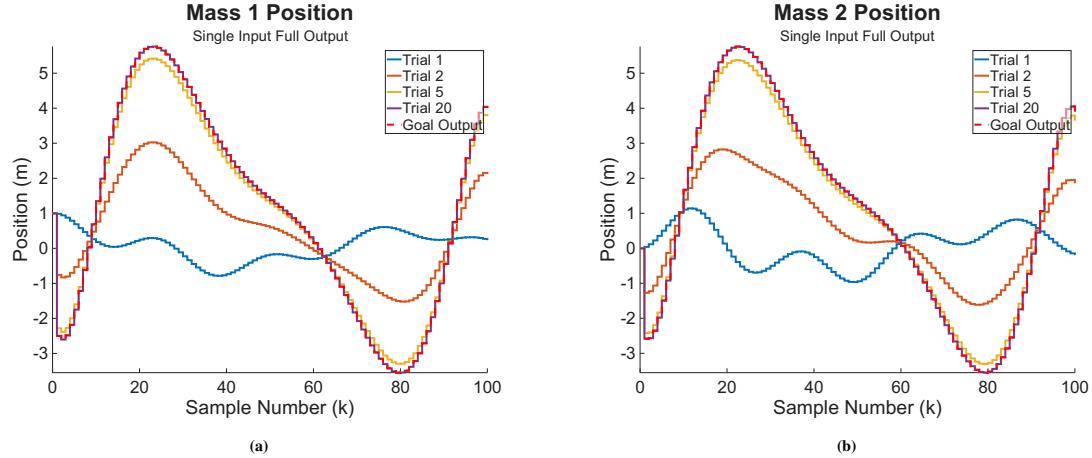


Figure 2.32: Progression of coefficients through trials when $\underline{u}^* = \Phi_u$ and $\underline{y}^* \in \Phi_y$



Shaped Outputs

Single Input Full Output

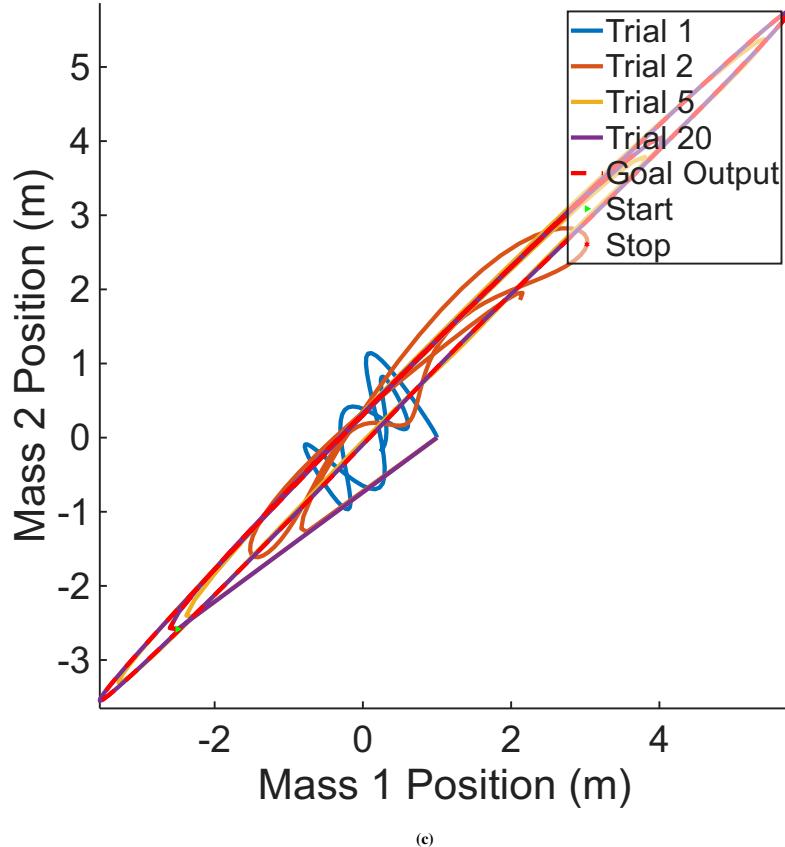


Figure 2.33: Progression of Positions through ILC trials when $u^* = \Phi_u$ and $y^* \in \Phi_y$. While the learned shape may appear arbitrary, the important take-away is that the error is zero, and the learned output matches that exactly of the goal.

FISO vs SIFO What this shows us is that in addition to our conditions on Φ_u , and lack thereof on Φ_y , we must meet certain criteria when selecting our dimensions. It can be shown that condition is $\eta_y \geq \eta_u$.

If we refer back to our $\eta_u \times \eta_y$ controller in Eq. 2.53, this makes sense.¹¹ For robust control, we cannot ask a controller to produce more outputs than inputs if we want those controller outputs to be the best system inputs.

Summary

We have established two conditions and one freedom. First, given that a \underline{u}^* exists, our input basis space Φ_u must include it.

$$\underline{u}^* \in \text{span}(\Phi_u) \quad (2.64)$$

Our second condition is we must have as many, if not more, basis functions describing the output than we do the input of the system.

$$\eta_y \geq \eta_u \quad (2.65)$$

The only afforded freedom is that Φ_y does not need to be chosen wisely, and will always allow for a controller to send the error to zero.

2.2.4 Creating a Dynamic Basis Space from Learned Inputs

To effectively learn, we are limited by our ability to capture \underline{u}^* . For every reduction we can make to η_u , we can similarly reduce η_y . The idea is to then determine the best possible input for a set of basis functions, then change our basis functions to include that learned input.

For a signal of length ℓ , we will at most need ℓ basis functions. To start, we define a

¹¹The wording here will get a little messy, as the input to a controller is the output of a system, and the output of a controller is the input to the system, so pay close attention to that.

complete basis space to draw from that we know will capture the entire input. As we now know that the distinction between Φ_y and Φ_u does not matter, we will now use the same notation of Φ for both¹².

$$\Phi_{full} = \begin{bmatrix} | & | & & | \\ \phi_1 & \phi_2 & \cdots & \phi_\ell \\ | & | & & | \end{bmatrix} \quad (2.66)$$

Next we define our η s. As per the condition stated in Eq. 2.65, we will set $\eta_y = \eta_u$ and use η for both now. This is both out of convenience and efficiency.

This is all we need to begin our process. To set some notation and terminology, we will refer to each attempt with a new basis space as an ‘episode’. Each episode will have its own $\ell \times \eta$ basis space Φ^e , and will learn an input \underline{u}^e ¹³.

After each episode, we take the learned input \underline{u}^e and set that as a basis function. We then update the remaining $\eta - 1$ basis functions with new ϕ s from Φ_{full} , as shown in Eq 2.67.

$$\Phi^{e+1} = \begin{bmatrix} | & | & & | \\ \underline{u}^e & \phi_j & \cdots & \phi_{j+\eta-1} \\ | & | & & | \end{bmatrix} \quad (2.67)$$

where $1 \leq j \leq \ell$, marking what basis function we draw from Φ_{full} , and ‘rolls-over’ as needed. So if $p = \ell = 10$, a j -sequence would be $j = 1, 2, \dots, 10, 1, \dots$.

One would hope that after sufficient episodes, there would be a Φ^e found such that \underline{u}^* is included.

Example – Rolling Basis Space

First we set up the problem as done in the FIPO example, with \underline{u}^* shown in Figures 2.16a and 2.16b and y^* in Figures 2.26a and 2.26b.

¹²For our given examples we have a 2-input, 2-output system such that $n_{ILC} = r_{ILC}$ but this is not always the case. In that case, Φ_y still does not matter, but may need to be defined in different dimensions and thus not be able to equal Φ_u .

¹³This is the input which generates $e_\alpha = 0$.

Although we have a signal of length $\ell = 200$ ¹⁴, we do not need ℓ basis functions to ensure we capture our \underline{u}^* . Recall for our FIPO example, we defined \underline{u}^* in the space of Chebyshev Polynomials $T_0 \rightarrow T_9$, so we can define Φ_{full} as we define Φ_u in Eq. 2.56 to ensure we capture \underline{u}^* . If we did not know this to be the case, we would have to define Φ_{full} to be a full rank 200×200 matrix to ensure we spanned the whole input space.

$$\Phi_{full} = \begin{bmatrix} T_0 & T_1 & \cdots & T_9 \end{bmatrix} \quad (2.68)$$

We next set our η , and for this example $\eta = 4$. For our first episode we do not have a previously learned input, so we will use a basis function.

$$\Phi^0 = \begin{bmatrix} T_0 & T_1 & T_2 & T_3 \end{bmatrix} \quad (2.69)$$

Recall that our \underline{u}^* is defined with weights on $T_0 \rightarrow T_3$, but also T_7 . If the learned input for a given basis space was the projection of \underline{u}^* onto that basis space, we would expect $\beta_{hoped}^e = \begin{bmatrix} 1 & 0.2 & -0.3 & 4 \end{bmatrix}^T$. As previously stated, this is not the case and so we end up with $\beta^e = \begin{bmatrix} 1.2596 & 0.7884 & 0.0403 & 4.2582 \end{bmatrix}^T$

The controller found a way to send e_α (as shown in Figure. 2.34) just as it was able to in our PIFO example. This does not contradict with any of our earlier observations.

We now construct a new Φ with the learned input as one of the basis functions

$$\Phi^1 = \begin{bmatrix} \underline{u}^0 & T_4 & T_5 & T_6 \end{bmatrix} \quad (2.70)$$

and repeat the process. Since we are using $\Phi = \Phi_u = \Phi_y$, make sure to update both basis spaces. If Φ_y is left fixed, although it does not matter, learning will be sub-optimal. This is because our first pass finds the \underline{u} that sends e_{α_0} , and so if we do not update Φ_y , then that same learned input will be applied again with the same results - the system will see no

¹⁴ $n_{ILC} = r_{ILC} = 200$

added benefit from including new inputs.

We do this enough times to try every basis function in Φ_{full} multiple times and see that each time while we are able to send $e_\alpha \rightarrow 0$, we never find \underline{u}^* nor send $e \rightarrow 0$.

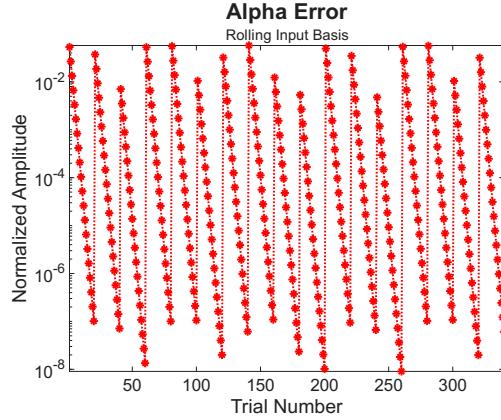


Figure 2.34: Progression of e_α through rolling basis space episodes

It can be seen that this approach does come close to producing our goal output in Figure 2.36c, and can be tempting to say it works. While it may be acceptable in some processes, it is not the exact model we are seeking. This is most evident by inspecting the learned inputs in Figures 2.36a and 2.36b, and seeing they do not match \underline{u}^* . Additionally, we have a final error magnitude $|e| = 0.0132^{15}$.

Figure 2.35 shows that after our first pass through all of the input basis we have a ‘pretty good’ input. This is logical, as it has now taken into consideration every possible basis function that makes up our \underline{u}^* . However, Figure 2.37 highlights the fact that while after the controller has been exposed to all the necessary information to span the space, it still attempts to make slight modifications to the learned input of the previous trial – it fails to find the \underline{u}^* .

In Figure 2.38 we ‘force’ the learned input that goes on to make up the next Φ to be exactly \underline{u}^* . It can be seen that the system, if it finds the proper \underline{u}^* in its cycle will properly stay converged on that solution, so the prior example clearly did not find our goal input.

¹⁵Recall that Matlab ‘zero’ $\leq 1 \times 10^{-6}$

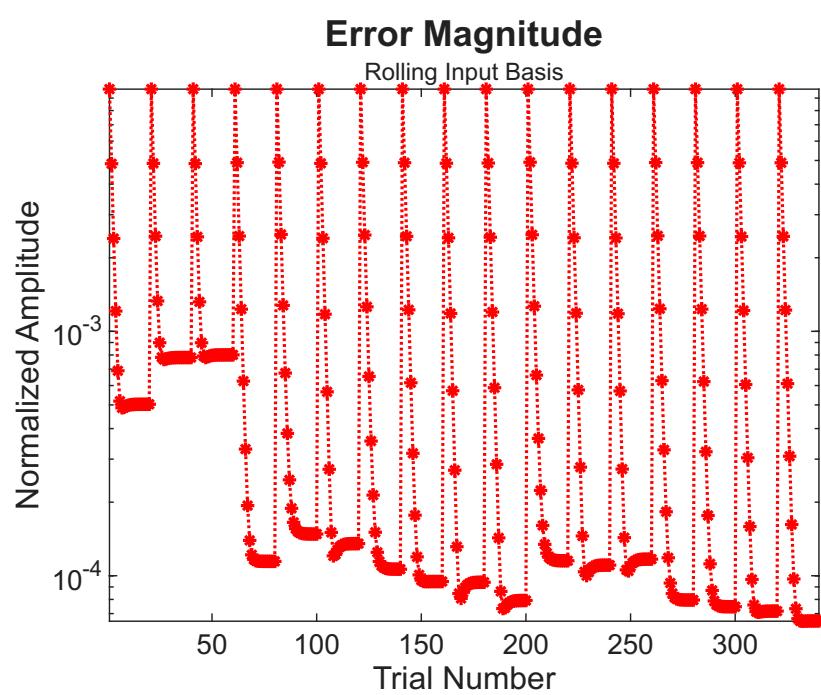


Figure 2.35: Progression of e through rolling basis space episodes

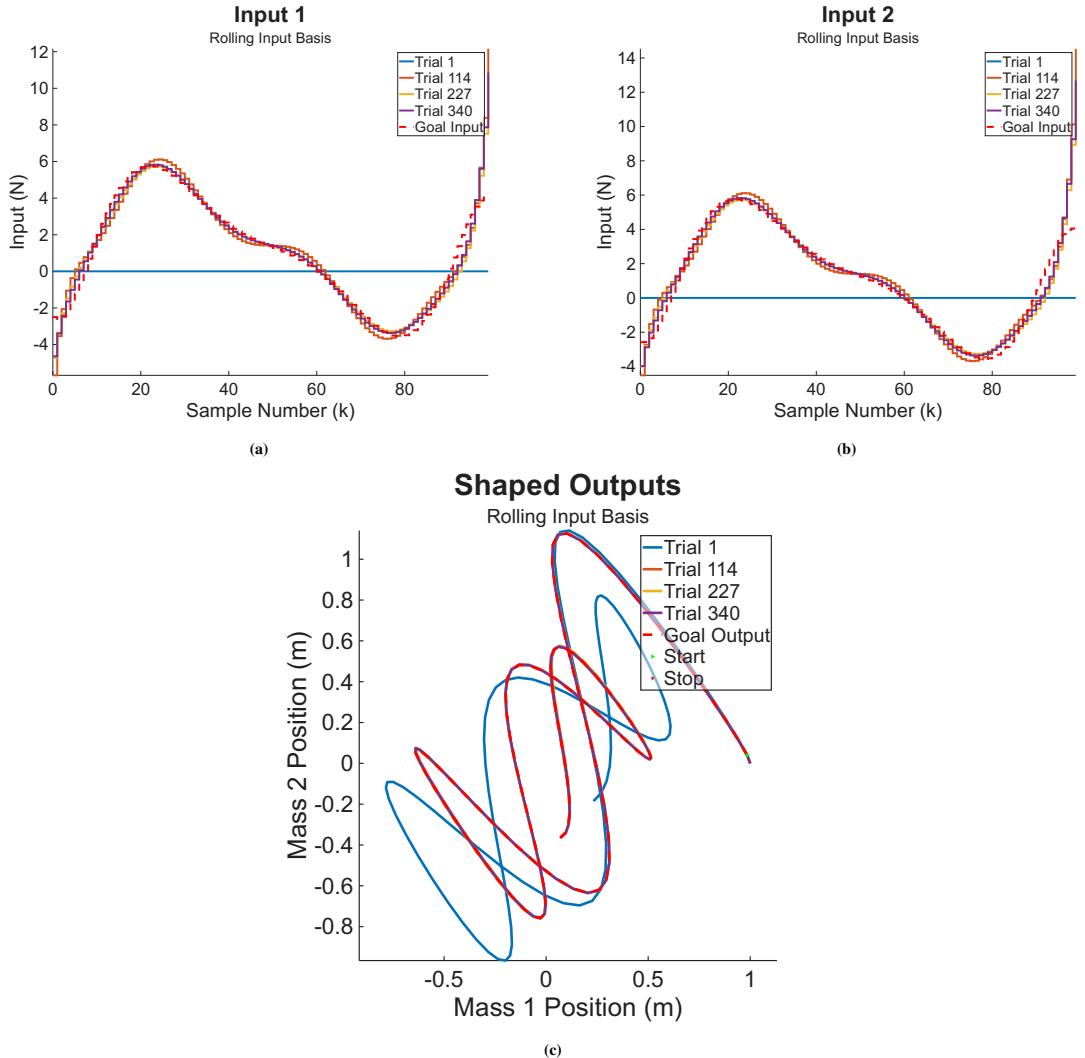


Figure 2.36: Progression of deconstructed goal inputs and the generated shaped output through rolling basis space episodes. While the output can be seen to be close to our goal, even after working our way through the entirety of the input basis space, we still fail to properly capture our y^* . Failure to capture u^* can best be seen on the fringes of the input signals.

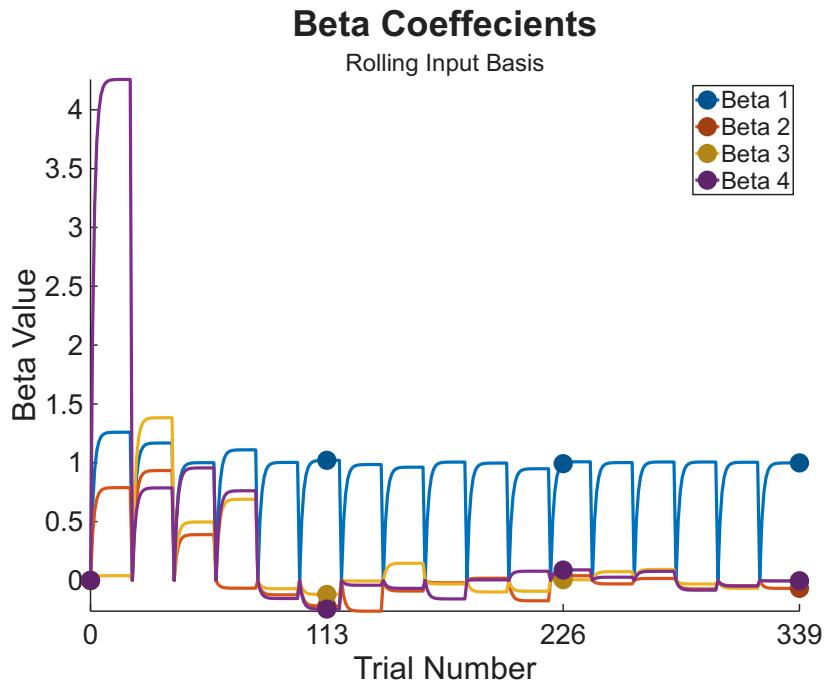


Figure 2.37: Progression of β through rolling basis space episodes. After sufficient trials have been made to cover all of Φ_{full} , it can be seen that β_1 sits around a value of 1 - meaning that the learned input is predominately that of the previously learned pass. However, there are still tiny additions from other components, evident by the non-zero β_{2-4} .

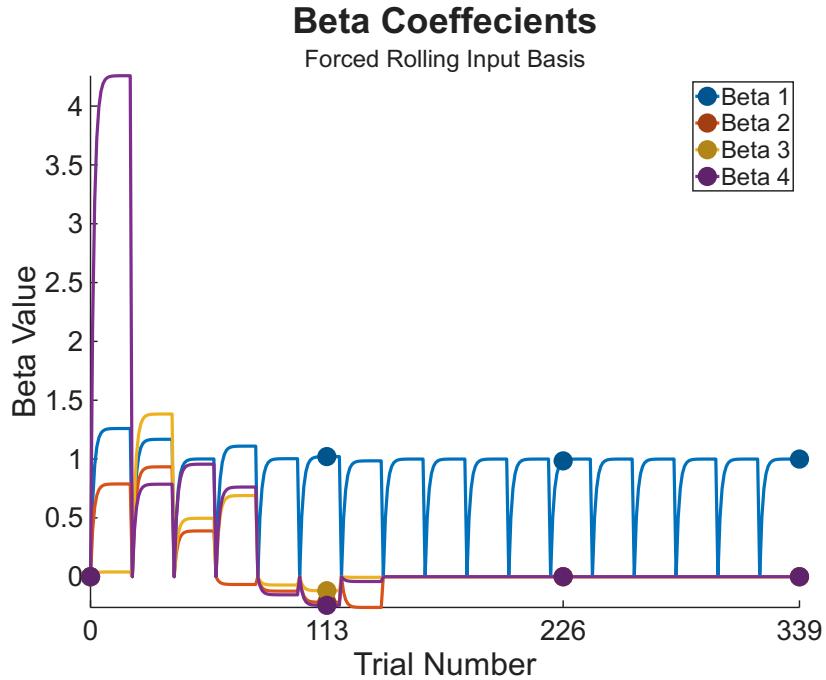


Figure 2.38: Progression of β through rolling basis space episodes. After 7 iterations, Φ is forced to have \underline{u}^* in it. It is demonstrated that the controller continues to properly identify that as the correct input. It can be seen, given this, that the earlier example never actually finds \underline{u}^* before being moved away – once \underline{u}^* is found, it is not lost.

Summary of Dynamic Spaces

While it is possible to start with a finite number, η , of basis functions ϕ to create a dynamic basis space which better captures \underline{u} , there is no guarantee to create one that captures \underline{u}^* .

There are two main reasons for the failings of this approach. Recall that there are numerous \underline{y} s that can produce the same same α for a given Φ . Additionally, while our basis functions may be independent of one another, as stated in Eq. 2.29, they are not necessarily independent in the response they illicit from the system in the basis space. So in addition to the different \underline{y} s that can get us α^* , there are also multiple \underline{u} s to get us to α^* . Since different inputs can lead to the same ‘perceived’ output through the lens of a basis space, defining our basis space and functions as such do not ensure that we are able to find the optimal \underline{u}^* .

2.2.5 Summary of Basis Function

Basis Functions offer some path forward for dimension reduction, but currently it is not an exact one. We have shown that the number of basis functions on the system must adhere by $\eta_u \leq \eta_u$, and that our input basis space Φ_u must capture \underline{u}^* – no such condition exists for \underline{y}^* and Φ_y . We have additionally shown that while it is possible to work with a dynamic basis space Φ , the possibility that different inputs can create the same α – or projection of \underline{y} on Φ_y – makes exact control impossible.

2.3 Conjugate Basis Functions

Since the major challenge to being able to dynamically select our input basis functions is their ability to produce similar outputs, we then want to find a set of inputs which produce independent (or orthogonal) outputs on a system.

2.3.1 What are Conjugate Basis Functions

Conjugate Basis Functions are a special case of Basis Functions, defined for a system and cost functions. Whereas before we had functions that only had to be independent of each other, now the functions must be independent in the output they produce and the cost they incur.

Conjugate Basis Functions have all the same properties shown in **What are Basis Functions** (Section 2.2.1), and then some. As Frueh and Phan describe in their paper on Linear Quadratic Optimal Learning Control (LQL)¹⁶, the goal is to define basis functions such that when they are added to pre-existing basis space Φ , the optimality of the functions already included does not change. So the parameters generated from one episode are a subset of those from the next.

Suppose we have e episodes worth of data, and we run one more episode to improve our Φ and β . The properties of LQL dictate that

$$\Phi^{e+1} = \begin{bmatrix} \Phi^e & \phi_{e+1} \end{bmatrix} \quad \beta^{e+1} = \begin{bmatrix} \beta^e \\ \beta_{e+1} \end{bmatrix} \quad (2.71)$$

See that if \underline{u}^e was the optimal input for Φ^e , then \underline{u}^{e+1} will be optimal for Φ^{e+1} and

$$\underline{u}^{e+1} = \underline{u}^e + \phi_{e+1}\beta_{e+1} \quad (2.72)$$

¹⁶J.A. Frueh and M.Q. Phan (1998). “Linear quadratic optimal learning control (LQL)”. in: *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*. Vol. 1, 678–683 vol.1. DOI: 10.1109/CDC.1998.760762.

such that our pre-existing optimal input components do not change from the addition of a new basis function.

2.3.2 Deriving Conjugate Basis Functions

To derive our Conjugate Basis Functions, we must first define a cost function. Recall we want to send our error e to zero via control of $\delta \underline{u}$. In the converged ILC state, both e and $\delta \underline{u}$ are 0. We will define our cost-per-episode the same way we defined our utility function for the ILC problem in Eq. 2.1, except expressing it as J instead of U_j

$$J = \delta_j \underline{u}^T R \delta_j \underline{u} + e_j^T Q e_j \quad (2.73)$$

where R is some $r_{ILC} \times r_{ILC}$ cost matrix, and Q a $n_{ILC} \times n_{ILC}$ one.

We want to find a $\delta_j \beta$ then to minimize this cost function, as with all our previous controls. We begin with the following relationships

$$\begin{aligned} \delta_j \underline{u} &= \Phi(\beta_{j-1} + \delta_j \beta) \\ e_j &= e_{j-1} - P \delta_j \underline{u} \end{aligned} \quad (2.74)$$

and substitute them into Eq. 2.73

$$J = (\Phi(\beta_{j-1} + \delta_j \beta))^T R (\Phi(\beta_{j-1} + \delta_j \beta)) + (e_{j-1} - P \delta_j \underline{u})^T Q (e_{j-1} - P \delta_j \underline{u}) \quad (2.75)$$

For reasons that will be seen shortly, as well as in the efforts of minimizing page space, we will define matrices R_b and \mathcal{C}

$$R_b = \Phi^T R \Phi \quad (2.76)$$

$$\mathcal{C} = R_b + \Phi^T P^T Q P \Phi \quad (2.77)$$

By setting $\frac{\partial J}{\partial \delta_j \beta} = 0$, we can solve for the $\delta_j \beta$ which minimizes our cost function as

$$\begin{aligned}\delta_j \beta &= (-2\mathcal{C})^{-1} 2\mathcal{C} \beta_{j-1} + (2\mathcal{C})^{-1} 2\Phi^T P^T Q e_{j-1} \\ &= -\beta_{j-1} + \mathcal{C}^{-1} \Phi^T P^T Q e_{j-1}\end{aligned}\tag{2.78}$$

By the definition of the δ operator, we can write

$$\beta_j = \mathcal{C}^{-1} \Phi^T P^T Q e_{j-1}\tag{2.79}$$

We will make one final step to simplify future logic, and that is assume every attempt is a deadbeat attempt. Meaning $j = 1$ always. This enables us to substitute e_{j-1} as our open loop error e_0 , and do away with episodic β notation:

$$\beta = \mathcal{C}^{-1} \Phi^T P^T Q (\underline{y}^* - \underline{d})\tag{2.80}$$

where \underline{d} can be collected from an open-loop response ($\underline{u} = 0$). The complete derivation can be seen in Appendix A.1.

Suppose we set $\eta = 2$. Our basis space Φ would then have 2 functions ϕ and we would have 2 β s:

$$\Phi = \begin{bmatrix} \phi_1 & \phi_2 \end{bmatrix}\tag{2.81}$$

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} \mathcal{C}_1^{-1} \phi_1^T P^T Q e_{j-1} \\ \mathcal{C}_2^{-1} \phi_2^T P^T Q e_{j-1} \end{bmatrix}\tag{2.82}$$

where \mathcal{C}_1 and \mathcal{C}_2 are rows 1 and 2 of \mathcal{C}^{-1}

We would like it that if we expanded Φ to include ϕ_3 , that β_3 would be generated without impacting β_1 and β_2 – as by the goal property of conjugate basis functions shown in Eq. 2.71. Yet \mathcal{C} is a function of Φ so we run the risk of our previous β s changing when Φ changes - that is unless \mathcal{C} is somehow defined to behave as a constant. In the world of

matrix math, that is any diagonal matrix. For our purposes, we will use the identity matrix for simplicity.

This brings us to our ‘conjunctionality condition’:

$$\mathcal{C} = I_{\eta \times \eta} \quad (2.83)$$

which when written in terms of our basis space Φ

$$\Phi^T (R + P^T Q P) \Phi = I_{\eta \times \eta} \quad (2.84)$$

or if explicitly written by ϕ_i

$$\phi_i^T (R + P^T Q P) \phi_j = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases} \quad (2.85)$$

So long as the condition in Eq. 2.84 is met, we can create β s which solely depend on their associated ϕ , are optimal to minimize our cost function 2.73, and do not impact the optimality of other $\phi_i \beta_i$ inputs as we previously saw in 2.2.4.

$$\beta_i = \phi_i^T P Q (\underline{y}^* - \underline{d}) \quad (2.86)$$

2.3.3 Extracting Conjugate Basis Functions

The issue we face now is we do not know P , so we must derive a method to extract β . We of course need a collection of input-output. We can more easily show the underlying logic with the batch approach, then we will explore a more computationally and logically efficient iterative approach.

The Batch Approach

Suppose we have a collection of episodic data for both input \underline{u}_i^e and their associated output \underline{y}_i^e where $i = 0, 1, \dots, b$ and b is the number of conjugate basis functions to generate. We can apply our δ operator to this data set to produce $\delta_1 \underline{u}^e, \delta_2 \underline{u}^e, \dots, \delta_b \underline{u}^e$ and the same sequence for our outputs. We will define the batch notation $[\underline{x}^e]_b$

$$[\underline{x}^e]_b = [x_1^e, x_2^e, \dots, x_b^e] \quad (2.87)$$

so that $[\delta \underline{u}^e]_b$ is all the changes in our episodes inputs, and $[\delta \underline{y}^e]_b$ the change in outputs. We can apply this delta-batch operation to Eq. 1.47 to show:

$$[\delta \underline{y}^e]_b = P[\delta \underline{u}^e]_b \quad (2.88)$$

As with all earlier examples, the δ operator removes the existence of a \underline{d} term.

We now want to find conjugate functions spanned by our inputs $[\delta \underline{u}^e]_b$ in our conjugate space Φ . That is:

$$[\delta \underline{u}^e]_b = \Phi[\rho]_b \quad (2.89)$$

We create matrix \mathbf{W} from this delta-batch data

$$\mathbf{W} = [\delta \underline{u}^e]_b^T R [\delta \underline{u}^e]_b + [\delta \underline{y}^e]_b^T Q [\delta \underline{y}^e]_b \quad (2.90)$$

substituting in Eq. 2.88 and Eq. 2.89

$$\begin{aligned}
\mathbf{W} &= [\delta \underline{u}^e]_b^T R [\delta \underline{u}^e]_b + (P [\delta \underline{u}^e]_b^T) Q P [\delta \underline{u}^e]_b \\
&= [\delta \underline{u}^e]_b^T R [\delta \underline{u}^e]_b + [\delta \underline{u}^e]_b^T P^T Q P [\delta \underline{u}^e]_b \\
&= [\delta \underline{u}^e]_b^T (R + P^T Q P) [\delta \underline{u}^e]_b \\
&= [\rho]_b^T (R + P^T Q P) \Phi [\rho]_b \\
&= [\rho]_b^T [\rho]_b
\end{aligned} \tag{2.91}$$

The final step can be made due to our conjunctionality condition of $\Phi^T (R + P^T Q P) \Phi = I$ from Eq. 2.84.

To extract $[\rho]_b$ then, we preform the Cholesky decomposition on \mathbf{W} . It can be shown that $[\rho]_b$ is then an upper-right triangular $b \times b$ matrix. From $[\rho]_b$, we can reverse Eq. 2.89 to extract our conjugate basis Φ :

$$\Phi = [\delta \underline{u}^e]_b [\rho]_b^{-1} \tag{2.92}$$

One will note that we use $[\rho]_b^{-1}$ and not $[\rho]_b^+$. We must ensure that $[\rho]_b^{-1}$ exists by generating a $[\delta \underline{u}^e]_b$ that is full rank by our choice of inputs.

As well as generating the optimal basis function, LQL also finds the β to associate with said function to minimize our costs. Recall Eq. 2.86, which gives us our optimal β in terms of basis space, system, costs, and error. We do not know P , yet we can extract $P\Phi$ by starting with Eq. 2.88 and Eq. 2.89 so that

$$P\Phi[\rho]_b = [\delta \underline{y}^e]_b \tag{2.93}$$

which can be re-written as

$$P\Phi = [\delta \underline{y}^e]_b [\rho]_b^{-1} \tag{2.94}$$

Define $H_b = P\Phi$.

Given this, we can write

$$\begin{aligned} H_b &= [\delta \underline{y}^e]_b [\rho]_b^{-1} \\ &= \begin{bmatrix} h_1 & h_2 & \dots & h_b \end{bmatrix} \end{aligned} \tag{2.95}$$

which can be transposed and plugged into Eq. 2.86 so that

$$\beta^* = H_b^T Q e_{j-1} \tag{2.96}$$

Recall our assumption of deadbeat operation, so we can re-formulate as

$$\beta^* = H_b^T Q (\underline{y}^* - \underline{d}) \tag{2.97}$$

Batch Example We begin the setup of this example exactly as we did the example shown in 2.2.3. So we set $p = 100$, create a Φ_u of the first 10 chebyshev polynomials, and build $\underline{u}^* = \Phi_u \beta^*$, where

$$\beta^* = \begin{bmatrix} 1 & 0.2 & -0.3 & 4 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}^T \tag{2.57}$$

This then defines our $\underline{y}^* = P \underline{u}^* + \underline{d}$. Once again we can be assured that the inputs we will sweep do capture \underline{u}^* , and this is always a safe assumption. Recall we can always try more inputs, and there will be a definitive number $\ell = r_{ILC}$ that will guarantee a complete capture.

We then define our sequence of inputs as the basis functions chosen. It is not necessary to use chebyshev polynomials – we simply choose them for consistency with our earlier section. Recall we need $b + 1$ pairs of $\underline{u}/\underline{y}$ data, so we will also include the open-loop

response. So

$$\begin{aligned} [\underline{u}^e]_b &= \begin{bmatrix} \underline{u}_0^e & \underline{u}_0^e & \cdots & \underline{u}_b^e \end{bmatrix} \\ &= \begin{bmatrix} 0_{\ell \times 1} & T_0 & \cdots & T_9 \end{bmatrix} \end{aligned} \quad (2.98)$$

$$[\underline{y}^e]_b = \begin{bmatrix} \underline{d} & PT_0 + \underline{d} & \cdots & PT_9 + \underline{d} \end{bmatrix} \quad (2.99)$$

The next step is to apply the δ operator. Eq. 1.52 defined the operator to be the difference between the previous and current trial. It is important to recognize this is largely arbitrary and we could change the order of the inputs to change our δ results without compromising the underlying logic. As such, we create a slightly modified operator, $\underline{\delta}$. This is just to add some notation to the deadbeat assumption we have been employing. We define this as:

$$\underline{\delta}_j x = x_j - x_0 \quad (2.100)$$

Again note this changes none of our logic, it is purely for convenience. Now we can relate each change in input/output to our open loop response. So

$$[\underline{\delta}u^e]_b = \begin{bmatrix} T_0 & T_1 & \cdots & T_9 \end{bmatrix} \quad (2.101)$$

and

$$[\underline{\delta}y^e]_b = \begin{bmatrix} PT_0 & PT_1 & \cdots & PT_9 \end{bmatrix} \quad (2.102)$$

Notice we have removed any dependency on \underline{d} and we can conveniently represent our $\underline{\delta}_j \underline{u}^e$ as just the \underline{u}^e . Eq. 2.88 can be seen to still apply. So we easily generate our batched output as

$$[\underline{\delta}y^e]_b = P[\underline{\delta}u^e]_b \quad (2.103)$$

Next we must define our cost matrices R and Q so we may compute \mathbf{W} . Here we set

$R = 0_{r_{ILC} \times r_{ILC}}$ ¹⁷ and $Q = 100I_{n_{ILC} \times n_{ILC}}$. Thus

$$\mathbf{W} = [\underline{\delta y}^e]_b^T Q [\underline{\delta y}^e]_b \quad (2.104)$$

which we can take the Cholesky decomposition of to get

$$[\rho]_b = \text{chol}(\mathbf{W}) \quad (2.105)$$

In this example, our upper-triangular, $b \times b$ matrix $[\rho]_b$ comes out to be

$$[\rho]_b = \begin{bmatrix} 3.4701 & -1.0633 & \cdots & -0.0665 & 0.0120 \\ 0 & 2.8076 & \cdots & 0.0835 & -0.0029 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 0.1818 & -0.1143 \\ 0 & 0 & \cdots & 0 & 0.1922 \end{bmatrix} \quad (2.106)$$

Our Φ extracted as in Eq. 2.92 is then $\ell \times b = 200 \times 10$ and is partially shown below.

$$\Phi = \begin{bmatrix} 0.2882 & -0.2470 & \cdots & 15.0650 & -15.1293 \\ 0.2882 & -0.2435 & \cdots & 10.5444 & -9.6316 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.2882 & 0.4617 & \cdots & 54.8935 & 69.7895 \\ 0.2882 & 0.4653 & \cdots & 66.6008 & 87.3660 \end{bmatrix} \quad (2.107)$$

We verify this satisfies the condition of Eq. 2.84 and indeed Φ is such that $\mathcal{C} = I_{10 \times 10}$

¹⁷Unlike in our RL examples where small R s can lead to numerical ill-conditioning, the construction of our conjunct condition matrix \mathcal{C} prevents this. Since it relies on P and Q as well, we can simplify our math by removing R from our computations. It would be possible to conversely set $Q = 0$ and have $R \neq 0$.

The associated β^* can be calculated as

$$\beta^* = \begin{bmatrix} 8.0301 \\ -13.0317 \\ -11.3577 \\ 5.2253 \\ -0.4801 \\ 0.5204 \\ 0.1702 \\ -0.2106 \\ 0.0000 \\ 0.0000 \end{bmatrix} \quad (2.108)$$

One will note that after 8 basis functions, the coefficients drop to 0. This is because once we have tried all the inputs that make up \underline{u}^* , we can perfectly produce the true signal without any additional information. If we apply $\underline{u} = \Phi\beta^*$, we see we capture our goal output perfectly (Figure 2.39)

Batch LQL Output - 10 Conjugate Basis Functions

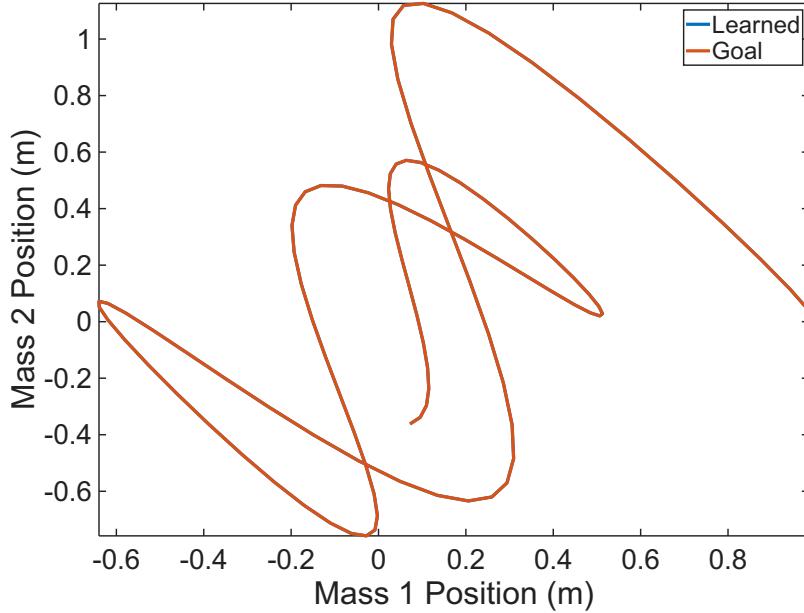


Figure 2.39: Output Generated from Batch Generated Basis Space Φ with weights β^* overlaid on the Goal Output. As the \underline{u}^* was generated within a space spanned by all of the \underline{u}^e 's that constructed the basis space, the perfect output is captured.

The Iterative Approach

While this batch approach is convenient for notation, it is computational sloppy and inefficient. The whole logic of conjugate basis functions is that we should only need to compute one conjugate basis function at a time, and it would be preferable to do so in real time. We will now show a process to derive just ϕ_{b+1} given b pre-existing basis functions.

To generate a new ϕ we need to introduce a new input to $[\underline{u}^e]_b$, ensuring full rank is maintained. This new input \underline{u}_{b+1}^e can be added to our batch so that

$$[\underline{u}^e]_{b+1} = [[\underline{u}^e]_b, \underline{u}_{b+1}^e] \quad (2.109)$$

\underline{u}_{b+1}^e logically produces a \underline{y}_{b+1}^e which can similarly be added to the output batch.

$$[\underline{y}^e]_{b+1} = [[\underline{y}^e]_b, \underline{y}_{b+1}^e] \quad (2.110)$$

By applying our δ operator¹⁸ and returning to Eq. 2.91, we can write

$$[\rho]_{b+1}^T [\rho]_{b+1} = [\delta \underline{u}^e]_{b+1}^T R [\delta \underline{u}^e]_{b+1} + [\delta \underline{y}^e]_{b+1}^T Q [\delta \underline{y}^e]_{b+1} \quad (2.111)$$

For our $b + 1$ th trial, we want to get ρ_{b+1} , or the last column of $[\rho]_{b+1}$. We can then write Eq. 2.111 in terms of only its last columns

$$[\rho]_{b+1}^T \rho_{b+1} = [\delta \underline{u}^e]_{b+1}^T R \delta \underline{u}_{b+1}^e + [\delta \underline{y}^e]_{b+1}^T Q \delta \underline{y}_{b+1}^e \quad (2.112)$$

We can expand this as

$$\begin{bmatrix} \rho_1^T \\ \vdots \\ \rho_{b+1}^T \end{bmatrix} \rho_{b+1} = \begin{bmatrix} (\delta_1 \underline{u}^e)^T R \delta_{b+1} \underline{u}^e + (\delta_1 \underline{y}^e)^T Q \delta_{b+1} \underline{y}^e \\ \vdots \\ (\delta_{b+1} \underline{u}^e)^T R \delta_{b+1} \underline{u}^e + (\delta_{b+1} \underline{y}^e)^T Q \delta_{b+1} \underline{y}^e \end{bmatrix} \quad (2.113)$$

Recall that that $[\rho]_{b+1}$ is upper-right triangular. Given this, we can extract the first element of ρ_{b+1} as

$$\rho_{b+1}(1) = \frac{1}{\rho_1(1)} \left[(\delta_1 \underline{u}^e)^T R \delta_{b+1} \underline{u}^e + (\delta_1 \underline{y}^e)^T Q \delta_{b+1} \underline{y}^e \right] \quad (2.114)$$

The 2nd through b th element can be similarly calculate, but must account for the already computed elements of ρ , such that

$$\rho_{b+1}(i) = \frac{1}{\rho_i(i)} \left[(\delta_i \underline{u}^e)^T R \delta_{b+1} \underline{u}^e + (\delta_i \underline{y}^e)^T Q \delta_{b+1} \underline{y}^e - \sum_{j=1}^{i-1} \rho_i(j) \rho_{b+1}(j) \right] \quad (2.115)$$

$$i = 2, 3, \dots, b$$

and the final element $\rho_{b+1}(b+1)$ can be more easily computed by first defining γ to capture

¹⁸Note we are not using the $\underline{\delta}$ shown in the Batch example for the derivation, even though we could

all the previous components

$$\begin{aligned}\gamma &= \sum_{j=1}^b (\rho_{b+1}(j))^2 \\ &= \begin{bmatrix} \rho_{b+1}(1) & \cdots & \rho_{b+1}(b) \end{bmatrix} \begin{bmatrix} \rho_{b+1}(1) \\ \vdots \\ \rho_{b+1}(b) \end{bmatrix}\end{aligned}\tag{2.116}$$

so we can write

$$\rho_{b+1}(b+1) = \sqrt{(\delta_{b+1}\underline{u}^e)^T R \delta_{b+1}\underline{u}^e + (\delta_{b+1}\underline{y}^e)^T Q \delta_{b+1}\underline{y}^e - \gamma}\tag{2.117}$$

We can then derive our new conjugate basis function ϕ_{b+1}

$$\phi_{b+1} = \frac{1}{\rho_{b+1}(b+1)} \left[\delta_{b+1}\underline{u}^e - \sum_{i=1}^b \phi_i \rho_{b+1}(i) \right]\tag{2.118}$$

and the associated β^* from

$$h_{b+1} = \frac{1}{\rho_{b+1}(b+1)} \left[\delta_{b+1}\underline{y}^e - \sum_{i=1}^b h_i \rho_{b+1}(i) \right]\tag{2.119}$$

$$\beta^*(b+1) = h_{b+1}^T Q (\underline{y}^* - \underline{d})\tag{2.120}$$

With this approach, we can efficiently increase our number of basis functions as we go, while maintaining the conjunct properties of optimality on all previous inputs. The iterative algorithm follows a simple sequence for a p step process

1. Choose your Q and R
2. Define your input sequences \underline{u}_i^e such that they span the whole input space
3. Perform two initial experiments \underline{u}_0^e and \underline{u}_1^e to generate \underline{y}_0^e and \underline{y}_1^e ¹⁹

¹⁹Typically \underline{u}_0^e is no input such that $\underline{y}_0^e = \underline{d}$

4. Extract ϕ_1 from Eq. 2.91 and 2.92
 - Compute β^* for ϕ_1 (Eq. 2.95 and Eq. 2.96) and apply it to the system
5. Perform a third experiment with \underline{u}_2^e to generate \underline{y}_2^e . Use Eqs. 2.114 and 2.117 to extract ρ_2 , then plug into Eq. 2.118 to compute ϕ_2
 - Compute β^* for the given ϕ_2 (Eq. 2.119 and Eq. 2.120) and apply it to the system
6. Repeat with \underline{u}_i^e to extract ϕ_i as needed, using Eqs. 2.114–2.118
 - Compute β^* for the given ϕ_i (Eq. 2.119 and Eq. 2.120) and apply it to the system
7. Stop once the cost function is minimized, or error is acceptable

Iterative Example We start off by repeating the setup of \underline{u}^* and \underline{y}^* from the Batch Example 2.3.3.

To begin the iterative approach, we must first run two episode to such that we may compute our first $\underline{\delta}_1$ ²⁰. As before, we let our 0th episode contain the open loop data, and apply T_0 as \underline{u}_1^e to create \underline{y}_1^e .

Since we are using the $\underline{\delta}$ operator, as in the batch example, $\underline{\delta}_1 \underline{u}^e = \underline{u}_1^e$, and $\underline{\delta}_1 \underline{y}^e = \underline{y}_1^e - \underline{d}$. This applies across all episodes, and allows for neater notation.

Working through the calculation of our first basis function, we find

$$\mathbf{W} = 12.0415$$

$$\rho_1 = 3.4701$$

$$\begin{aligned}\phi_1 &= \begin{bmatrix} 0.2882 & 0.2882 & \cdots & 0.2882 & 0.2882 \end{bmatrix}^T \\ h_1 &= \begin{bmatrix} 0.0000 & 0.0000 & \cdots & 0.0059 & 0.0073 \end{bmatrix}^T\end{aligned}\tag{2.121}$$

$$\beta_1 = 8.0301$$

²⁰We are once again using the $\underline{\delta}_j x = x_j - x_0$ for convenience

the most important result being that our β_1 matches that of the β_1 found in the batch approach.

Applying then just $\phi_1\beta_1$ to our system we see from Figure 2.40, it clearly is not sufficient to produce the output we desire. So we proceed with our second basis function.

We excite the system with $\underline{u}_2^e = T_1$ to generate \underline{y}_2^e . We must solve for ρ_2 in multiple steps now, but we find

$$\rho_2(1) = -1.0633$$

$$\gamma = 1.1305$$

$$\rho_2(2) = 2.8076$$

$$\phi_2 = \begin{bmatrix} -0.2470 & -0.2435 & \cdots & 0.4617 & 0.4653 \end{bmatrix}^T \quad (2.122)$$

$$h_2 = \begin{bmatrix} 0.0000 & 0.0000 & \cdots & 0.0077 & 0.0098 \end{bmatrix}^T$$

$$\beta_2 = -13.0317$$

once again matching that exactly of our batched approach.

Just for completeness, we will show one more example where we excite with $\underline{u}_3^e = T_2$.

Following the steps outlined, we get

$$\rho_3(1) = -1.3392$$

$$\rho_3(2) = -1.7959$$

$$\gamma = 5.0187$$

$$\rho_3(3) = 2.8832 \quad (2.123)$$

$$\phi_3 = \begin{bmatrix} 0.3268 & 0.3152 & \cdots & 0.7544 & 0.7705 \end{bmatrix}^T$$

$$h_3 = \begin{bmatrix} 0.0000 & 0.0000 & \cdots & 0.0179 & 0.0222 \end{bmatrix}^T$$

$$\beta_2 = -11.3577$$

With explicit examples for all the three major steps, we now carry out the rest of the trials.²¹ After 10 trials (plus the one open-loop response), we find

$$\Phi = \begin{bmatrix} 0.2882 & -0.2470 & \cdots & 15.0650 & -15.1293 \\ 0.2882 & -0.2435 & \cdots & 10.5444 & -9.6316 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.2882 & 0.4617 & \cdots & 54.8935 & 69.7895 \\ 0.2882 & 0.4653 & \cdots & 66.6008 & 87.3660 \end{bmatrix} \quad (2.124)$$

$$\beta^* = \begin{bmatrix} 8.0301 \\ -13.0317 \\ -11.3577 \\ 5.2253 \\ -0.4801 \\ 0.5204 \\ 0.1702 \\ -0.2106 \\ 0.0000 \\ 0.0000 \end{bmatrix}^T \quad (2.125)$$

which we can see completely matches our batched approach.

With each additional basis function, the output we are able to capture gets closer to that of our goal y^* . Figure 2.40 shows that with just one basis function, we do not capture much. By Figure 2.41 where we have three basis functions, things are looking better. And as we saw in the batched example, once we have tried enough inputs to span that space that contains u^* , we can perfectly capture y^* . Except now through the iterative approach, we can stop once we have enough, as in Figure 2.42 where we stop after 8 basis functions are

²¹See function `generate_iterative_conjugate` in Code Appendix D.9 to provide a more structured understanding.

generated.

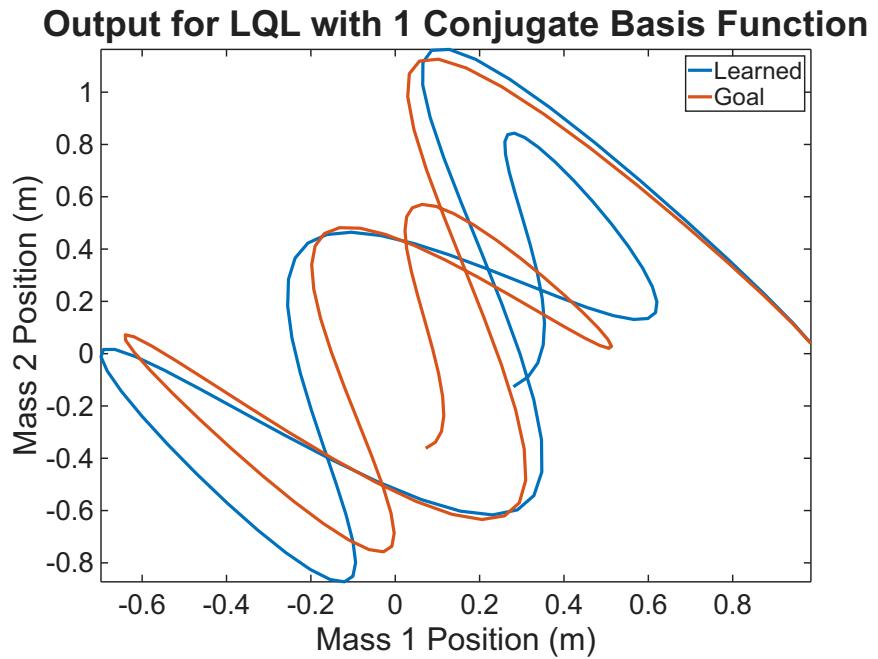


Figure 2.40: Output Generated from Iteratively Generated Basis Space Φ with weights β^* when $\eta = 1$ overlaid on the Goal Output

We have now shown a methodology to efficiently calculate conjugate basis functions on the fly, without compromising the optimality of pre-existing inputs.

Output for LQL with 3 Conjugate Basis Functions

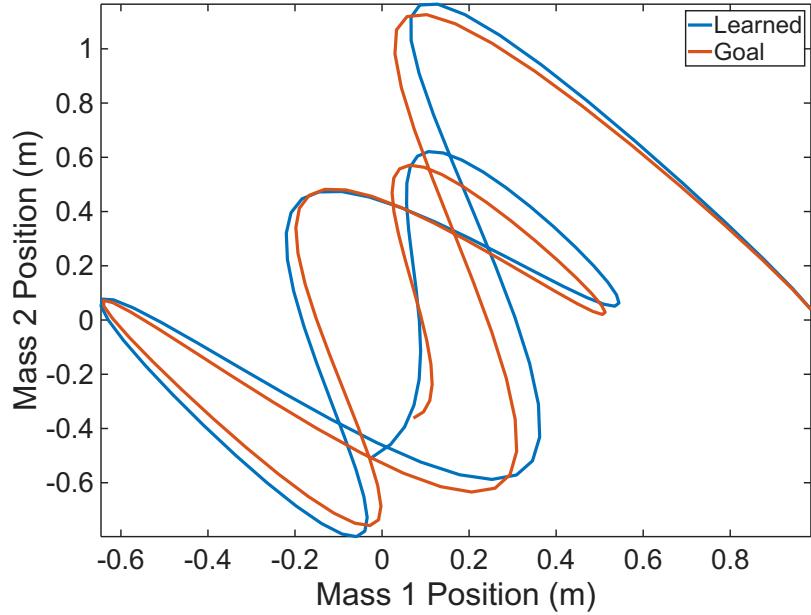


Figure 2.41: Output Generated from Iteratively Generated Basis Space Φ with weights β^* when $\eta = 3$ overlaid on the Goal Output

Output for LQL with 8 Conjugate Basis Functions

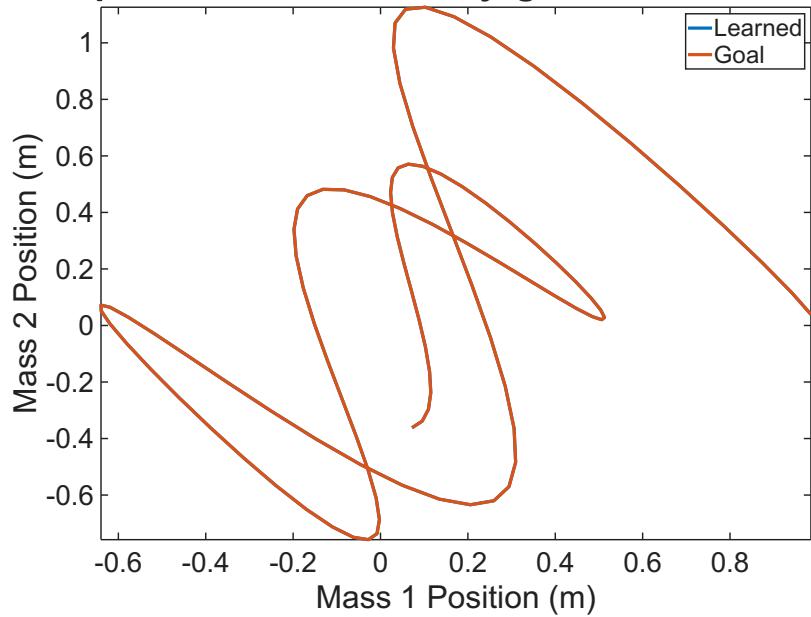


Figure 2.42: Output Generated from Iteratively Generated Basis Space Φ with weights β^* when $\eta = 8$ overlaid on the Goal Output. It is demonstrated that we do not explore the full space, as done in the batch example, and can instead learn one basis at a time. By learning iteratively, we can stop once we have determined our space to be good enough for our desired output.

Properties

We see that both the batch and the iterative approach yield the same conjugate basis space Φ and optimal weights β . By definition, these basis functions are created such that their application as inputs are optimal, and independent of any other basis functions in the output they produce on a system. The addition of one new basis function does not impact the learned portion of any prior one on the system.

Chapter 3

Results

3.1 RL on ILC in a Conjugate Basis Space

We have explored now a way to safely extract a number of basis functions for a problem, and a way to grow that number without impact previously learned optimal weightings of them. We now wish to see if these learned basis functions can be applied back to describe a space that we can apply RL in, and the properties we may exploit.

3.1.1 Tying it all together

Recall the original problem of in the Iterative Control Problem, the dimensions on states and inputs scale by the number of steps p . This translates exponentially to the number of trials needed to update a controller in the RL framework, and given that p is typically large this can result in millions of failed trials before learning is even attempted.

We showed that with basis functions, the dimensions of a problem can be infinitely reduced. We discovered that $\underline{u}^* \in \Phi$ and $\eta_y \geq \eta_u$ were the only requirements in order to capture \underline{y}^* . In a best-case scenario, one would be able to infinitely reduce their input and output to a single dimension, meaning RL controllers could be updated in as few as 4 trials. This ideal scenario may seem pointless at first, as we can only do so if $\Phi = \underline{u}^*$. However, in the RL framework we introduce exploration noise, so it would be possible to have a $\Phi \approx \underline{u}^*$ with the same learning results.

To can determine this approximate \underline{u}^* however, we need to build out a starting Φ . Conjugate Basis Functions provide us with a way to add a single basis function at a time, with a guarantee that it does not interfere with any previously learn optimal inputs.

Before we can make the final leap to learning a controller, we first wish to verify the conjunct properties stretch into the controller land. Just as each $\Phi_b \subset \Phi^{b+1}$ (Eq. 2.71), we hope to find that each additional ϕ^{e+1} does not impact the previous learned controllers. If F^{Φ_b} is the controller found for the entire $\ell \times b$ basis space Φ , then F_{ϕ_e} is the controller

learned for the ϕ from episode e , such that

$$F^{\Phi_b} = \begin{bmatrix} F_{\phi_1} \\ F_{\phi_2} \\ \vdots \\ F_{\phi_b} \end{bmatrix} \quad (3.1)$$

and the addition of one additional basis function would have it such that

$$F^{\Phi_{b+1}} = \begin{bmatrix} F^{\Phi_b} \\ F_{\phi_{b+1}} \end{bmatrix} \quad (3.2)$$

3.1.2 The Conjugate LQR Controller

We know the conditions to ensure a found controller captures \underline{y}^* in the basis space, so that is not what we will be testing here. We are now interested the scenario where we do not capture \underline{u}^* in our Φ , but we still find a controller in this space. It should be that if we find F_{ϕ_1} and F_{ϕ_2} separately, they would have the exact same parameters as controller F^{Φ_2} ¹.

Example – LQR in The Conjugate Space

We know our RL techniques are capable of extracting the discount LQR controller, so we will cut through the noise in our example and simply use *discounted_LQR*.

We return to our earlier \underline{y}^* of a circle. We set $p = 10$, and use Eq. 1.62 to set the goal. Our first step is to generate our conjugate basis functions. We use the batch approach in this example. With the parameters

$$Q = 100I_{20 \times 20} \quad R = 0_{20 \times 20} \quad (3.3)$$

¹So long as they all use the same basis space on the output.

we generate 20 conjugate basis functions

$$\Phi = \begin{bmatrix} 6.2 & -31.2 & \cdots & 178.5 & 62.1 \\ 6.2 & -25.8 & \cdots & -54.8 & 26.2 \\ \vdots & \vdots & & \vdots & \vdots \\ 6.2 & 67.1 & \cdots & -268.2 & -2,575.9 \\ 6.2 & 72.6 & \cdots & 1,375 & -11.0 \end{bmatrix} \quad (3.4)$$

We then move on to our LQR solution. Here we set a new Q , R , and a γ .

$$Q = 100I_{20 \times 20} \quad R = 10I_{20 \times 20} \quad \gamma = 0.8 \quad (3.5)$$

With our parameters now set, we now learn our LQR controllers.

We begin with ϕ_1 . Recall that for these test to be logical, we must maintain the same Φ on the outputs, so our output will still have 20 states – n_{ILC} . For the ILC problem in a basis space, $e_{\alpha_{j+1}} = Ie_{\alpha_j} - H\delta_{j+1}\underline{u}$, so for our LQR controller's A and B must be modified. Using Eq. 2.44, we can write

$$A_{LQR} = I_{20 \times 20} \quad B_{ILC} = -\Phi^+ P \phi_1 \quad (3.6)$$

It is important to remember to grab the right component of R now. We have equal weightings for all our inputs, so this is easy to mess up and not realize. For ϕ_1 , we grab $R_{1,1}$ ².

Solving for the $1 \times 20 F_{LQR}$ under these parameters, we find

$$F_{\phi_1} = \begin{bmatrix} -0.0554 & -0.0183 & \cdots & -0.0001 & -0.0005 \end{bmatrix} \quad (3.7)$$

We repeat the same process for ϕ_2 , defining B_{ILC} with respect to ϕ_2 and updating our

²or $\mathbb{R}(1, 1)$ in Matlab syntax

R for completeness. Computing the associated F_{LQR} we find

$$F_{\phi_2} = \begin{bmatrix} 0.0925 & 0.0071 & \cdots & -0.0001 & 0.0004 \end{bmatrix} \quad (3.8)$$

These two results by themselves are not that important. It is when we learn a new controller with $\Phi = \begin{bmatrix} \phi_1 & \phi_2 \end{bmatrix}$ that we see the conjugate property emerge. For the found 2×20 controller is

$$F^{\Phi_2} = \begin{bmatrix} -0.0554 & -0.0183 & \cdots & -0.0001 & -0.0005 \\ 0.0924 & 0.0071 & \cdots & -0.0001 & 0.0004 \end{bmatrix} \quad (3.9)$$

We will note that the controllers are ‘almost’ exactly identical. F_{ϕ_2} ’s first term differs from that of F^{Φ_2} ’s first term of the second row by 0.0001. Given the rest of the controller matches exactly and the rest of the theory seems to be consistent, we attribute this to computational rounding³.

For completeness, we will run an ILC Trial as we did when exploring the requirements on basis functions. For our $p = 10$, $n_{ILC} = r_{ILC} = 20$ system trying to draw a circle, we generate 20 conjugate basis functions. Once again employing our $\mathcal{L}_\beta = 0.5H^+$ controller, we see that perfect control is possible just as it was before (Figures 3.1 and 3.2). So we are guaranteed to fully learn the necessary output / controller in at most ℓ basis attempts.

³On the topic of computational limits, we once again are limited on how small we can set are R matrix. This will lead to some visually slow processes in our demonstration, but it is all in the effort of proving the theory still works. Methods to ensure proper numerical condition can be further explored in the future, as mentioned in the Future Work section

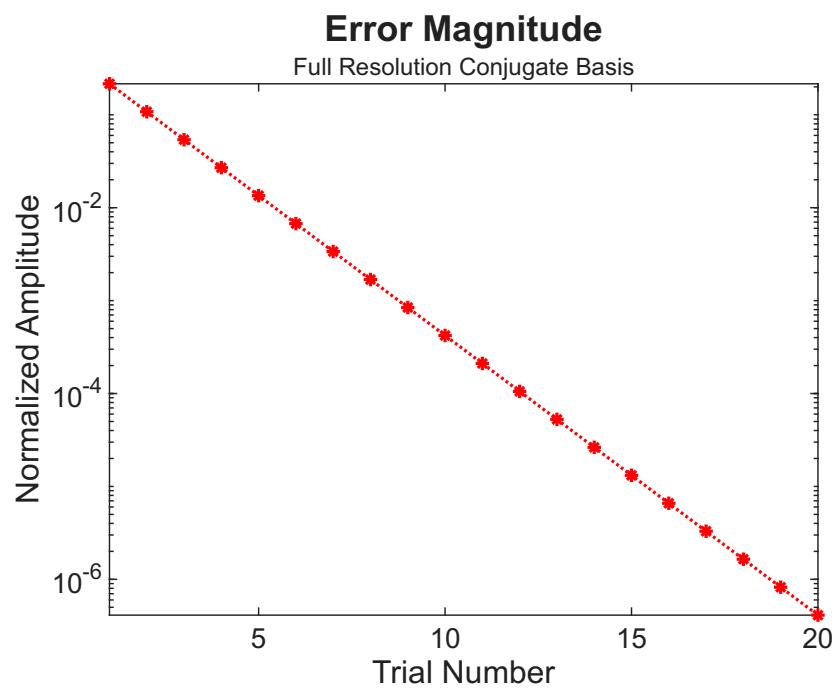


Figure 3.1: Error Progression through ILC Trials with a Perfect Knowledge Controller in a Full Conjugate Basis Space

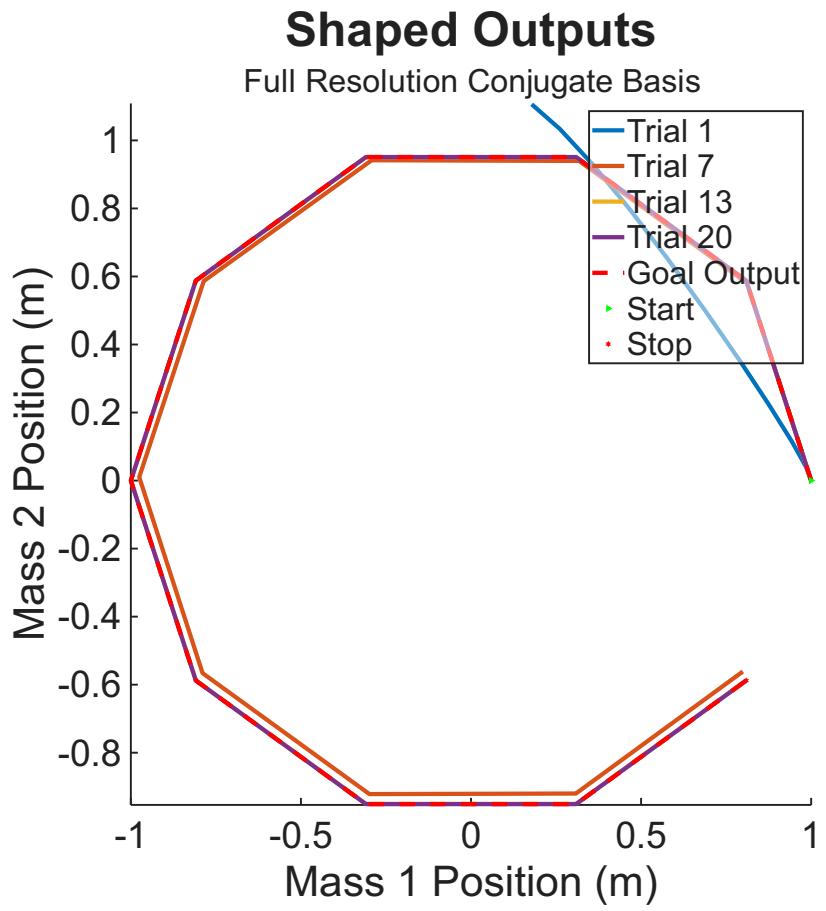


Figure 3.2: Shaped Output Progression through ILC Trials with a Perfect Knowledge Controller in a Full Conjugate Basis Space

3.1.3 Dynamic Conjugate Basis Space

Now we have shown that one controller can be learned at a time, in any order, with any number at a time, regardless of the representative space chosen. This will allow us to revisit our idea of a dynamic input basis space explored earlier in Section 2.2.4.

For our dynamic basis space, there are a few different scenarios to consider. Recall our starting notation in section 2.2 where our input basis space had notation Φ_u and our output had notation Φ_y . We will re-introduce this now to consider our options.

Fixed Resolution: $\Phi_y = I$

In the first scenario, we define a fixed resolution $\Phi_y = I_{\eta_y \times \eta_y}$ and a $\Phi_u = \phi_i$, where ϕ_i is the most recently learned conjugate basis function. We use a fixed, full-resolution Φ_y such that the controller we learn in this input space ϕ_i can be stacked with other controllers for spaces ϕ_j (where $i \neq j$) to form a larger controller, as we did with input decoupling. Here, we would run two initial trials to generate our first ϕ_1 , then begin learning the controller in that space. While learning, we would produce $\underline{u}s$ that could be used to generate a ϕ_2 and have that ready to go when we are done learning with ϕ_1 . This process could be repeated, and would at worst be done in a finite number of r_{ILC} ϕ s. However, one would need to properly choose η_y to ensure that the η_u used to capture \underline{u}^* obeyed our condition in Eq. 2.65 of $\eta_y \geq \eta_u$. At worst, this means $\eta_y = r_{ILC}$ ⁴.

Example – Full Resolution Identity on Output We are back to our goal shown in Figure 3.2. To handle the worst case scenario, set $\eta_y = 20$. Our conjugate basis parameters (generated iteratively, but will still work out to match Eq. 3.4) are

$$Q = 100I_{20 \times 20} \quad R = 0_{20 \times 20} \quad (3.10)$$

Our RL parameters will be similar. Recall the earlier computational issues with $R = 0$

⁴Remember we only need to fully capture \underline{u}^* , so if \underline{y}^* is not fully captured, that is ok.

and the need for proper exploration noise, shown in our derivation of F_{LQR} in Eq. 2.16.

$$\begin{aligned} Q &= 100I_{\eta_y \times \eta_y} & R &= 1 \times 10^{-6}I_{r_{ILC} \times r_{ILC}} \\ \gamma &= 0.8 & v(k) &\in [-10, 10] \end{aligned} \quad (3.11)$$

We generate our first ϕ iteratively, and learn the associated controller F_{LQR} in that basis space. Denoting that as F_{ϕ_1} , we find

$$F_{\phi_1} = \begin{bmatrix} 0.0309 & 0.0616 & \dots & 3.2006 & 5.2332 \end{bmatrix} \quad (3.12)$$

We repeat this process of generating conjugate basis functions (using Chebyshev Polynomials as our \underline{u}^e s for consistency with earlier examples, but we could use any input that was independent of previous \underline{u}^e s). We then take our ϕ_i and learn the new F_{ϕ_i} . We do this learning process without applying the previously learned controller to highlight it is possible. It would be possible to apply learned controllers as we learn, just as in input decoupling.

By assembling all the F_{ϕ_i} s in a stack, we find

$$\begin{bmatrix} F_{\phi_1} \\ F_{\phi_2} \\ \vdots \\ F_{\phi_{19}} \\ F_{\phi_{20}} \end{bmatrix} = \begin{bmatrix} 0.0309 & 0.0616 & \dots & 3.2006 & 5.2332 \\ -0.1558 & -0.2569 & \dots & 0.4908 & 5.7784 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.8842 & -0.5381 & \dots & -0.0071 & 0.0002 \\ 0.3088 & 0.2620 & \dots & -0.0033 & 0.0001 \end{bmatrix} \quad (3.13)$$

We then take $\Phi = [\phi_1 \ \phi_2 \ \dots \ \phi_{19} \ \phi_{20}]$ and compute our F_{LQR}^γ . Similar to in Eq. 3.6, we compute the Basis Space ILC Matrices as

$$\begin{aligned} A_{ILC} &= I_{20 \times 20} \\ B_{ILC} &= -\Phi^+ P \Phi \end{aligned} \quad (3.14)$$

When we solve for the F_{LQR}^γ with the process shown in Eq. 1.42 and the parameters from Eq. 3.11, we get

$$F_{LQR}^\gamma = \begin{bmatrix} 0.0309 & 0.0616 & \dots & 3.2006 & 5.2332 \\ -0.1558 & -0.2569 & \dots & 0.4908 & 5.7784 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.8842 & -0.5381 & \dots & -0.0071 & 0.0002 \\ 0.3088 & 0.2620 & \dots & -0.0033 & 0.0001 \end{bmatrix} \quad (3.15)$$

which perfectly matches the stacking of our controllers learned one at a time.

As mentioned above, we do not apply the previously learned controllers through the learning process. This is why our error progression in Figure 3.3 appears so poor. However after a single trial of the complete controller, we have near zero error, as shown in Figure 3.4.

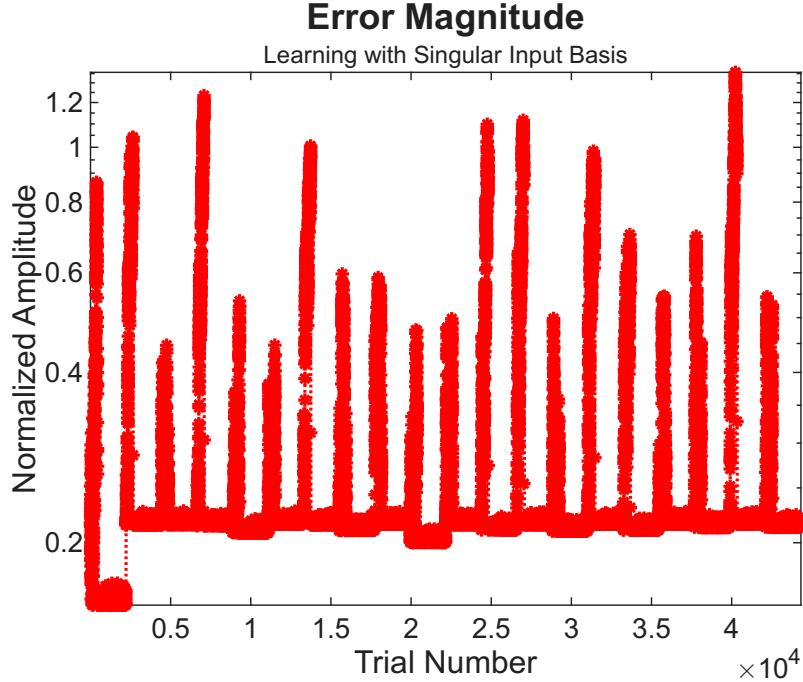


Figure 3.3: Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs. For a fixed output basis $\Phi_y = I$, 20 ϕ s are tried to capture the input, and the associated F_{LQR}^γ is learned (but not applied).

It can be easy to confuse this approach with input decoupling. While they are very sim-

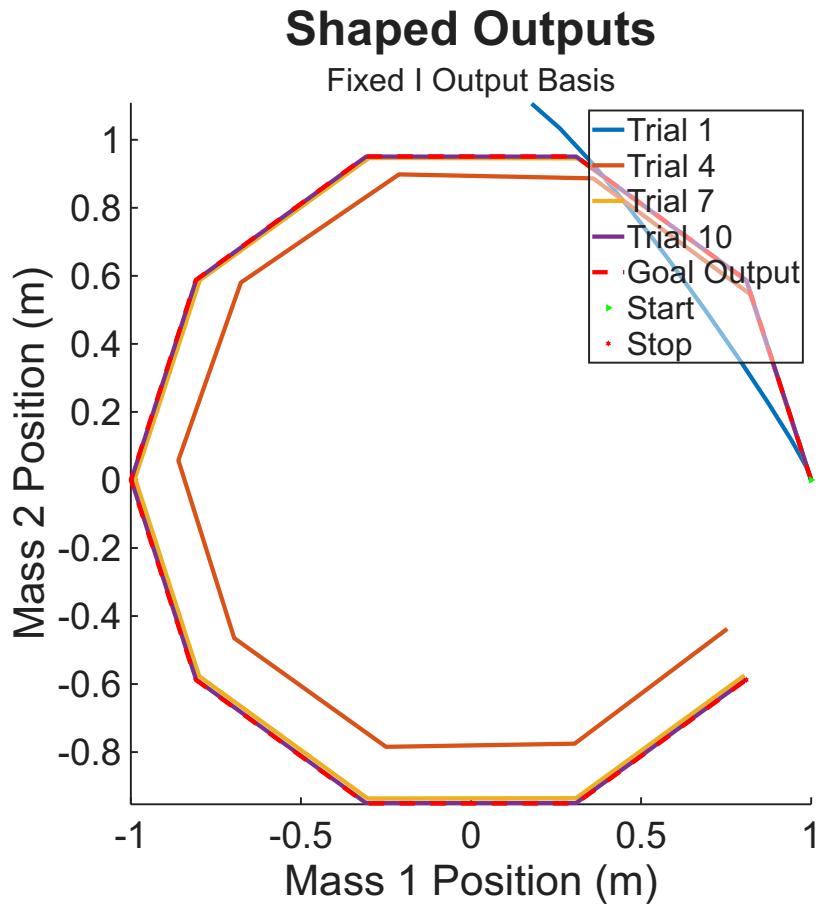


Figure 3.4: Shaped Output Progression through Application of a Final Controller that was Learned One ϕ at a time. The resultant controller is then built out of the individually learned F_{LQR}^γ s for each ϕ

ilar, this approach has the benefit of where it will not need to loop back through to update certain controllers after determining subsequent ones. Once a controller is determined for its space, it is known to be optimal and independent of any other controllers.

Fixed Resolution $\Phi_y = \Phi^b$

Another option is to use a fixed resolution $\Phi_y = \Phi^b$, where we have b conjugate basis functions already generated. The issue that arises in this scenario is one very similar to our earlier attempts of RL on ILC, where we had to introduce more exploration ($v(k)$) when our R was small. While the math and logic remains sound with our conjugate basis, behind the scenes were are numerically poor.

This can best be seen by inspecting ϕ_1 in our examples, in which every component is 6.1844. Compare this to when we used I and only one component had a value of 1. It is easy to see that the basis coefficients that results from these different basis functions (see Eq. 2.40) will differ drastically in magnitude. To rectify this discrepancy in magnitude, we are best off scaling our Φ_y and Φ_u such that $|\Phi_y| > |\Phi_u|$. A scaling which increases the magnitude of Φ_y results in smaller α s, meaning our \mathcal{L}_β will have larger values. This helps for numerical conditioning when solving.

We must be careful when scaling conjugate basis functions, so that they continue to obey our conjunctionality condition from Eqs. 2.84. We can safely normalize individual basis functions and still have our conjunct condition of diagonality, but for simplicity we scale the entire Φ by some x . Then our conjunctionality matrix from Eq. 2.77 just equals $x^2 I$.

So where to deal with small R s we had to introduce more exploration noise, for large Φ s we must scale Φ_y up relative to Φ_u .

Example – Full Resolution Conjugate Basis on Output In this example, we start with the entirety of our basis functions pre-generated (computed through $p + 1$ trials). Keeping

the same parameters and goal from above, our Φ^b can be seen in Eq. 3.4.

We then set our LQR parameters as

$$\begin{aligned} Q &= 100I_{20 \times 20} & R &= I_{20 \times 20} \\ \gamma &= 0.8 & v(k) &\in [-1, 1] \end{aligned} \tag{3.16}$$

As we are working with a relatively large R , we can use this smaller exploration term. If we were to reduce R as we have in the past, the same steps must be made to ensure rich data by increasing the range from which $v(k)$ draws from.

If we were to run this example as is, setting $\Phi_u = \Phi_y = \Phi$, where Φ is our conjugate basis functions, we see that our LQR Controller constructed from perfect knowledge

$$F_{LQR}^\gamma = \begin{bmatrix} 0.5314 & 0.1829 & \dots & 0.0011 & 0.0051 \\ -0.8870 & -0.0719 & \dots & 0.0009 & -0.0032 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.0022 & -0.0033 & \dots & -0.0123 & -0.0123 \\ 0.0192 & -0.0093 & \dots & 0.0128 & -0.0247 \end{bmatrix} \tag{3.17}$$

does not match that of the one produced from Policy Iteration

$$F_{policy} = \begin{bmatrix} 0.5520 & 0.1819 & \dots & 0.0011 & 0.0054 \\ -0.9157 & -0.0700 & \dots & 0.0009 & -0.0035 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.0022 & -0.0033 & \dots & -0.0123 & -0.0123 \\ 0.0202 & -0.0093 & \dots & 0.0128 & -0.0247 \end{bmatrix} \tag{3.18}$$

Even increasing our $v(k)$ s range by a factor of a million does not resolve this discrepancy. If we set $R = 100I_{20 \times 20}$ we can get the controllers to match, but we have previously seen that this form of controller is often undesirable.

The alternative is to scale our basis function. Just as the Q and R values are arbitrary and only mean something in relation to each other, the magnitudes of Φ_u and Φ_y only matter in relativity. In this case

$$\Phi_y = 10\Phi \quad \Phi_u = \Phi \quad (3.19)$$

Holding all the other parameters the same, we once again find that our F_{LQR}^γ controller can be found by learning one controller F_ϕ at a time and then stacking them together, such that $F_{LQR}^\gamma = F_{policy} = F$. In this example

$$F = \begin{bmatrix} 0.0554 & 0.0183 & \cdots & 0.0001 & 0.0005 \\ -0.0925 & -0.0071 & \cdots & 0.0001 & -0.0004 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.0002 & -0.0003 & \cdots & -0.0012 & -0.0012 \\ 0.0020 & -0.0009 & \cdots & 0.0013 & -0.0025 \end{bmatrix} \quad (3.20)$$

We can see this controller works to reduce error in Figure 3.5, but predictably takes a extreme amount of trials due to the higher R .

To confirm that this descent continues, we check the poles of $(I_{n_{ILC} \times n_{ILC}} - \Phi_y^+ P \Phi_u) F$ and see that they are within the unit circle - though just barely.

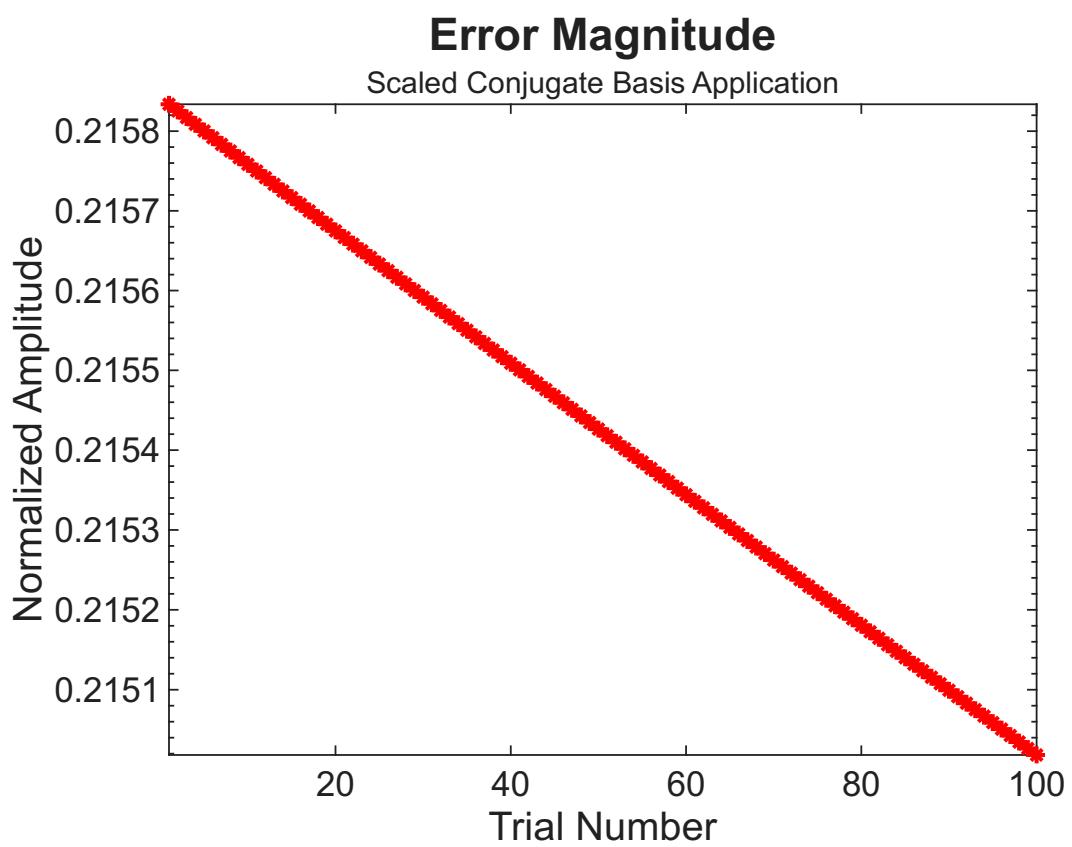


Figure 3.5: Error Progression through Policy Learning ILC Trials with Scaled Conjugate Output Basis

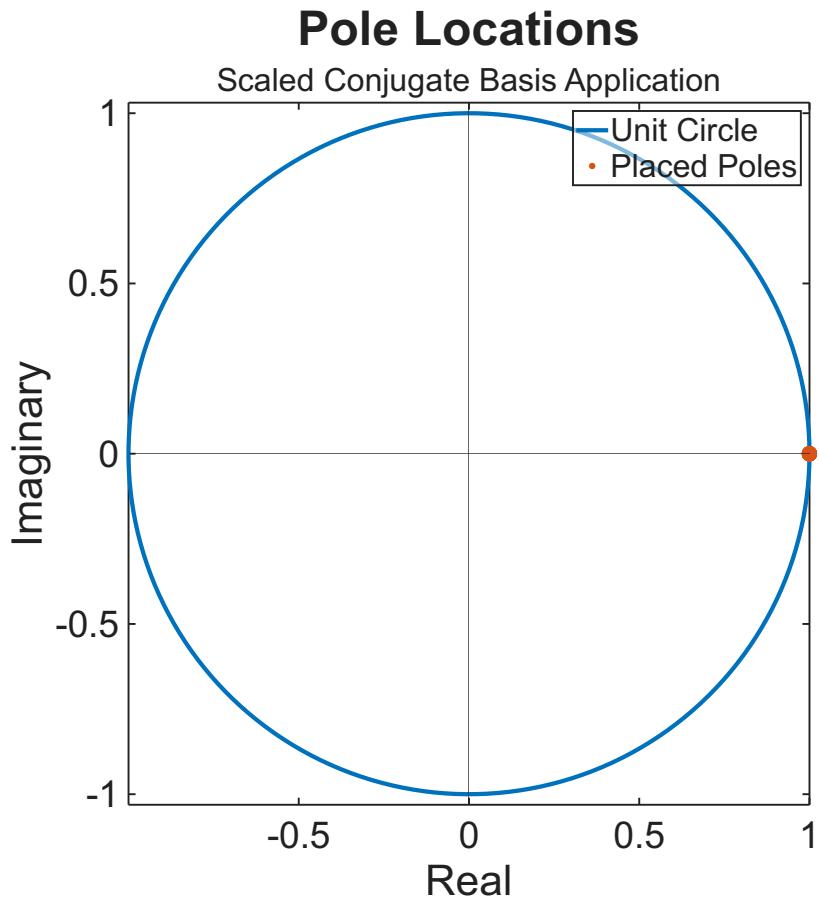


Figure 3.6: Pole locations of the system under a controller found with a fixed $\Phi_y = 10\Phi$ and by rotating through $\Phi_u = \Phi$. Of the 20 poles, 18 are exactly 1 while 2 are 0.9991 and 0.9998. This is due to the required R values.

Fixed Resolution $\Phi_y = \underline{y}^*$

If we have shown we can learn one controller at a time, then it should stand to reason that we could only use one output basis function at a time. Logically, we would choose this to be \underline{y}^* , though as previously stated it does not matter – we will see that some additional steps must be taken.

Example – $\Phi_y = \underline{y}^*$ The first step is to establish our basis functions. Φ_u will be our conjugate basis functions Φ , and $\Phi_y = 100\underline{y}^*$. We have previously shown that there is some trade-off between the magnitudes of the basis functions, that can lead to ill-conditioning. For this example, it was found that scaling \underline{y}^* by 100 (bringing the norm to 316) was sufficient to prevent issues.

We then set our LQR parameters as⁵

$$Q = 1000 \quad R = 10I_{r_{ILC} \times r_{ILC}} \quad (3.21)$$

$$\gamma = 0.8 \quad v(k) \in [-1, 1]$$

We then iteratively learn F_{ϕ_i} s. For each ϕ_i , our Q stays fixed at 100 since we only have one ‘output’ α , but our R rolls along the diagonals of R . In this example, that they are all the same value, but it is an important consideration to be aware of. As before, we stack each learned F_{ϕ_i} and find after all 20 possible controllers

$$F = \begin{bmatrix} -1.2603 \times 10^{-2} \\ -3.6281 \times 10^{-3} \\ \vdots \\ 1.8588 \times 10^{-4} \\ -9.6489 \times 10^{-3} \end{bmatrix} \quad (3.22)$$

⁵Similar to the scaling of Φ_y , there is potential for future work here. The produced result of this approach of $Q/R = 100$ is not the same when we scale both items down by 10. Why the ratio no longer seems to be the absolute determining factor for LQR outcomes is worth nailing down.

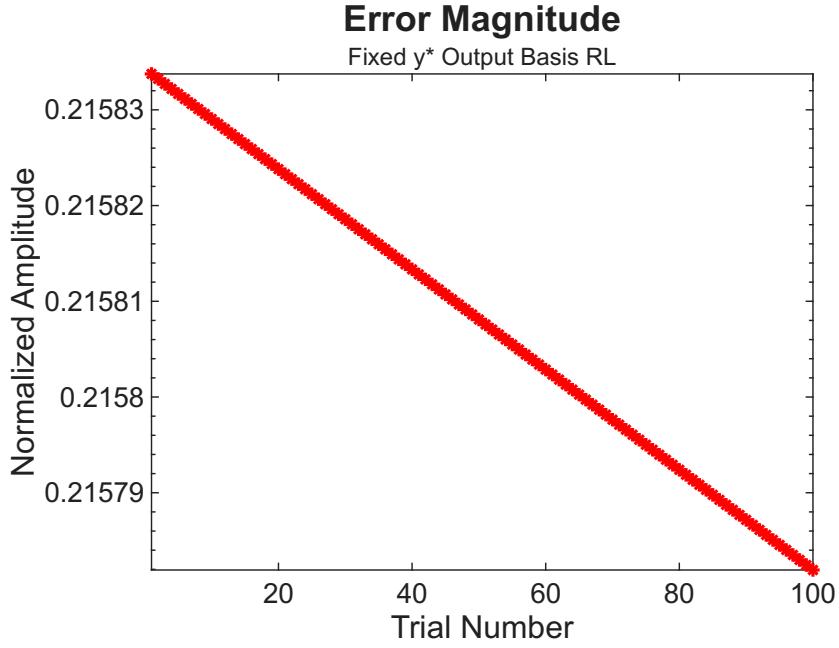


Figure 3.7: Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs and $\Phi_y = \underline{y}^*$. Given the relatively high R , 0 error is not achieved quickly but it can be seen to be being reduced.

for the output basis space $\Phi_y = \underline{y}^*$ and input basis space $\Phi_u = \Phi$.

We check this against our LQR solution of perfect knowledge to get

$$F_{LQR}^\gamma = \begin{bmatrix} -1.2603 \times 10^{-2} \\ -3.6281 \times 10^{-3} \\ \vdots \\ 1.8588 \times 10^{-4} \\ -9.6487 \times 10^{-3} \end{bmatrix} \quad (3.23)$$

which is practically identical⁶.

Applying this controller, we see in Figure 3.7 that it does bring down our error!

If this result seems too good to be true, that is because it is. Unfortunately, it is not possible to build a controller that is tall (more inputs produced from a smaller number of states) and have it be guaranteed stable. With some modifications to our Q and R , we end

⁶The small error on the final term is negligible.

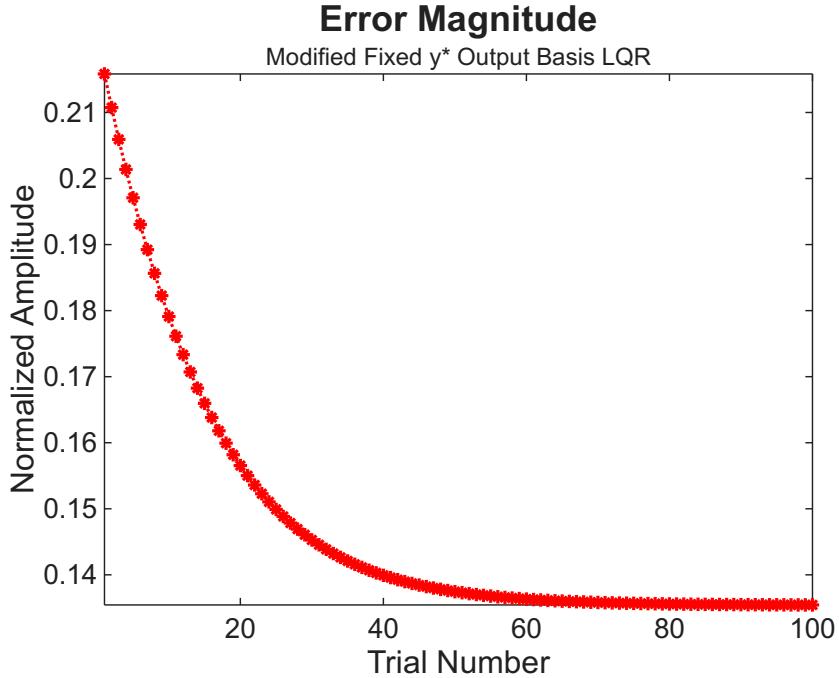


Figure 3.8: Complete Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs and $\Phi_y = \underline{y}^*$. It can now be seen that the tall controller constructed one element at a time, while efficient to build, does not bring about desired results on no error.

up with the results in Figure 3.8. Our earlier decrease in error can be seen to eventually hit a steady state. For initial improvements this may be a desirable approach then, but is not nearly sufficient to produce zero error.

Growing $\Phi_y = \Phi^b$ The final approach to try is relaxing our assumption of holding Φ_y fixed. Suppose we learn one conjugate function ϕ_1 , and use that for our basis functions. So $\Phi_u = \phi_1$ and $\Phi_y = \lambda\phi_1$ (scaled for reasons previously shown). The associated LQR controller would be 1×1 , and thus very unlikely to capture \underline{u}^* . So we generated and add another conjugate basis function ϕ_2 . Here, we learned the LQR controller where $\Phi_u = \phi_2$ and $\Phi_y = \lambda \begin{bmatrix} \phi_1 & \phi_2 \end{bmatrix}$. Now our LQR controller is 1×2 , and by stacking our previously learned controller on top, we form a lower-triangular matrix.

The appeal of this approach is that we get to minimize η in both input and output space, as we are applying RL to learn the associated controller. So after only 4 trials, we could update our first controller. Then 9, and so on, where each added controller i needs $(1 + i)^2$

trials to update itself once through RL techniques.

Example – Growing $\Phi_y = \Phi^b$ We establish the parameters our parameters from which we will define our conjugate basis functions ϕ , as we will be learning iteratively. Sticking with the earlier examples

$$Q = 100I_{20 \times 20} \quad R = 0_{20 \times 20} \quad (3.24)$$

Next, we establish the learning parameters with which we will be working. There is a frustrating balancing act between our R , $v(k)$, and now the amount by which we scale our Φ_y vs $\Phi_u - \lambda$. To demonstrate the functionality of this approach without extreme numbers, we choose

$$\begin{aligned} Q &= 100I_{20 \times 20} & R &= I_{20 \times 20} \\ \gamma &= 0.8 & v(k) &\in [-1, 1] & \lambda &= 10 \end{aligned} \quad (3.25)$$

The process now is very similar to our earlier examples with a fixed $\Phi_y = I$.

To start, we generate our first ϕ_1 . However, with each generated ϕ , we now must update our Φ_u and Φ_y . On this initial pass:

$$\Phi_u = \phi_1 \quad \Phi_y = 10\phi_1 \quad (3.26)$$

We now take these basis spaces and learn the ILC controller in their space. Taking care to grab $Q(1, 1)$ and $R(1, 1)$ as our costs, we learn that our first controller is

$$F_{\phi_1} = 0.1082 \quad (3.27)$$

Keep in mind, we want this to be a scalar. Both η_y and η_u are 1 in this initial pass.

We now move to our second trial. We generate ϕ_2 iteratively and updated our basis

spaces as

$$\Phi_u = \phi_2 \quad \Phi_y = 10 [\phi_1 \quad \phi_2] \quad (3.28)$$

Notice that now $\eta_y = 2$, and as such our found controller is the $1 \times 2 F_{\phi_2}$

$$F_{\phi_2} = [-0.0945 \quad 0.0085] \quad (3.29)$$

To combine this with our previous controller, which found the optimal input in the ϕ_1 space for error in that same space, we stack the two and pad it with zeros such that

$$F = \begin{bmatrix} F_{\phi_1} & 0 \\ F_{\phi_2} & \end{bmatrix} = \begin{bmatrix} 0.1082 & 0 \\ -0.0945 & 0.0085 \end{bmatrix} \quad (3.30)$$

We repeat the process until we have explored all 20 necessary ϕ s to ensure we fully capture \underline{u}^* . However in reality, one could check after each new controller and see if the result was satisfactory for the given situation.

In the end, we find

$$F = \begin{bmatrix} 0.1082 & 0 & \cdots & 0 & 0 \\ -0.0945 & 0.0085 & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0.0089 & 0.0044 & \cdots & -0.0012 & 0 \\ 0.0020 & -0.0009 & \cdots & 0.0013 & -0.0025 \end{bmatrix} \quad (3.31)$$

Note that this is not our F_{LQR}^γ in the space of $\Phi_u = \Phi$ and $\Phi_y = 10\Phi$. However, the bottom row of both should perfectly match as they are both defined in that space.

Due to our relatively large R , we wouldn't expect the application of this controller to converge quickly. It would seem that from Figure 3.9 this approach appears to work.

Fearing similar results from our earlier example of a fixed $\Phi_y = \lambda \underline{y}^*$, we check the poles of our system. Recall that anything > 1 indicates the system is unstable under the

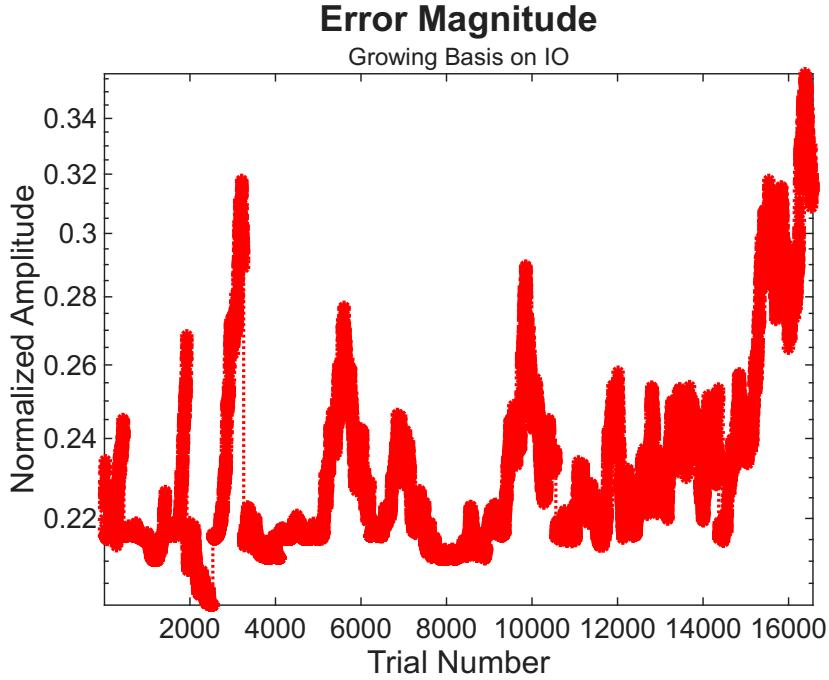


Figure 3.9: Error Progression through Policy Learning ILC Trials with Rolling ϕ on inputs and outputs

listed controller. For this example, we have 5 such poles that indicate instability. They exceed 1 by magnitudes of 1×10^{-7} or smaller. However small they may be, though, an unstable pole is still unstable. After running one billion trials, under our above parameters, the controller was still unable to send our system error to zero.

It would of course be desirable to lower the R . We have seen that smaller R s help reduce the magnitude of these poles. More on this in Section 5 on Future Work.

Chapter 4

Summary

Reinforcement Learning offers a powerful way to determine the Optimal Linear Quadratic Regulator Controller for a system, without any information on the system model. It's functionality translates well into the Iterative Learning Control problem, but the very formulation of ILC cripples the usefulness of RL techniques. A p step process will take our number of states (n) and inputs (r) to learn and scale them to $n_{ILC} = pn$ and $r_{ILC} = pr$, respectively. This exponentially increases the number of trials needed for any RL technique to learn.

To reverse this increase in dimensions, we explored the realm of basis spaces. The presented a path to infinitely reduce the dimensions of a given problem. For our ILC problem, it did not necessarily matter that our goal \underline{y}^* was capture, but we had to be able to properly generate our \underline{u}^* . Additionally, we showed that we cannot accurately create more rich enough control data when we have less error data coming in. As such, the conditions on basis spaces were found to be

$$\underline{u}^* \in \Phi_u \quad \eta_u \leq \eta_y \quad (4.1)$$

Under these conditions, we were practically no better off than we started. The only way to guarantee $\underline{u}^* \in \Phi_u$ was to define a full rank input basis space. Any attempt to start small and grow additionally made learning inefficient, as previously learned functions could be weighted differently to obtain optimality in the presence of other functions. The fact that independent inputs could lead to related outputs on the system forced us to take a different approach if we wished the grow a basis space. Enter conjugate basis functions.

Conjugate basis functions behave just as basis functions do, but they are specifically designed for a system and its costs. A conjugate basis function is defined such that its presence in a given basis space of other similarly-conjugate functions does not affect the optimality of learned weights on the others. This feature enables us to learn a controller in RL with respect to a finite number of basis functions at a time, and then grow those basis

functions without having to relearn what was already found.

Even while learning and expanding the basis space describing the system, the RL problem can be applied in reduced dimensions without compromising the optimality of previous controllers.

In the process of solving for conjugate basis functions, we can also find the optimal weighting for them. This means for a controllerless ILC problem, we can approach learning \underline{u}^* from two directions at once. The conjugate approach offers a way to learn the \underline{u}^* in a discrete number of steps, while the basis functions ϕ generated in the process can be used to learn a controller. This two-pronged approach means that even if we end up needing r_{ILC} basis functions to capture our \underline{u}^* , we can find that in a definite number of trials. Then both while finding this \underline{u}^* and after it has been found, we can learn the controller to keep it there should new disturbances or errors be introduced.

Chapter 5

Recommendations for Future Work

This work is by no means complete. There are plenty of ideas I wish to pursue in the future.

Note that this theory is complete for time-variant systems. The P matrix would have to be redefined but the logic is consistent when $ABCD$ are some function of k . This is remarkable because non-linear systems can be modelled with high-accuracy as a sum of many linear time-variant systems. Exploring a non-linear application of the found techniques could then be interesting.

The first item to attack more rigorously and quantify are the trade-offs between Q , R , Φ_u , Φ_y , and $v(k)$ values. For many examples, it was necessary to drastically increase R to show that the RL framework continued to function, but obviously increased the learning time necessary for a given problem. Exploring ways to reduce R without increasing the necessary exploration terms would drastically improve practicality of learning a controller. Especially in an ILC manufacturing scenario, if one could learn without purposely messing up parts, this would be extremely attractive. Several times, we had to scale up R , $v(k)$, and Φ_y to achieve desirable results in-line with our theory and hopes. Further understanding of why this had to be done, beyond my (questionable) mathematical intuition could open up avenues for incredibly robust and rapid learning controllers in a limited basis space.

Another idea to pursue is to expand on the final example shown in Section 3.1. Or rather - reduce it. Instead of constructing a lower-triangular matrix, imagine a scenario where one builds a diagonal controller. Now, each controller could be updated in 4 trials when a new basis function comes along. If this formulation worked (with a sufficiently small R), then it would be possible to find a controller for a complete system in $p(4 + 1)$ trials - 4 trials for the RL approach and 1 to generate the associated conjugate basis function. This is essentially what we shown when we learned with $\Phi_y = \lambda \underline{y}^*$, except our Φ_y changes instead of remaining fixed. There may be some benefits to this approach if the system has odd dynamics, or perhaps there are some techniques that benefit from diagonal controllers. I have tried initial passes at this, finding them all to be just barely unstable via pole inspection

– I hope that with a better understanding of R and Φ_y magnitudes, it would be possible to isolate whether this is an issue of the theory or numerical conditioning.

It would additionally be interesting to figure out what the learned \underline{u} is when the Φ_u does not capture the \underline{u}^* . It has been shown to not be the projection of \underline{u}^* . This is likely due to the fact that the \underline{u} is transformed by the P matrix, and then looped through Φ_y^+ conversions before going back through a controller \mathcal{L} to get β s (or $\delta\beta$ s) all before going back to being a \underline{u} . I would suspect that the projection of \underline{u}^* is then onto some space that is a transform of Φ_u , with matrices P and Φ_y involved.

Finally, we have investigated initial steps to reduce dimensions then grow, but no techniques to then work backwards. As shown in Eq. 2.108, it is possible to learn the weighting of some inputs even when they were not in the original \underline{u}^* . It would be desirable to see if there were a way to work backwards to remove certain basis functions, or combine them in with other functions. Take the basis space Φ

$$\Phi = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{b-1} & \phi_b \end{bmatrix} \quad (5.1)$$

that previously satisfied our conjunctionality condition such that

$$\Phi^T(R + P^T Q P)\Phi = I_{b \times b} \quad (5.2)$$

If we were to combine ϕ_{b-1} and ϕ_b into one term, we could reduce the dimension of our basis space to created Φ' . Now

$$\Phi'^T(R + P^T Q P)\Phi' = \begin{bmatrix} I_{(b-2) \times (b-2)} & 0 \\ 0 & 2 \end{bmatrix} \quad (5.3)$$

The feasibility of this approach is something worth pursing I believe, as it would enable initial input learning through LQL, then an evolving basis space of a fixed dimension that could be set manually. Thus any controllers learned would be able to be done if a fixed

amount of trials, set by the user.

Appendix A

Derivations

A.1 LQL Derivation

Recall we are starting with the cost function J :

$$J = \delta_j \underline{u}^T R \delta_j \underline{u} + e_j^T Q e_j \quad (\text{A.1})$$

where R is some $r_{ILC} \times r_{ILC}$ cost matrix, and Q a $n_{ILC} \times n_{ILC}$ one.

We need a $\delta_j \beta$ that minimizes the cost function, so we must express all relevant items in term of $\delta_j \beta$.

$$\begin{aligned} \delta_j \underline{u} &= \Phi(\beta_{j-1} + \delta_j \beta) \\ e_j &= e_{j-1} - P \delta_j \underline{u} \end{aligned} \quad (\text{A.2})$$

Plugging those into Eq. A.1

$$J = (\Phi(\beta_{j-1} + \delta_j \beta))^T R (\Phi(\beta_{j-1} + \delta_j \beta)) + (e_{j-1} - P \delta_j \underline{u})^T Q (e_{j-1} - P \delta_j \underline{u}) \quad (\text{A.3})$$

We first re-arrange items under their transpose operators. We utilize the fact that $(AB)^T = B^T A^T$ to expand our cost function to

$$\begin{aligned} J &= \beta_{j-1}^T \Phi^T R \Phi \beta_{j-1} + \beta_{j-1}^T \Phi^T R \Phi \delta_j \beta \\ &\quad + \delta_j \beta^T \Phi^T R \Phi \beta_{j-1} + \delta_j \beta^T \Phi^T R \Phi \delta_j \beta \\ &\quad + e_{j-1}^T Q e_{j-1} - e_{j-1}^T Q P \delta_j \underline{u} \\ &\quad - \delta_j \underline{u}^T P^T Q e_{j-1} + \delta_j \underline{u}^T P^T Q P \delta_j \underline{u} \end{aligned} \quad (\text{A.4})$$

It can be helpful here to remind ourselves of the dimensions of each component. Sticking to our convention of number of states = n , number of inputs = r , number of steps in the ILC process = p , and number of basis functions = b ¹, our components follow as

¹We are only describing our input in the basis space here

Variable	Dimensions
P	$pn \times pr$
Q	$pn \times pn$
R	$pr \times pr$
Φ	$pr \times b$
β_j	$b \times 1$
$\delta_j u$	$pr \times 1$
e_j	$pn \times 1$

As expected, each component in Eq. A.4 is a scalar. It is then important to remember that the transpose of a scalar is merely itself, and our R and Q matrices are symmetric such that $R = R^T$ and $Q = Q^T$. This leads to use being able to recognize certain components in Eq. A.4 are equal to each other as

$$\begin{aligned} \beta_{j-1}^T \Phi^T R \Phi \delta_j \beta &= \delta_j \beta^T \Phi^T R \Phi \beta_{j-1} \\ -e_{j-1}^T Q P \delta_j u &= -\delta_j u^T P^T Q e_{j-1} \end{aligned} \tag{A.5}$$

So that we may now write our cost function as

$$\begin{aligned} J &= \beta_{j-1}^T \Phi^T R \Phi \beta_{j-1} + 2\beta_{j-1}^T \Phi^T R \Phi \delta_j \beta \\ &\quad + \delta_j \beta^T \Phi^T R \Phi \delta_j \beta \\ &\quad + e_{j-1}^T Q e_{j-1} - 2e_{j-1}^T Q P \delta_j u \\ &\quad + \delta_j u^T P^T Q P \delta_j u \end{aligned} \tag{A.6}$$

Before we can take our derivative with respect to $\delta_j \beta$, we must make one final substitution. Recognize that

$$\delta_j u = \Phi (\beta_{j-1} + \delta_j \beta) \tag{A.7}$$

and expand our cost function to

$$\begin{aligned}
J = & \beta_{j-1}^T \Phi^T R \Phi \beta_{j-1} + 2\beta_{j-1}^T \Phi^T R \Phi \delta_j \beta \\
& + \delta_j \beta^T \Phi^T R \Phi \delta_j \beta \\
& + e_{j-1}^T Q e_{j-1} \\
& - 2e_{j-1}^T Q P \Phi \beta_{j-1} - 2e_{j-1}^T Q P \Phi \delta_j \beta \\
& + \beta_{j-1}^T \Phi^T P^T Q P \Phi \beta_{j-1} + \beta_{j-1}^T \Phi^T P^T Q P \Phi \delta_j \beta \\
& + \delta_j \beta^T \Phi^T P^T Q P \Phi \beta_{j-1} + \delta_j \beta^T \Phi^T P^T Q P \Phi \delta_j \beta
\end{aligned} \tag{A.8}$$

It should be evident now why this was included in the appendix, and not the main report. Now we may safely take the derivative of J with respect to $\delta_j \beta$, and to do so it is helpful to remember the following identities of matrix calculus

$$\frac{\partial}{\partial x} (x^T A x) = 2Ax \tag{A.9}$$

$$\frac{\partial}{\partial x} (y^T A x) = A^T y \tag{A.10}$$

$$\frac{\partial}{\partial x} (x^T A y) = Ay \tag{A.11}$$

Then,

$$\begin{aligned}
\frac{\partial J}{\partial [\delta_j \beta]} = & 0 + 2(\Phi^T R \Phi)^T \beta_{j-1} \\
& + 2\Phi^T R \Phi \delta_j \beta \\
& + 0 \\
& - 0 - 2\Phi^T P^T Q e_{j-1} \\
& + 0 + \Phi^T P^T Q P \Phi \beta \\
& + \Phi^T P^T Q P \Phi \beta_{j-1} + 2\Phi^T P^T Q P \Phi \delta_j \beta
\end{aligned} \tag{A.12}$$

Recognizing the common pattern of $\Phi^T R \Phi$ and $\Phi^T P^T Q P \Phi$, we introduce the shorthand

$$R_b = \Phi^T R \Phi \quad (\text{A.13})$$

$$Q_b = \Phi^T P^T Q P \Phi \quad (\text{A.14})$$

where we can see that $R_b^T = R_b$ and $Q_b^T = Q_b$

We re-write the above Eq. A.12 as

$$\begin{aligned} \frac{\partial J}{\partial [\delta_j \beta]} &= 2R_b \beta_{j-1} + 2R_b \delta_j \beta \\ &\quad - 2\Phi^T P^T Q e_{j-1} + Q_b \beta_{j-1} \\ &\quad + Q_b \beta_{j-1} + 2Q_b \delta_j \beta \end{aligned} \quad (\text{A.15})$$

Setting the derivative $\frac{\partial J}{\partial [\delta_j \beta]} = 0$ and re-arranging, we find

$$-2(R_b + Q_b) \delta_j \beta = 2(R_b + Q_b) \beta_{j-1} - 2\Phi^T P^T Q e_{j-1} \quad (\text{A.16})$$

Now we define the \mathcal{C} matrix we see in Eq. 2.77

$$\mathcal{C} = R_b + Q_b \quad (\text{A.17})$$

and divide both sides of our equation by 2 so

$$-\mathcal{C} \delta_j \beta = \mathcal{C} \beta_{j-1} - \Phi^T P^T Q e_{j-1} \quad (\text{A.18})$$

Remember the δ_j operator defined as in Eq. 1.52. Reversing that, we see

$$-\mathcal{C}(\beta_j - \beta_{j-1}) = \mathcal{C} \beta_{j-1} - \Phi^T P^T Q e_{j-1} \quad (\text{A.19})$$

Subtracting $\mathcal{C}\beta_{j-1}$ from both sides,

$$-\mathcal{C}\beta_j = -\Phi^T P^T Q e_{j-1} \quad (\text{A.20})$$

\mathcal{C} can be shown to be $b \times b$, so multiplying it by its inverse produces the identity matrix. We left multiply by $-\mathcal{C}^{-1}$ to get

$$\beta_j = \mathcal{C}^{-1} \Phi^T P^T Q e_{j-1} \quad (\text{A.21})$$

Appendix B

Presentation

B.1 Final Presentation

An uploaded recording of the final presentation may be found at <https://youtu.be/z7bC4mVHPNk>

B.2 Background

```
1 %A General Intro to State Space and Modern Control Theory
2 %Noah Dunleavy
3 %Honors Thesis under direction of Professor Minh Q. Phan
4 %Thayer School of Engineering
5 clc; clear;
6 addpath('Saved Data', 'Functions');
7 setDefaultFigProp();
8 plot_all = false; %most run time goes into generating figs
9 update_file_path = -1;

    %'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis
    Images\General Intro'; %set this to the save location if want
    to update figures, or -1 if not
10 if update_file_path ~= -1
11     keyboard %ensure we have to be very concious about ever
        updating all the images
12 end
13
14 %% Description
15 %
16 This intro code provides the basics of 'known' status in the
    field. To those familiar with the Thayer School of
    Engineering's curriculum, it is a recap of ENGS145 - Modern
    Control Theory.
17
18 We begin with system formulation and representation in the matrix
    form, and how we resolve the difference between the
    continuous nature of the world, and the discrete ability of
```

computers. Here, we use the '**Zero-Order-Hold**' approach. Next the idea of pole placement is introduced, and it is demonstrated how the further from the origin the poles are, the longer control takes. A deadbeat controller is used to highlight this, **which** is the time-optimal solution **for any** system.

19

20 One will note that the deadbeat controller, **while** time optimal, requires significant control effort that may not be realistic or safe **for a real** system. That leads us to our introduction of the Linear Quadratic Regulation (LQR) controller, **which** minimizes a cost **function** composed of both system inputs and states. We show the difference in responses when **input** has two different relative strengths when compared to the state.

21

22 Next we introduce the Iterative Learning Control (ILC) problem and a controller, and show that it can learn to generate **any** output (so long as permitted by the physical characteristics of the system).

23

24 All of the previous examples rely on perfect knowledge not typically possible in the **real** world (although we can approximate and identify with System Identification Methods), the process of Reinforcement Learning (RL) is shown via the Policy Iteration method. It can be, and is, shown to **find** the LQR controller as defined by its cost **function**.

25

26 Finally, the idea of basis functions is introduced briefly to lay the

```

27 groundwork for subsequent explorations.

28 %}

29

30 %% System Creation

31

32 %Scalar Representation

33 m1 = 1; %Mass of the block, [kg]

34 m2 = 0.5;

35 num_masses = 2;

36 k1 = 100; %Spring constant, [N/m]

37 k2 = 200;

38 c1 = 1; %Damping Coeffeceint, [Ns/m]

39 c2 = 0.5;

40

41 %Matrix Formulations

42 M = diag([m1, m2]); %mass matrix

43 K = [k1 + k2, -k2; -k2, k2]; %stiffness matrix

44 C_damp = [c1 + c2, -c2; -c2, c2]; %Damping matrix

45

46 %Continuous Statespace Formulations

47 Ac = [zeros(num_masses), eye(num_masses); -M^-1 * K, -M^-1 * C_damp];

48 Bc = [zeros(num_masses); M^-1];

49 C = [1, 0, 0, 0; 0, 1, 0, 0]; %Monitor the block positions as outputs

50 D = [0, 0; 0, 0]; %

51

52 %Dimensional Variables

53 num_states = height(Ac); %Ac is n x n

```

```

54 num_inputs = width(Bc); %Bc is n x r
55 num_outputs = height(C); %C is m x n
56
57 %Ensure against Nyquist
58 eigens = eig(Ac); %Ac captures the modes of a given system
59 natural_freqs = imag(eigens)/(2*pi);%Get all the natural
   frequencies in Hz (convert from rad/sec)
60 max_freq = max(natural_freqs); %The highest frequency is the one
   we must cater to
61 nyquist_freq = 2 * max_freq; %Minimum sampling frequency to avoid
   nyquist sampling
62 nyquist_rate = 1 / nyquist_freq; %Sample frequency to period
63
64 dt = 0.01; %Set the sampling rate we will be using
65 if (dt > nyquist_rate) %Ensure that the true sampling period is
   below the nyquist specifications
66   dt = nyquist_rate / 10; %Set the dt to one that is sufficient,
   best practice is to scale by a factor of 2-10 beyond nyquist
67   sprintf('Selected sample rate was insufficient, setting dt =
   %.2e seconds', dt)
68 end
69
70 %Check Controllability
71 if ~is_controllable(Ac, Bc) %always check it is even possible to
   control
72   printf('The Continuous System is not Controllable!')
73 end
74
75 %Convert to Discrete

```

```

76 [A, B] = c2d(Ac, Bc, dt); %Performs the necessary discretization
    operations - verify this for yourself
77
78 %Set initial conditions
79 initial_displacement = [1; 0]; %displace the cooresponding blocks
    initially by this amount, [meters]
80 initial_velocity = [0; 2]; %Start cooresponding blocks with this
    velocity [m/s]
81 x0 = [initial_displacement; initial_velocity]; %The top half sets
    position ICs, and the bottom half sets velocity
82
83 %Simulation Duartion
84 max_samples = 500; %set the maximum number of steps out to
    simulate (note: max k will be -1, since we start at k=0)
85 max_time = max_samples * dt; %Important distinction: sample STEPS
    is not the same as time
86
87 save('Saved Data\thesis_system.mat', 'A', 'B', 'C', 'D', 'x0',
    'num_states', 'num_outputs', 'num_inputs'); %save the system
    for consistency across all following code
88 %% Open-Loop Simulation
89 cont_vs_disc_resolution = 1; %Additional resolution to give
    continuous simulations for smoother plots, and so the evident
    differences can be seen between continious and discrete
90 %Setting to 1 speeds up rendering process, and can just use
    matlab's
91 %'plot' vs stairs to highlight
92

```

```

93 open_time_continuous = linspace(0, max_time, max_samples *
94                                         cont_vs_disc_resolution)'; %Generate the time scale at the
95                                         higher resolution
96 continuous_system = ss(Ac, Bc, C, D); %Turn the continuous
97                                         matrixies into a system for simulation
98 continuous_open_outputs = lsim(continuous_system,
99                                         zeros(height(open_time_continuous), num_inputs),
100                                        open_time_continuous, x0);
101 [discrete_open_outputs, discrete_open_states] = dlsim(A, B, C, D,
102                                         zeros(max_samples, num_inputs), x0); %Generate x(0) ->
103                                         x(max_samples - 1), and cooresponding outputs (y)
104
105 %% Render Open Loop Responses
106 if plot_all
107     %Block 1 Position
108     fig = figure('Name', 'Open-loop Mass 1 Position');
109     plot(open_time_continuous, continuous_open_outputs(:, 1));
110
111     title('Mass 1 Position');
112     subtitle('Open-Loop', 'FontSize', getappdata(groot,
113                                         'DefaultFontSize'));
114     xlabel('Time (sec)') %only point where we will be using time
115                                         as the x axis
116     ylabel('Position (m)')
117     save_figure(update_file_path, fig, 'Continuous Open-Loop -
118                                         Mass 1'); %save just the continuous Model
119
120     %Add in discrete element
121     hold on;

```

```

111    stairs(dt * (0:(max_samples-1)), discrete_open_outputs(:, 1));

    %note the dt scaling such that axeses are the same

112    hold off;

113    legend('Continuous', 'Discrete')

114    save_figure(update_file_path, fig);

115

116    %Mass 2 position

117    fig = figure('Name', 'Open-loop Mass 2 Position');

118    plot(open_time_continuous, continuous_open_outputs(:, 2));

119    title('Mass 2 Position');

120    subtitle('Open-Loop', 'FontSize', getappdata(groot,
121        'DefaultSubtitleFontSize'));

121    xlabel('Time (sec)')

122    ylabel('Position (m)')

123    save_figure(update_file_path, fig, 'Continuous Open-Loop -
124        Mass 2'); %save just the continuous Model

125    %Add in discrete element

126    hold on;

127    stairs(dt * (0:(max_samples-1)), discrete_open_outputs(:, 2));

128    hold off;

129    legend('Continuous', 'Discrete')

130    save_figure(update_file_path, fig);

131

132    %Zoomed in to show the models match

133    num_zoomed_in = 11;

134    fig = figure('Name', 'Zoomed-in Open-loop Mass 1 Position');

135    plot(open_time_continuous(1:num_zoomed_in),
continuous_open_outputs(1:num_zoomed_in, 1));
    hold on;

```

```

136 stairs(dt * (0:(num_zoomed_in-1)),
137 discrete_open_outputs(1:num_zoomed_in, 1)); %note the dt
scaling such that axes are the same
138 scatter(dt * (0:(num_zoomed_in-1)),
139 discrete_open_outputs(1:num_zoomed_in, 1), 'filled', '*',
'MarkerFaceColor', 'red', 'MarkerEdgeColor', 'red',
'SizeData', 90);
140 hold off;
141 legend('Continuous', 'Discrete')
142 title('Mass 1 Position');
143 subtitle('Open-Loop', 'FontSize', getappdata(groot,
'DefaultSubtitleFontSize'));
144 xlabel('Time (sec)') %only point where we will be using time
as the x axis
145 ylabel('Position (m)')
146
147 fig = figure('Name', 'Zoomed-in Open-loop Mass 2 Position');
148 plot(open_time_continuous(1:num_zoomed_in),
continuous_open_outputs(1:num_zoomed_in, 2));
149 hold on;
150 stairs(dt * (0:(num_zoomed_in-1)),
discrete_open_outputs(1:num_zoomed_in, 2)); %note the dt
scaling such that axes are the same
151 scatter(dt * (0:(num_zoomed_in-1)),
discrete_open_outputs(1:num_zoomed_in, 2), 'filled', '*',
'MarkerFaceColor', 'red', 'MarkerEdgeColor', 'red',
'SizeData', 90);

```

```

152 hold off;
153 legend('Continuous', 'Discrete')
154 title('Mass 2 Position');
155 subtitle('Open-Loop', 'FontSize', getappdata(groot,
156 'DefaultSubtitleFontSize'));
157 xlabel('Time (sec)') %only point where we will be using time
as the x axis
158 ylabel('Position (m)')
159 xlim([0, dt*(num_zoomed_in-1)])
160 save_figure(update_file_path, fig);
161 end
162 %% Simple Pole Placement Controller
163 pole_locations = [0.5 + 0.5i, 0.5 - 0.5i, -0.7 + 0.1i, -0.7 -
0.1i]; %if imaginary, must be complex conjugates of eachother
164 placed_controller = -place(A, B, pole_locations);
165 verify_poles = eig(A + B * placed_controller);
166 if any(abs(verify_poles) >= 1)
167 printf('A Pole was placed outside of the unit circle, unstable
controller!')
168 end
169
170 %Visualize the poles
171 if plot_all
172 plot_pole_placement('Simple Pole Placement', 'Simple Pole
Placement', verify_poles, pole_locations, update_file_path);
173 end
174 closed_pole_samples = 60;
175 pole_states = zeros(num_states, closed_pole_samples);

```

```

176 pole_inputs = zeros(num_inputs, closed_pole_samples);
177 pole_outputs = zeros(num_outputs, closed_pole_samples);
178 pole_states(:, 1) = x0; %set the initial conditions
179
180 for k = 1:closed_pole_samples
181     pole_inputs(:, k) = placed_controller * pole_states(:, k);
182     %u(k) = F * x(k) - linear control law
183     pole_states(:, k + 1) = A * pole_states(:, k) + B *
184         pole_inputs(:, k); %x(k+1) = A*x(k) + B*u(k)
185     pole_outputs(:, k) = C * pole_states(:, k) + D *
186         pole_inputs(:, k); %y(k) = C*x(k) + D*u(k)
187 end
188
189 if plot_all
190     plot_two_mass('Pole Placement', 'Closed-Loop System with Pole
191     Placement', pole_outputs, pole_inputs, update_file_path);
192     %function to plot positions and forces
193 end
194
195 %% Deadbeat Controller
196 deadbeat_poles = [-0.00001, 0.00001, 0.00000i, -0.00000i]; %place
197     will not allow for multiple poles at same location, so make
198     all really close to 0
199 deadbeat_controller = -place(A, B, deadbeat_poles);
200 verify_poles = eig(A + B * deadbeat_controller);
201 if any(abs(verify_poles) > 1)
202     printf('A Pole was placed outside of the unit circle, unstable
203         controller!')
204 end

```

```

197
198 %Visualize the poles
199 if plot_all
200     plot_pole_placement('Deadbeat Pole Placement', 'Deadbeat Pole
201         Placement', verify_poles, deadbeat_poles, update_file_path);
202 end
203 deadbeat_samples = 20;
204 deadbeat_states = zeros(num_states, deadbeat_samples);
205 deadbeat_inputs = zeros(num_inputs, deadbeat_samples);
206 deadbeat_outputs = zeros(num_outputs, deadbeat_samples);
207 deadbeat_states(:, 1) = x0; %set the initial conditions
208
209 for k = 1:deadbeat_samples
210     deadbeat_inputs(:, k) = deadbeat_controller *
211         deadbeat_states(:, k); %u(k) = F * x(k) - linear control law
212     deadbeat_states(:, k + 1) = A * deadbeat_states(:, k) + B *
213         deadbeat_inputs(:, k); %x(k+1) = A*x(k) + B*u(k)
214     deadbeat_outputs(:, k) = C * deadbeat_states(:, k) + D *
215         deadbeat_inputs(:, k); %y(k) = C*x(k) + D*u(k)
216 end
217
218 if plot_all
219     plot_two_mass('Deadbeat Controller', 'Deadbeat Controller',
220         deadbeat_outputs, deadbeat_inputs, update_file_path);
221 end
222
223 %% LQR Controller
224 Q = 100 * eye(num_states); %cost of each states being away from 0
225 R = 1 * eye(num_inputs); %cost od each input away from 0

```

```

221 gamma = 0.8; %learning factor
222
223 %Consturct an LQR controller
224 %The LQR controller minimizes the cost function  $J = u' * R * u +$ 
225 % $x' * x$ 
226 F_lqr = discounted_LQR(A, B, gamma, Q, R);
227
228 lqr_samples = 200;
229 lqr_states = zeros(num_states, lqr_samples);
230 lqr_inputs = zeros(num_inputs, lqr_samples);
231 lqr_outputs = zeros(num_outputs, lqr_samples);
232 lqr_states(:, 1) = x0; %set the initial conditions
233
234 for k = 1:lqr_samples
235     lqr_inputs(:, k) = F_lqr * lqr_states(:, k); % $u(k) = F * x(k)$ 
236     % linear control law
237     lqr_states(:, k + 1) = A * lqr_states(:, k) + B *
238     lqr_inputs(:, k); % $x(k+1) = A*x(k) + B*u(k)$ 
239     lqr_outputs(:, k) = C * lqr_states(:, k) + D * lqr_inputs(:, k);
240     % $y(k) = C*x(k) + D*u(k)$ 
241 end
242 %Find where it placed the poles
243 lqr_poles = eig(A + B * F_lqr);
244 if plot_all
245     plot_pole_placement('Big Q LQR Pole Locations', sprintf('Q/R =
246     %.d Pole Placement', Q(1, 1)/R(1, 1)), lqr_poles, -1,
247     update_file_path);
248 end

```

```

244 %Plot full system out
245 if plot_all
246     plot_two_mass('Big Q LQR Controller', sprintf('With LQR
247         Controller (Q/R=%d)', Q(1, 1)/R(1, 1)), lqr_outputs,
248         lqr_inputs, update_file_path);
249 end
250
251 %% Demo Higher R LQR
252 R_big = 10 * eye(num_inputs); %make relative weightings more
253 % skewed to input
254 gamma = 0.8;
255
256 big_R_F_lqr = discounted_LQR(A, B, gamma, Q, R_big);
257
258 big_R_lqr_samples = 1200;
259 big_R_lqr_states = zeros(num_states, big_R_lqr_samples);
260 big_R_lqr_inputs = zeros(num_inputs, big_R_lqr_samples);
261 big_R_lqr_outputs = zeros(num_outputs, big_R_lqr_samples);
262 big_R_lqr_states(:, 1) = x0; %set the initial conditions
263
264 for k = 1:big_R_lqr_samples
265     big_R_lqr_inputs(:, k) = big_R_F_lqr * big_R_lqr_states(:, k);
266     %u(k) = F * x(k) - linear control law
267     big_R_lqr_states(:, k + 1) = A * big_R_lqr_states(:, k) + B *
268         big_R_lqr_inputs(:, k); %x(k+1) = A*x(k) + B*u(k)
269     big_R_lqr_outputs(:, k) = C * big_R_lqr_states(:, k) + D *
270         big_R_lqr_inputs(:, k); %y(k) = C*x(k) + D*u(k)
271 end

```

```

267 %Find where it placed the poles
268 big_R_lqr_poles = eig(A + B * big_R_F_lqr);
269 if plot_all
270     plot_pole_placement('Big R Pole Placement', sprintf('Q/R = %.d
271 Pole Placement', Q(1, 1)/R_big(1, 1)), big_R_lqr_poles, -1,
272 update_file_path);
273 end
274 %Plot full system out
275 if plot_all
276     plot_two_mass('Big R LQR Controller', sprintf('With LQR
277 Controller (Q/R = %.d)', Q(1, 1)/R_big(1, 1)),
278 big_R_lqr_outputs, big_R_lqr_inputs, update_file_path);
279 end
280
281 %% ILC Introduction
282 %Setup ILC Parameters
283 p = 100; %number of steps to a process
284 num_ilc_states = num_outputs * p; %effective states for ILC are
285 %the errors, generated from outputs
286 num_ilc_inputs = num_inputs * p; %r inputs for every time step
287
288 x0_ilc = x0;
289
290 %Descriptive Matrix
291 [P_full, d_full] = P_from_ABCD(A, B, C, D, p, x0_ilc); %generate
292 %a descriptive matrix which captures the relation of inputs to
293 %outputs
294
295 %ILC State-space:

```

```

289 % e(j+1) = e(j) - P*del_u(j)
290 % y(j) = e(j)
291 A_ilc = eye(num_ilc_states);
292 B_ilc = -P_full;
293 C_ilc = eye(num_ilc_states);
294 D_ilc = 0;
295
296 if ~is_controllable(A_ilc, B_ilc) %always check it is even
    possible to control
    printf('The ILC System is not Controllable!')
298 end
299
300 F_ilc = 0.8 * pinv(P_full); %Note: any controller that places
    poles inside the unit circle works.
301
302 %Set the Goal Outputs (only have goals for p steps, regardless of
    lead)
303 %Use function draw_to_XY(p) if you would like to set your own
304 %drawing/special output. This is not written to be super robust,
    choose
305 %matching p values
306 %[drawn_x, drawn_y] = draw_to_XY(p);
307 %save(sprintf('Saved Data\shape_p%.d.mat',p), 'drawn_x',
    'drawn_y'); %save to a file so we only have to do this once
308
309 y_star_x = cos(2 * pi * (0:(p-1)) / p)';
310 y_star_y = sin(2 * pi * (0:(p-1)) / p)';
311 goal_matrix = [y_star_x, y_star_y]; %stack inputs next to
    eachother

```

```

312 y_star = reshape(goal_matrix', [], 1); %combine the seperate
     goals of each output into one vertical vector, alternating as
     necessary
313
314 %ILC Structure
315 num_trials = 10; %number of trials to learn input
316 ILC_Trial(num_trials).output = [];
317 ILC_Trial(num_trials).input = [];
318 ILC_Trial(num_trials).del_u = []; %relevant control parameter
319 ILC_Trial(num_trials).output_error = []; %relevant control
     parameter
320
321 %First Trial
322 trial_num = 1;
323 ILC_Trial(trial_num).del_u = zeros(num_ilc_inputs, 1); %first
     trial we set whatever del_u we want
324 ILC_Trial(trial_num).input = zeros(num_ilc_inputs, 1);
     %Similarly, first input is arbitrary
325
326 ILC_Trial(trial_num).output = [C * x0_ilc + D *
     ILC_Trial(trial_num).input(1:num_inputs); P_full *
     ILC_Trial(trial_num).input + d_full]; %simulate y(0) -> y(p +
     num_lead), since y = Pu + d ignores y(0), use IC
327 relevant_output = ILC_Trial(trial_num).output((num_outputs +
     1):end); %only compare to the ones we can / want to control
328 ILC_Trial(trial_num).output_error = y_star - relevant_output;
329
330 %Subsequent Iterations
331 for trial_num = 2:num_trials

```

```

332 %Inputs
333 ILC_Trial(trial_num).del_u = F_ilc * ILC_Trial(trial_num -
334 1).output_error;
335 ILC_Trial(trial_num).input = ILC_Trial(trial_num).del_u +
336 ILC_Trial(trial_num - 1).input; %d_u(j) = u(j) - u(j-1)
337 %Output
338 ILC_Trial(trial_num).output = [C * x0_ilc + D *
339 ILC_Trial(trial_num).input(1:num_inputs); P_full *
340 ILC_Trial(trial_num).input + d_full];
341 relevant_output = ILC_Trial(trial_num).output((1 +
342 num_outputs):end);
343 %Error
344 ILC_Trial(trial_num).output_error = y_star - relevant_output;
345 end
346
347 if plot_all
348 to_plot_ilc = [1, 2, 5, num_trials];
349 plot_dual_ilc('Placed Controller ILC', 'ILC with a Placed
350 Controller', ILC_Trial, y_star, -1, to_plot_ilc,
351 update_file_path); %update_file_path
352 end
353
354 %% Arbitrary ILC
355 %Setup ILC Parameters
356 if false
357 clear 'ILC_Trial' %make sure we do not carry over any ILC from
358 previous
359 p = 500; %number of steps to a process

```

```

352 num_ilc_states = num_outputs * p; %effective states for ILC
353 %are the errors, generated from outputs
354 num_ilc_inputs = num_inputs * p; %r inputs for every time step
355
356 x0_ilc = x0;
357
358 %Descriptive Matrix
359 [P_full, d_full] = P_from_ABCD(A, B, C, D, p, x0_ilc);
360 %generate a descriptive matrix which captures the relation of
361 %inputs to outputs
362
363 A_ilc = eye(num_ilc_states);
364 B_ilc = -P_full;
365 C_ilc = eye(num_ilc_states);
366 D_ilc = 0;
367
368 if ~is_controllable(A_ilc, B_ilc) %always check it is even
369 %possible to control
370 %printf('The ILC System is not Controllable!')
371 end
372
373 F_ilc = 0.8 * pinv(P_full); %Note: any controller that places
374 %poles inside the unit circle works.
375
376 loadedShape = load('Saved Data\dartmouth_p500.mat'); %read in
377 %the file
378 y_star_x = loadedShape.drawn_x';
379 y_star_y = loadedShape.drawn_y';
380 scale = 10;

```

```

375 goal_matrix = [y_star_x, y_star_y]; %stack inputs next to
eachother
376 y_star = scale * reshape(goal_matrix', [], 1); %combine the
seperate goals of each output into one vertical vector,
alternating as necessary
377
378 %ILC Structure
379 num_trials = 10; %number of trials to learn input
380 ILC_Trial(num_trials).output = [];
381 ILC_Trial(num_trials).input = [];
382 ILC_Trial(num_trials).del_u = []; %relevant control parameter
383 ILC_Trial(num_trials).output_error = []; %relevant control
parameter
384
385 %First Trial
386 trial_num = 1;
387 ILC_Trial(trial_num).del_u = zeros(num_ilc_inputs, 1); %first
trial we set whatever del_u we want
388 ILC_Trial(trial_num).input = zeros(num_ilc_inputs, 1);
%Similarly, first input is arbitrary
389
390 ILC_Trial(trial_num).output = [C * x0_ilc + D *
ILC_Trial(trial_num).input(1:num_inputs); P_full *
ILC_Trial(trial_num).input + d_full]; %simulate y(0) -> y(p +
num_lead), since y = Pu + d ignores y(0), use IC
391 relevant_output = ILC_Trial(trial_num).output((num_outputs +
1):end); %only compare to the ones we can / want to control
392 ILC_Trial(trial_num).output_error = y_star - relevant_output;
393

```

```

394 %Subsequent Iterations
395 for trial_num = 2:num_trials
396     %Inputs
397     ILC_Trial(trial_num).del_u = F_ilc * ILC_Trial(trial_num -
398         1).output_error;
399     ILC_Trial(trial_num).input = ILC_Trial(trial_num).del_u +
400         ILC_Trial(trial_num - 1).input; %d_u(j) = u(j) - u(j-1)
401     %Output
402     ILC_Trial(trial_num).output = [C * x0_ilc + D *
403         ILC_Trial(trial_num).input(1:num_inputs); P_full *
404         ILC_Trial(trial_num).input + d_full];
405     relevant_output = ILC_Trial(trial_num).output((1 +
406         num_outputs):end);
407     %Error
408     ILC_Trial(trial_num).output_error = y_star -
409         relevant_output;
410 end
411
412 if plot_all
413     to_plot_ilc = [1, 2, 5, num_trials];
414
415     resave_dartmouth = update_file_path
416     %Super explicitly re-save the dartmouth plot. When doing
417     so, go into
418         %save_figure and adjust legend location for ONLY this one.
419         Just
420         %because where it draws things

```

```

414     plot_dual_ilc('Dartmouth ILC', 'Arbitrary ILC with a Placed
415         Controller', ILC_Trial, y_star, -1, to_plot_ilc,
416         resave_dartmouth); %update_file_path
417
418 end
419
420 %% Reinforcement Learning - Policy Iteration
421 converged_k = 20; %number of steps to run out for with
422     'converged' controller
423 num_controllers = 5; %number of iterations of each controller
424     before updating
425
426 total_tries = num_controllers * num_collections + converged_k;
427
428 F_policy = zeros(num_inputs, num_states, 1); %start with no
429     controller
430
431 policy_states = zeros(num_states, total_tries);
432 policy_states(:, 1) = x0;
433 policy_inputs = zeros(num_inputs, total_tries);
434 policy_outputs = zeros(num_outputs, total_tries);
435 k = 1;
436
437

```

```

436
437 for iteration = 1:num_controllers
438     %Reset Stacks each iteration
439     Uk_stack = zeros(num_collections, 1);
440     Xk_stack = zeros(num_collections, (Pj_dim)^2);
441
442     %Compute stack infos
443     for collection = 1:num_collections
444         %Simulate nature
445         %policy_inputs(:, k) = rand(num_inputs, 1); %can replicate
446         %the 'random state' approach by just randomizing inputs
447         policy_inputs(:, k) = F_policy(:, :, end) *
448         policy_states(:, k) + rand_range(num_inputs, 1, -1, 1);
449         %compute input and exploration term
450
451         %Next Step regardless of nature vs random
452         policy_states(:, k + 1) = A * policy_states(:, k) + B *
453         policy_inputs(:, k);
454         policy_outputs(:, k) = C * policy_states(:, k) + D *
455         policy_inputs(:, k);
456
457
458     %Construct stacks
459     xu_stack = [policy_states(:, k); policy_inputs(:, k)];
460     xu_next_stack = [policy_states(:, k + 1); F_policy(:, :, end) *
461     policy_states(:, k + 1)];
462     Xk_stack(collection, :) = kron(xu_stack', xu_stack') -
463     gamma * kron(xu_next_stack', xu_next_stack');

```

```

457     Uk_stack(collection, :) = policy_states(:, k)' * Q *
458         policy_states(:, k) + policy_inputs(:, k)' * R *
459         policy_inputs(:, k); %where Q and R come into play
460
461     %Move to next
462     k = k + 1;
463
464 end
465
466 % [P, S, V] = svd(Xk_stack);
467
468 % rank(S), iteration %we do not need the stack to be full
469 %rank, but it
470
471 % must not drop ranks thorough iterations
472
473 %Calculate P and new controller
474
475 PjS = pinv(Xk_stack) * Uk_stack;
476
477 Pj = reshape(PjS, Pj_dim, Pj_dim); %undo the stack
478
479 Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
480 %reduces error)
481
482 Pjuu = Pj((num_states+1):end, (num_states+1):end); %grab the
483 %bottom right quadrant, Puu
484
485 PjxuT = Pj((num_states+1):end, 1:num_states); %Grab bottom
486 %left, Pxu transpose
487
488 new_F = -pinv(Pjuu) * PjxuT; %definition of controller
489
490 F_policy(:, :, end + 1) = new_F; %add to end of list
491
492 end
493
494
495 %Plot / simulate past convergance (without exploration/noise) so
496 %actual
497
498 %control possible
499
500 for ndx = 1:converged_k
501
502     %Simulate nature

```

```

479 policy_inputs(:, k) = F_policy(:, :, end) * policy_states(:, k);
480 policy_states(:, k + 1) = A * policy_states(:, k) + B *
481 policy_inputs(:, k);
482 policy_outputs(:, k) = C * policy_states(:, k) + D *
483 policy_inputs(:, k);
484 k = k + 1;
485 end
486
487 if plot_all
488 plot_two_mass('Policy Iteration IO', 'Under Policy Iteration',
489 policy_outputs, policy_inputs, update_file_path);
490 plot_controller_history('Policy Iteration Controller', 'Under
491 Policy Iteration', F_policy, -1, -1, update_file_path); %1s
492 % denote to plot every state and input weight
493 end
494
495
496
497 %% Reinforcement Learning - Input Decoupling
498 converged_k = 20;
499 num_controllers = 5;
500 Pj_dim = (num_states + 1); %effective input count is 1 now
501 num_collections = Pj_dim^2;
502
503 total_tries = num_controllers * num_collections + converged_k;
504
505 F_decoupled = zeros(num_inputs, num_states, 1); %start with no
506 controller

```

```

501 decoupled_states = zeros(num_states, total_tries);
502 decoupled_states(:, 1) = x0;
503 decoupled_inputs = zeros(num_inputs, total_tries);
504 decoupled_outputs = zeros(num_outputs, total_tries);
505 k = 1;
506
507 for iteration = 1:num_controllers
508     for input_num = 1:num_inputs
509         %Reset Stacks each iteration
510         Uk_stack = zeros(num_collections, 1);
511         Xk_stack = zeros(num_collections, (Pj_dim)^2);
512         F_i = F_decoupled(input_num, :, end); %input on the current
513         %input of inspection
514         %Compute stack infos
515         for collection = 1:num_collections
516             %Simulate nature
517             %decoupled_inputs(:, k) = rand(num_inputs, 1); %can
518             %replicate the 'random state' approach by just randomizing
519             %inputs
520             decoupled_inputs(:, k) = F_decoupled(:, :, end) *
521             decoupled_states(:, k); %compute input
522             decoupled_inputs(input_num, k) = rand_range(1, 1, -1,
523             1); %+decoupled_inputs(input_num, k); %
524
525             %Next Step regardless of nature vs random
526             decoupled_states(:, k + 1) = A * decoupled_states(:, k)
527             + B * decoupled_inputs(:, k);
528             decoupled_outputs(:, k) = C * decoupled_states(:, k) + D
529             * decoupled_inputs(:, k);

```

```

523
524
525     %Construct stacks
526
527     xu_stack = [decoupled_states(:, k);
528
529         decoupled_inputs(input_num, k)];
530
531     xu_next_stack = [decoupled_states(:, k + 1); F_i *
532
533         decoupled_states(:, k + 1)];
534
535     Xk_stack(collection, :) = kron(xu_stack', xu_stack') -
536
537     gamma * kron(xu_next_stack', xu_next_stack');
538
539     Uk_stack(collection, :) = decoupled_states(:, k)' * Q *
540
541     decoupled_states(:, k) + decoupled_inputs(:, k)' * R *
542
543     decoupled_inputs(:, k); %where Q and R come into play
544
545
546     %Move to next
547
548     k = k + 1;
549
550 end
551
552
553 %Calculate P and new controller
554
555 PjS = pinv(Xk_stack) * Uk_stack;
556
557 Pj = reshape(PjS, Pj_dim, Pj_dim); %undo the stack
558
559 Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
560
561 reduces error)
562
563 Pjuu = Pj((num_states+1):end, (num_states+1):end); %grab
564
565 the bottom right quadrant, Puu
566
567 PjxuT = Pj((num_states+1):end, 1:num_states); %Grab bottom
568
569 left, Pxu transpose
570
571 new_F_i = -pinv(Pjuu) * PjxuT; %definition of controller
572
573 F_decoupled(:, :, end + 1) = F_decoupled(:, :, end); %copy
574
575 old controller

```

```

543     F_decoupled(input_num, :, end) = new_F_i; %update current
      input
544   end
545 end
546
547 %Plot / simulate past convergance (without exploration/noise) so
      actual
548 %control possible
549 for ndx = 1:converged_k
      %Simulate nature
550   decoupled_inputs(:, k) = F_decoupled(:, :, end) *
      decoupled_states(:, k);
551   decoupled_states(:, k + 1) = A * decoupled_states(:, k) + B *
      decoupled_inputs(:, k);
552   decoupled_outputs(:, k) = C * decoupled_states(:, k) + D *
      decoupled_inputs(:, k);
553   k = k + 1;
554 end
555
556
557 if plot_all
558   plot_two_mass('Input Decoupling IO', 'Under Input Decoupling',
      decoupled_outputs, decoupled_inputs, update_file_path);
559   plot_controller_history('Input Decoupling Controller', 'Under
      Input Decoupling', F_decoupled, -1, -1, update_file_path,
      true); %note only updates every other controller #
560 end
561
562
563 %% Compute Costs of Different Controllers

```

```

564 cost_max_k = 500;
565 %LQR
566 cost_lqr_states = zeros(num_states, cost_max_k);
567 cost_lqr_inputs = zeros(num_inputs, cost_max_k);
568 %Policy
569 cost_policy_states = zeros(num_states, cost_max_k);
570 cost_policy_inputs = zeros(num_inputs, cost_max_k);
571 %Decoupled
572 cost_decoupled_states = zeros(num_states, cost_max_k);
573 cost_decoupled_inputs = zeros(num_inputs, cost_max_k);
574
575 %ICs
576 cost_decoupled_states(:, 1) = x0;
577 cost_policy_states(:, 1) = x0;
578 cost_lqr_states(:, 1) = x0;
579
580 %Cost values
581 lqr_cost = 0;
582 policy_cost = 0;
583 decoupled_cost = 0;
584
585 %Simulate
586 for k = 1:cost_max_k
587     %LQR
588     cost_lqr_inputs(:, k) = F_lqr * cost_lqr_states(:, k);
589     cost_lqr_states(:, k + 1) = A * cost_lqr_states(:, k) + B *
590         cost_lqr_inputs(:, k);
591     lqr_cost = lqr_cost + (cost_lqr_inputs(:, k)' * R *
592         cost_lqr_inputs(:, k) + cost_lqr_states(:, k)' * Q *

```

```

cost_lqr_states(:, k));

591 %Policy
592 cost_policy_inputs(:, k) = F_policy(:, :, end) *
593 cost_policy_states(:, k);
594 cost_policy_states(:, k + 1) = A * cost_policy_states(:, k) +
595 B * cost_policy_inputs(:, k);
596 policy_cost = policy_cost + (cost_policy_inputs(:, k)')' * R *
597 cost_policy_inputs(:, k) + cost_policy_states(:, k)' * Q *
598 cost_policy_states(:, k));
599 %Decoupled
600 cost_decoupled_inputs(:, k) = F_decoupled(:, :, end) *
601 cost_decoupled_states(:, k);
602 cost_decoupled_states(:, k + 1) = A * cost_decoupled_states(:, k) +
603 B * cost_decoupled_inputs(:, k);
604 decoupled_cost = decoupled_cost + (cost_decoupled_inputs(:, k)')' * R *
605 cost_decoupled_inputs(:, k) +
606 cost_decoupled_states(:, k)' * Q * cost_decoupled_states(:, k));
607
608 end
609
610 %% Compare LQR, Policy, and Input Decoupled Controller
611 lqr_cost, policy_cost, decoupled_cost %all controllers should
612 have same cost
613 F_lqr
614 F_policy(:, :, end)
615 F_decoupled(:, :, end)
616 norm(F_lqr - F_policy(:, :, end))/numel(F_lqr)
617 norm(F_lqr - F_decoupled(:, :, end))/numel(F_lqr)

```

Listing B.1: Background Intro Code

Appendix C

Matlab Simulations and Examples

C.1 Github

All code, images, and final content can be found at the repository https://github.com/NoahDunleavy/thesis_rl_on_ilc.git

C.2 Background

```
1 %A General Intro to State Space and Modern Control Theory
2 %Noah Dunleavy
3 %Honors Thesis under direction of Professor Minh Q. Phan
4 %Thayer School of Engineering
5 clc; clear;
6 addpath('Saved Data', 'Functions');
7 setDefaultFigProp();
8 plot_all = false; %most run time goes into generating figs
9 update_file_path = -1;

    %'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis
    Images\General Intro'; %set this to the save location if want
    to update figures, or -1 if not
10 if update_file_path ~= -1
11     keyboard %ensure we have to be very concious about ever
        updating all the images
12 end
13
14 %% Description
15 %
16 This intro code provides the basics of 'known' status in the
    field. To those familiar with the Thayer School of
    Engineering's curriculum, it is a recap of ENGS145 - Modern
    Control Theory.
17
18 We begin with system formulation and representation in the matrix
    form, and how we resolve the difference between the
    continuous nature of the world, and the discrete ability of
```

computers. Here, we use the '**Zero-Order-Hold**' approach. Next the idea of pole placement is introduced, and it is demonstrated how the further from the origin the poles are, the longer control takes. A deadbeat controller is used to highlight this, **which** is the time-optimal solution **for any** system.

19

20 One will note that the deadbeat controller, **while** time optimal, requires significant control effort that may not be realistic or safe **for a real** system. That leads us to our introduction of the Linear Quadratic Regulation (LQR) controller, **which** minimizes a cost **function** composed of both system inputs and states. We show the difference in responses when **input** has two different relative strengths when compared to the state.

21

22 Next we introduce the Iterative Learning Control (ILC) problem and a controller, and show that it can learn to generate **any** output (so long as permitted by the physical characteristics of the system).

23

24 All of the previous examples rely on perfect knowledge not typically possible in the **real** world (although we can approximate and identify with System Identification Methods), the process of Reinforcement Learning (RL) is shown via the Policy Iteration method. It can be, and is, shown to **find** the LQR controller as defined by its cost **function**.

25

26 Finally, the idea of basis functions is introduced briefly to lay the

```

27 groundwork for subsequent explorations.

28 %}

29

30 %% System Creation

31

32 %Scalar Representation

33 m1 = 1; %Mass of the block, [kg]

34 m2 = 0.5;

35 num_masses = 2;

36 k1 = 100; %Spring constant, [N/m]

37 k2 = 200;

38 c1 = 1; %Damping Coeffeient, [Ns/m]

39 c2 = 0.5;

40

41 %Matrix Formulations

42 M = diag([m1, m2]); %mass matrix

43 K = [k1 + k2, -k2; -k2, k2]; %stiffness matrix

44 C_damp = [c1 + c2, -c2; -c2, c2]; %Damping matrix

45

46 %Continuous Statespace Formulations

47 Ac = [zeros(num_masses), eye(num_masses); -M^-1 * K, -M^-1 * C_damp];

48 Bc = [zeros(num_masses); M^-1];

49 C = [1, 0, 0, 0; 0, 1, 0, 0]; %Monitor the block positions as outputs

50 D = [0, 0; 0, 0]; %

51

52 %Dimensional Variables

53 num_states = height(Ac); %Ac is n x n

```

```

54 num_inputs = width(Bc); %Bc is n x r
55 num_outputs = height(C); %C is m x n
56
57 %Ensure against Nyquist
58 eigens = eig(Ac); %Ac captures the modes of a given system
59 natural_freqs = imag(eigens)/(2*pi);%Get all the natural
   frequencies in Hz (convert from rad/sec)
60 max_freq = max(natural_freqs); %The highest frequency is the one
   we must cater to
61 nyquist_freq = 2 * max_freq; %Minimum sampling frequency to avoid
   nyquist sampling
62 nyquist_rate = 1 / nyquist_freq; %Sample frequency to period
63
64 dt = 0.01; %Set the sampling rate we will be using
65 if (dt > nyquist_rate) %Ensure that the true sampling period is
   below the nyquist specifications
66   dt = nyquist_rate / 10; %Set the dt to one that is sufficient,
   best practice is to scale by a factor of 2-10 beyond nyquist
67   sprintf('Selected sample rate was insufficient, setting dt =
   %.2e seconds', dt)
68 end
69
70 %Check Controllability
71 if ~is_controllable(Ac, Bc) %always check it is even possible to
   control
72   printf('The Continuous System is not Controllable!')
73 end
74
75 %Convert to Discrete

```

```

76 [A, B] = c2d(Ac, Bc, dt); %Performs the necessary discretization
    operations - verify this for yourself
77
78 %Set initial conditions
79 initial_displacement = [1; 0]; %displace the cooresponding blocks
    initially by this amount, [meters]
80 initial_velocity = [0; 2]; %Start cooresponding blocks with this
    velocity [m/s]
81 x0 = [initial_displacement; initial_velocity]; %The top half sets
    position ICs, and the bottom half sets velocity
82
83 %Simulation Duartion
84 max_samples = 500; %set the maximum number of steps out to
    simulate (note: max k will be -1, since we start at k=0)
85 max_time = max_samples * dt; %Important distinction: sample STEPS
    is not the same as time
86
87 save('Saved Data\thesis_system.mat', 'A', 'B', 'C', 'D', 'x0',
    'num_states', 'num_outputs', 'num_inputs'); %save the system
    for consistency across all following code
88 %% Open-Loop Simulation
89 cont_vs_disc_resolution = 1; %Additional resolution to give
    continuous simulations for smoother plots, and so the evident
    differences can be seen between continious and discrete
90 %Setting to 1 speeds up rendering process, and can just use
    matlab's
91 '%plot' vs stairs to highlight
92

```

```

93 open_time_continuous = linspace(0, max_time, max_samples *
94                                         cont_vs_disc_resolution)'; %Generate the time scale at the
95                                         higher resolution
96 continuous_system = ss(Ac, Bc, C, D); %Turn the continuous
97                                         matrixies into a system for simulation
98 continuous_open_outputs = lsim(continuous_system,
99                                         zeros(height(open_time_continuous), num_inputs),
100                                        open_time_continuous, x0);
101 [discrete_open_outputs, discrete_open_states] = dlsim(A, B, C, D,
102                                         zeros(max_samples, num_inputs), x0); %Generate x(0) ->
103                                         x(max_samples - 1), and cooresponding outputs (y)
104
105 %% Render Open Loop Responses
106 if plot_all
107     %Block 1 Position
108     fig = figure('Name', 'Open-loop Mass 1 Position');
109     plot(open_time_continuous, continuous_open_outputs(:, 1));
110
111     title('Mass 1 Position');
112     subtitle('Open-Loop', 'FontSize', getappdata(groot,
113                                         'DefaultFontSize'));
114     xlabel('Time (sec)') %only point where we will be using time
115                                         as the x axis
116     ylabel('Position (m)')
117     save_figure(update_file_path, fig, 'Continuous Open-Loop -
118                                         Mass 1'); %save just the continuous Model
119
120     %Add in discrete element
121     hold on;

```

```

111    stairs(dt * (0:(max_samples-1)), discrete_open_outputs(:, 1));

    %note the dt scaling such that axeses are the same

112    hold off;

113    legend('Continuous', 'Discrete')

114    save_figure(update_file_path, fig);

115

116    %Mass 2 position

117    fig = figure('Name', 'Open-loop Mass 2 Position');

118    plot(open_time_continuous, continuous_open_outputs(:, 2));

119    title('Mass 2 Position');

120    subtitle('Open-Loop', 'FontSize', getappdata(groot,
121        'DefaultSubtitleFontSize'));

121    xlabel('Time (sec)')

122    ylabel('Position (m)')

123    save_figure(update_file_path, fig, 'Continuous Open-Loop -
124        Mass 2'); %save just the continuous Model

125    %Add in discrete element

126    hold on;

127    stairs(dt * (0:(max_samples-1)), discrete_open_outputs(:, 2));

128    hold off;

129    legend('Continuous', 'Discrete')

130    save_figure(update_file_path, fig);

131

132    %Zoomed in to show the models match

133    num_zoomed_in = 11;

134    fig = figure('Name', 'Zoomed-in Open-loop Mass 1 Position');

135    plot(open_time_continuous(1:num_zoomed_in),
continuous_open_outputs(1:num_zoomed_in, 1));
    hold on;

```

```

136 stairs(dt * (0:(num_zoomed_in-1)),
137 discrete_open_outputs(1:num_zoomed_in, 1)); %note the dt
scaling such that axes are the same
138 scatter(dt * (0:(num_zoomed_in-1)),
139 discrete_open_outputs(1:num_zoomed_in, 1), 'filled', '*',
'MarkerFaceColor', 'red', 'MarkerEdgeColor', 'red',
'SizeData', 90);
140 hold off;
141 legend('Continuous', 'Discrete')
142 title('Mass 1 Position');
143 subtitle('Open-Loop', 'FontSize', getappdata(groot,
'DefaultSubtitleFontSize'));
144 xlabel('Time (sec)') %only point where we will be using time
as the x axis
145 ylabel('Position (m)')
146
147 fig = figure('Name', 'Zoomed-in Open-loop Mass 2 Position');
148 plot(open_time_continuous(1:num_zoomed_in),
continuous_open_outputs(1:num_zoomed_in, 2));
149 hold on;
150 stairs(dt * (0:(num_zoomed_in-1)),
discrete_open_outputs(1:num_zoomed_in, 2)); %note the dt
scaling such that axes are the same
151 scatter(dt * (0:(num_zoomed_in-1)),
discrete_open_outputs(1:num_zoomed_in, 2), 'filled', '*',
'MarkerFaceColor', 'red', 'MarkerEdgeColor', 'red',
'SizeData', 90);

```

```

152 hold off;
153 legend('Continuous', 'Discrete')
154 title('Mass 2 Position');
155 subtitle('Open-Loop', 'FontSize', getappdata(groot,
156 'DefaultSubtitleFontSize'));
157 xlabel('Time (sec)') %only point where we will be using time
as the x axis
158 ylabel('Position (m)')
159 xlim([0, dt*(num_zoomed_in-1)])
160 save_figure(update_file_path, fig);
161 end
162 %% Simple Pole Placement Controller
163 pole_locations = [0.5 + 0.5i, 0.5 - 0.5i, -0.7 + 0.1i, -0.7 -
0.1i]; %if imaginary, must be complex conjugates of eachother
164 placed_controller = -place(A, B, pole_locations);
165 verify_poles = eig(A + B * placed_controller);
166 if any(abs(verify_poles) >= 1)
167 printf('A Pole was placed outside of the unit circle, unstable
controller!')
168 end
169
170 %Visualize the poles
171 if plot_all
172 plot_pole_placement('Simple Pole Placement', 'Simple Pole
Placement', verify_poles, pole_locations, update_file_path);
173 end
174 closed_pole_samples = 60;
175 pole_states = zeros(num_states, closed_pole_samples);

```

```

176 pole_inputs = zeros(num_inputs, closed_pole_samples);
177 pole_outputs = zeros(num_outputs, closed_pole_samples);
178 pole_states(:, 1) = x0; %set the initial conditions
179
180 for k = 1:closed_pole_samples
181     pole_inputs(:, k) = placed_controller * pole_states(:, k);
182     %u(k) = F * x(k) - linear control law
183     pole_states(:, k + 1) = A * pole_states(:, k) + B *
184         pole_inputs(:, k); %x(k+1) = A*x(k) + B*u(k)
185     pole_outputs(:, k) = C * pole_states(:, k) + D *
186         pole_inputs(:, k); %y(k) = C*x(k) + D*u(k)
187 end
188
189 if plot_all
190     plot_two_mass('Pole Placement', 'Closed-Loop System with Pole
191     Placement', pole_outputs, pole_inputs, update_file_path);
192     %function to plot positions and forces
193 end
194
195 %% Deadbeat Controller
196 deadbeat_poles = [-0.00001, 0.00001, 0.00000i, -0.00000i]; %place
197     will not allow for multiple poles at same location, so make
198     all really close to 0
199 deadbeat_controller = -place(A, B, deadbeat_poles);
200 verify_poles = eig(A + B * deadbeat_controller);
201 if any(abs(verify_poles) > 1)
202     printf('A Pole was placed outside of the unit circle, unstable
203         controller!')
204 end

```

```

197
198 %Visualize the poles
199 if plot_all
200     plot_pole_placement('Deadbeat Pole Placement', 'Deadbeat Pole
201         Placement', verify_poles, deadbeat_poles, update_file_path);
202 end
203 deadbeat_samples = 20;
204 deadbeat_states = zeros(num_states, deadbeat_samples);
205 deadbeat_inputs = zeros(num_inputs, deadbeat_samples);
206 deadbeat_outputs = zeros(num_outputs, deadbeat_samples);
207 deadbeat_states(:, 1) = x0; %set the initial conditions
208
209 for k = 1:deadbeat_samples
210     deadbeat_inputs(:, k) = deadbeat_controller *
211         deadbeat_states(:, k); %u(k) = F * x(k) - linear control law
212     deadbeat_states(:, k + 1) = A * deadbeat_states(:, k) + B *
213         deadbeat_inputs(:, k); %x(k+1) = A*x(k) + B*u(k)
214     deadbeat_outputs(:, k) = C * deadbeat_states(:, k) + D *
215         deadbeat_inputs(:, k); %y(k) = C*x(k) + D*u(k)
216 end
217
218 if plot_all
219     plot_two_mass('Deadbeat Controller', 'Deadbeat Controller',
220         deadbeat_outputs, deadbeat_inputs, update_file_path);
221 end
222
223 %% LQR Controller
224 Q = 100 * eye(num_states); %cost of each states being away from 0
225 R = 1 * eye(num_inputs); %cost od each input away from 0

```

```

221 gamma = 0.8; %learning factor
222
223 %Construct an LQR controller
224 %The LQR controller minimizes the cost function  $J = u' * R * u +$ 
225 % $x' * x$ 
226 F_lqr = discounted_LQR(A, B, gamma, Q, R);
227
228 lqr_samples = 200;
229 lqr_states = zeros(num_states, lqr_samples);
230 lqr_inputs = zeros(num_inputs, lqr_samples);
231 lqr_outputs = zeros(num_outputs, lqr_samples);
232 lqr_states(:, 1) = x0; %set the initial conditions
233
234 for k = 1:lqr_samples
235     lqr_inputs(:, k) = F_lqr * lqr_states(:, k); %u(k) = F * x(k)
236         - linear control law
237     lqr_states(:, k + 1) = A * lqr_states(:, k) + B *
238         lqr_inputs(:, k); %x(k+1) = A*x(k) + B*u(k)
239     lqr_outputs(:, k) = C * lqr_states(:, k) + D * lqr_inputs(:, k);
240         %y(k) = C*x(k) + D*u(k)
241 end
242 %Find where it placed the poles
243 lqr_poles = eig(A + B * F_lqr);
244 if plot_all
245     plot_pole_placement('Big Q LQR Pole Locations', sprintf('Q/R =
246         %.d Pole Placement', Q(1, 1)/R(1, 1)), lqr_poles, -1,
247         update_file_path);
248 end

```

```

244 %Plot full system out
245 if plot_all
246     plot_two_mass('Big Q LQR Controller', sprintf('With LQR
247         Controller (Q/R=%d)', Q(1, 1)/R(1, 1)), lqr_outputs,
248         lqr_inputs, update_file_path);
249 end
250
251 %% Demo Higher R LQR
252 R_big = 10 * eye(num_inputs); %make relative weightings more
253 % skewed to input
254 gamma = 0.8;
255
256 big_R_F_lqr = discounted_LQR(A, B, gamma, Q, R_big);
257
258 big_R_lqr_samples = 1200;
259 big_R_lqr_states = zeros(num_states, big_R_lqr_samples);
260 big_R_lqr_inputs = zeros(num_inputs, big_R_lqr_samples);
261 big_R_lqr_outputs = zeros(num_outputs, big_R_lqr_samples);
262 big_R_lqr_states(:, 1) = x0; %set the initial conditions
263
264 for k = 1:big_R_lqr_samples
265     big_R_lqr_inputs(:, k) = big_R_F_lqr * big_R_lqr_states(:, k);
266     %u(k) = F * x(k) - linear control law
267     big_R_lqr_states(:, k + 1) = A * big_R_lqr_states(:, k) + B *
268         big_R_lqr_inputs(:, k); %x(k+1) = A*x(k) + B*u(k)
269     big_R_lqr_outputs(:, k) = C * big_R_lqr_states(:, k) + D *
270         big_R_lqr_inputs(:, k); %y(k) = C*x(k) + D*u(k)
271 end

```

```

267 %Find where it placed the poles
268 big_R_lqr_poles = eig(A + B * big_R_F_lqr);
269 if plot_all
270     plot_pole_placement('Big R Pole Placement', sprintf('Q/R = %.d
271 Pole Placement', Q(1, 1)/R_big(1, 1)), big_R_lqr_poles, -1,
272 update_file_path);
273 end
274 %Plot full system out
275 if plot_all
276     plot_two_mass('Big R LQR Controller', sprintf('With LQR
277 Controller (Q/R = %.d)', Q(1, 1)/R_big(1, 1)),
278 big_R_lqr_outputs, big_R_lqr_inputs, update_file_path);
279 end
280
281 %% ILC Introduction
282 %Setup ILC Parameters
283 p = 100; %number of steps to a process
284 num_ilc_states = num_outputs * p; %effective states for ILC are
285 %the errors, generated from outputs
286 num_ilc_inputs = num_inputs * p; %r inputs for every time step
287
288 x0_ilc = x0;
289
290 %Descriptive Matrix
291 [P_full, d_full] = P_from_ABCD(A, B, C, D, p, x0_ilc); %generate
292 %a descriptive matrix which captures the relation of inputs to
293 %outputs
294
295 %ILC State-space:

```

```

289 % e(j+1) = e(j) - P*del_u(j)
290 % y(j) = e(j)
291 A_ilc = eye(num_ilc_states);
292 B_ilc = -P_full;
293 C_ilc = eye(num_ilc_states);
294 D_ilc = 0;
295
296 if ~is_controllable(A_ilc, B_ilc) %always check it is even
    possible to control
    printf('The ILC System is not Controllable!')
298 end
299
300 F_ilc = 0.8 * pinv(P_full); %Note: any controller that places
    poles inside the unit circle works.
301
302 %Set the Goal Outputs (only have goals for p steps, regardless of
    lead)
303 %Use function draw_to_XY(p) if you would like to set your own
304 %drawing/special output. This is not written to be super robust,
    choose
305 %matching p values
306 %[drawn_x, drawn_y] = draw_to_XY(p);
307 %save(sprintf('Saved Data\shape_p%.d.mat',p), 'drawn_x',
    'drawn_y'); %save to a file so we only have to do this once
308
309 y_star_x = cos(2 * pi * (0:(p-1)) / p)';
310 y_star_y = sin(2 * pi * (0:(p-1)) / p)';
311 goal_matrix = [y_star_x, y_star_y]; %stack inputs next to
    eachother

```

```

312 y_star = reshape(goal_matrix', [], 1); %combine the seperate
     goals of each output into one vertical vector, alternating as
     necessary
313
314 %ILC Structure
315 num_trials = 10; %number of trials to learn input
316 ILC_Trial(num_trials).output = [];
317 ILC_Trial(num_trials).input = [];
318 ILC_Trial(num_trials).del_u = []; %relevant control parameter
319 ILC_Trial(num_trials).output_error = []; %relevant control
     parameter
320
321 %First Trial
322 trial_num = 1;
323 ILC_Trial(trial_num).del_u = zeros(num_ilc_inputs, 1); %first
     trial we set whatever del_u we want
324 ILC_Trial(trial_num).input = zeros(num_ilc_inputs, 1);
     %Similarly, first input is arbitrary
325
326 ILC_Trial(trial_num).output = [C * x0_ilc + D *
     ILC_Trial(trial_num).input(1:num_inputs); P_full *
     ILC_Trial(trial_num).input + d_full]; %simulate y(0) -> y(p +
     num_lead), since y = Pu + d ignores y(0), use IC
327 relevant_output = ILC_Trial(trial_num).output((num_outputs +
     1):end); %only compare to the ones we can / want to control
328 ILC_Trial(trial_num).output_error = y_star - relevant_output;
329
330 %Subsequent Iterations
331 for trial_num = 2:num_trials

```

```

332 %Inputs
333 ILC_Trial(trial_num).del_u = F_ilc * ILC_Trial(trial_num -
334 1).output_error;
335 ILC_Trial(trial_num).input = ILC_Trial(trial_num).del_u +
336 ILC_Trial(trial_num - 1).input; %d_u(j) = u(j) - u(j-1)
337 %Output
338 ILC_Trial(trial_num).output = [C * x0_ilc + D *
339 ILC_Trial(trial_num).input(1:num_inputs); P_full *
340 ILC_Trial(trial_num).input + d_full];
341 relevant_output = ILC_Trial(trial_num).output((1 +
342 num_outputs):end);
343 %Error
344 ILC_Trial(trial_num).output_error = y_star - relevant_output;
345 end
346
347 if plot_all
348 to_plot_ilc = [1, 2, 5, num_trials];
349 plot_dual_ilc('Placed Controller ILC', 'ILC with a Placed
350 Controller', ILC_Trial, y_star, -1, to_plot_ilc,
351 update_file_path); %update_file_path
352 end
353
354 %% Arbitrary ILC
355 %Setup ILC Parameters
356 if false
357 clear 'ILC_Trial' %make sure we do not carry over any ILC from
358 previous
359 p = 500; %number of steps to a process

```

```

352 num_ilc_states = num_outputs * p; %effective states for ILC
353 %are the errors, generated from outputs
354 num_ilc_inputs = num_inputs * p; %r inputs for every time step
355
356 x0_ilc = x0;
357
358 %Descriptive Matrix
359 [P_full, d_full] = P_from_ABCD(A, B, C, D, p, x0_ilc);
360 %generate a descriptive matrix which captures the relation of
361 %inputs to outputs
362
363 A_ilc = eye(num_ilc_states);
364 B_ilc = -P_full;
365 C_ilc = eye(num_ilc_states);
366 D_ilc = 0;
367
368 if ~is_controllable(A_ilc, B_ilc) %always check it is even
369 %possible to control
370 %printf('The ILC System is not Controllable!')
371 end
372
373 F_ilc = 0.8 * pinv(P_full); %Note: any controller that places
374 %poles inside the unit circle works.
375
376 loadedShape = load('Saved Data\dartmouth_p500.mat'); %read in
377 %the file
378 y_star_x = loadedShape.drawn_x';
379 y_star_y = loadedShape.drawn_y';
380 scale = 10;

```

```

375 goal_matrix = [y_star_x, y_star_y]; %stack inputs next to
eachother
376 y_star = scale * reshape(goal_matrix', [], 1); %combine the
seperate goals of each output into one vertical vector,
alternating as necessary
377
378 %ILC Structure
379 num_trials = 10; %number of trials to learn input
380 ILC_Trial(num_trials).output = [];
381 ILC_Trial(num_trials).input = [];
382 ILC_Trial(num_trials).del_u = []; %relevant control parameter
383 ILC_Trial(num_trials).output_error = []; %relevant control
parameter
384
385 %First Trial
386 trial_num = 1;
387 ILC_Trial(trial_num).del_u = zeros(num_ilc_inputs, 1); %first
trial we set whatever del_u we want
388 ILC_Trial(trial_num).input = zeros(num_ilc_inputs, 1);
%Similarly, first input is arbitrary
389
390 ILC_Trial(trial_num).output = [C * x0_ilc + D *
ILC_Trial(trial_num).input(1:num_inputs); P_full *
ILC_Trial(trial_num).input + d_full]; %simulate y(0) -> y(p +
num_lead), since y = Pu + d ignores y(0), use IC
391 relevant_output = ILC_Trial(trial_num).output((num_outputs +
1):end); %only compare to the ones we can / want to control
392 ILC_Trial(trial_num).output_error = y_star - relevant_output;
393

```

```

394 %Subsequent Iterations
395 for trial_num = 2:num_trials
396     %Inputs
397     ILC_Trial(trial_num).del_u = F_ilc * ILC_Trial(trial_num -
398         1).output_error;
399     ILC_Trial(trial_num).input = ILC_Trial(trial_num).del_u +
400         ILC_Trial(trial_num - 1).input; %d_u(j) = u(j) - u(j-1)
401     %Output
402     ILC_Trial(trial_num).output = [C * x0_ilc + D *
403         ILC_Trial(trial_num).input(1:num_inputs); P_full *
404         ILC_Trial(trial_num).input + d_full];
405     relevant_output = ILC_Trial(trial_num).output((1 +
406         num_outputs):end);
407     %Error
408     ILC_Trial(trial_num).output_error = y_star -
409         relevant_output;
410 end
411
412 if plot_all
413     to_plot_ilc = [1, 2, 5, num_trials];
414
415     resave_dartmouth = update_file_path
416     %Super explicitly re-save the dartmouth plot. When doing
417     so, go into
418         %save_figure and adjust legend location for ONLY this one.
419         Just
420         %because where it draws things

```

```

414 plot_dual_ilc('Dartmouth ILC', 'Arbitrary ILC with a Placed
415 Controller', ILC_Trial, y_star, -1, to_plot_ilc,
416 resave_dartmouth); %update_file_path
417 end
418 else
419 sprintf('Not doing the big ILC to save computation time')
420 end
421
422 %% Reinforcement Learning - Policy Iteration
423 converged_k = 20; %number of steps to run out for with
424 %'converged' controller
425 num_controllers = 5; %number of iterations of each controller
426 %before updating
427 Pj_dim = (num_states + num_inputs); %minimum dimensions of the P
428 %matrix such that we can solve (noise free)
429 num_collections = Pj_dim^2; %number of data collections to make
430 %before solving the inv (noise free)
431
432 total_tries = num_controllers * num_collections + converged_k;
433
434 F_policy = zeros(num_inputs, num_states, 1); %start with no
435 %controller
436
437 policy_states = zeros(num_states, total_tries);
438 policy_states(:, 1) = x0;
439 policy_inputs = zeros(num_inputs, total_tries);
440 policy_outputs = zeros(num_outputs, total_tries);
441 k = 1;
442
```

```

436
437 for iteration = 1:num_controllers
438     %Reset Stacks each iteration
439     Uk_stack = zeros(num_collections, 1);
440     Xk_stack = zeros(num_collections, (Pj_dim)^2);
441
442     %Compute stack infos
443     for collection = 1:num_collections
444         %Simulate nature
445         %policy_inputs(:, k) = rand(num_inputs, 1); %can replicate
446         %the 'random state' approach by just randomizing inputs
447         policy_inputs(:, k) = F_policy(:, :, end) *
448         policy_states(:, k) + rand_range(num_inputs, 1, -1, 1);
449         %compute input and exploration term
450
451         %Next Step regardless of nature vs random
452         policy_states(:, k + 1) = A * policy_states(:, k) + B *
453         policy_inputs(:, k);
454         policy_outputs(:, k) = C * policy_states(:, k) + D *
455         policy_inputs(:, k);
456
457
458     %Construct stacks
459     xu_stack = [policy_states(:, k); policy_inputs(:, k)];
460     xu_next_stack = [policy_states(:, k + 1); F_policy(:, :, end) *
461     policy_states(:, k + 1)];
462     Xk_stack(collection, :) = kron(xu_stack', xu_stack') -
463     gamma * kron(xu_next_stack', xu_next_stack');

```

```

457     Uk_stack(collection, :) = policy_states(:, k)' * Q *
458         policy_states(:, k) + policy_inputs(:, k)' * R *
459         policy_inputs(:, k); %where Q and R come into play
460
461     %Move to next
462     k = k + 1;
463
464 end
465
466 % [P, S, V] = svd(Xk_stack);
467
468 % rank(S), iteration %we do not need the stack to be full
469 %rank, but it
470
471 % must not drop ranks thorough iterations
472
473 %Calculate P and new controller
474
475 PjS = pinv(Xk_stack) * Uk_stack;
476
477 Pj = reshape(PjS, Pj_dim, Pj_dim); %undo the stack
478
479 Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
480 %reduces error)
481
482 Pjuu = Pj((num_states+1):end, (num_states+1):end); %grab the
483 bottom right quadrant, Puu
484
485 PjxuT = Pj((num_states+1):end, 1:num_states); %Grab bottom
486 left, Pxu transpose
487
488 new_F = -pinv(Pjuu) * PjxuT; %definition of controller
489
490 F_policy(:, :, end + 1) = new_F; %add to end of list
491
492 end
493
494
495 %Plot / simulate past convergance (without exploration/noise) so
496 %actual
497 %control possible
498
499 for ndx = 1:converged_k
500
501     %Simulate nature

```

```

479 policy_inputs(:, k) = F_policy(:, :, end) * policy_states(:, k);
480 policy_states(:, k + 1) = A * policy_states(:, k) + B *
481 policy_inputs(:, k);
482 policy_outputs(:, k) = C * policy_states(:, k) + D *
483 policy_inputs(:, k);
484 k = k + 1;
485 end
486
487 if plot_all
488 plot_two_mass('Policy Iteration IO', 'Under Policy Iteration',
489 policy_outputs, policy_inputs, update_file_path);
490 plot_controller_history('Policy Iteration Controller', 'Under
491 Policy Iteration', F_policy, -1, -1, update_file_path); %1s
492 % denote to plot every state and input weight
493 end
494
495
496
497 %% Reinforcement Learning - Input Decoupling
498 converged_k = 20;
499 num_controllers = 5;
500 Pj_dim = (num_states + 1); %effective input count is 1 now
501 num_collections = Pj_dim^2;
502
503 total_tries = num_controllers * num_collections + converged_k;
504
505 F_decoupled = zeros(num_inputs, num_states, 1); %start with no
506 controller

```

```

501 decoupled_states = zeros(num_states, total_tries);
502 decoupled_states(:, 1) = x0;
503 decoupled_inputs = zeros(num_inputs, total_tries);
504 decoupled_outputs = zeros(num_outputs, total_tries);
505 k = 1;
506
507 for iteration = 1:num_controllers
508     for input_num = 1:num_inputs
509         %Reset Stacks each iteration
510         Uk_stack = zeros(num_collections, 1);
511         Xk_stack = zeros(num_collections, (Pj_dim)^2);
512         F_i = F_decoupled(input_num, :, end); %input on the current
513         %input of inspection
514         %Compute stack infos
515         for collection = 1:num_collections
516             %Simulate nature
517             %decoupled_inputs(:, k) = rand(num_inputs, 1); %can
518             %replicate the 'random state' approach by just randomizing
519             %inputs
520             decoupled_inputs(:, k) = F_decoupled(:, :, end) *
521             decoupled_states(:, k); %compute input
522             decoupled_inputs(input_num, k) = rand_range(1, 1, -1,
523             1); %+decoupled_inputs(input_num, k); %
524
525             %Next Step regardless of nature vs random
526             decoupled_states(:, k + 1) = A * decoupled_states(:, k)
527             + B * decoupled_inputs(:, k);
528             decoupled_outputs(:, k) = C * decoupled_states(:, k) + D
529             * decoupled_inputs(:, k);

```

```

523
524
525     %Construct stacks
526
527     xu_stack = [decoupled_states(:, k);
528
529         decoupled_inputs(input_num, k)];
530
531     xu_next_stack = [decoupled_states(:, k + 1); F_i *
532
533         decoupled_states(:, k + 1)];
534
535     Xk_stack(collection, :) = kron(xu_stack', xu_stack') -
536
537     gamma * kron(xu_next_stack', xu_next_stack');
538
539     Uk_stack(collection, :) = decoupled_states(:, k)' * Q *
540
541     decoupled_states(:, k) + decoupled_inputs(:, k)' * R *
542
543     decoupled_inputs(:, k); %where Q and R come into play
544
545
546     %Move to next
547
548     k = k + 1;
549
550 end
551
552
553 %Calculate P and new controller
554
555 PjS = pinv(Xk_stack) * Uk_stack;
556
557 Pj = reshape(PjS, Pj_dim, Pj_dim); %undo the stack
558
559 Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
560
561 reduces error)
562
563 Pjuu = Pj((num_states+1):end, (num_states+1):end); %grab
564
565 the bottom right quadrant, Puu
566
567 PjxuT = Pj((num_states+1):end, 1:num_states); %Grab bottom
568
569 left, Pxu transpose
570
571 new_F_i = -pinv(Pjuu) * PjxuT; %definition of controller
572
573 F_decoupled(:, :, end + 1) = F_decoupled(:, :, end); %copy
574
575 old controller

```

```

543     F_decoupled(input_num, :, end) = new_F_i; %update current
      input
544   end
545 end
546
547 %Plot / simulate past convergance (without exploration/noise) so
      actual
548 %control possible
549 for ndx = 1:converged_k
      %Simulate nature
550   decoupled_inputs(:, k) = F_decoupled(:, :, end) *
      decoupled_states(:, k);
551   decoupled_states(:, k + 1) = A * decoupled_states(:, k) + B *
      decoupled_inputs(:, k);
552   decoupled_outputs(:, k) = C * decoupled_states(:, k) + D *
      decoupled_inputs(:, k);
553   k = k + 1;
554 end
555
556
557 if plot_all
558   plot_two_mass('Input Decoupling IO', 'Under Input Decoupling',
      decoupled_outputs, decoupled_inputs, update_file_path);
559   plot_controller_history('Input Decoupling Controller', 'Under
      Input Decoupling', F_decoupled, -1, -1, update_file_path,
      true); %note only updates every other controller #
560 end
561
562
563 %% Compute Costs of Different Controllers

```

```

564 cost_max_k = 500;
565 %LQR
566 cost_lqr_states = zeros(num_states, cost_max_k);
567 cost_lqr_inputs = zeros(num_inputs, cost_max_k);
568 %Policy
569 cost_policy_states = zeros(num_states, cost_max_k);
570 cost_policy_inputs = zeros(num_inputs, cost_max_k);
571 %Decoupled
572 cost_decoupled_states = zeros(num_states, cost_max_k);
573 cost_decoupled_inputs = zeros(num_inputs, cost_max_k);
574
575 %ICs
576 cost_decoupled_states(:, 1) = x0;
577 cost_policy_states(:, 1) = x0;
578 cost_lqr_states(:, 1) = x0;
579
580 %Cost values
581 lqr_cost = 0;
582 policy_cost = 0;
583 decoupled_cost = 0;
584
585 %Simulate
586 for k = 1:cost_max_k
587     %LQR
588     cost_lqr_inputs(:, k) = F_lqr * cost_lqr_states(:, k);
589     cost_lqr_states(:, k + 1) = A * cost_lqr_states(:, k) + B *
590         cost_lqr_inputs(:, k);
591     lqr_cost = lqr_cost + (cost_lqr_inputs(:, k)' * R *
592         cost_lqr_inputs(:, k) + cost_lqr_states(:, k)' * Q *

```

```

cost_lqr_states(:, k));
591 %Policy
592 cost_policy_inputs(:, k) = F_policy(:, :, end) *
593 cost_policy_states(:, k);
593 cost_policy_states(:, k + 1) = A * cost_policy_states(:, k) +
594 B * cost_policy_inputs(:, k);
594 policy_cost = policy_cost + (cost_policy_inputs(:, k)')' * R *
595 cost_policy_inputs(:, k) + cost_policy_states(:, k)' * Q *
595 cost_policy_states(:, k));
596 %Decoupled
596 cost_decoupled_inputs(:, k) = F_decoupled(:, :, end) *
597 cost_decoupled_states(:, k);
597 cost_decoupled_states(:, k + 1) = A * cost_decoupled_states(:, k) +
598 B * cost_decoupled_inputs(:, k);
598 decoupled_cost = decoupled_cost + (cost_decoupled_inputs(:, k))' * R *
599 cost_decoupled_inputs(:, k) +
600 cost_decoupled_states(:, k)' * Q * cost_decoupled_states(:, k));
599 end
600
601 %% Compare LQR, Policy, and Input Decoupled Controller
602 lqr_cost, policy_cost, decoupled_cost %all controllers should
603 have same cost
603 F_lqr
604 F_policy(:, :, end)
605 F_decoupled(:, :, end)
606 norm(F_lqr - F_policy(:, :, end))/numel(F_lqr)
607 norm(F_lqr - F_decoupled(:, :, end))/numel(F_lqr)

```

Listing C.1: Background Intro Code

C.3 Basis Functions

```
1 %Basis Function Requirements and Applications
2 %Noah Dunleavy
3 %Honors Thesis under direction of Professor Minh Q. Phan
4 %Thayer School of Engineering
5 clc; clear;
6 addpath('Saved Data', 'Functions');
7 setDefaultFigProp();
8 plot_all = true;
9 update_file_path =
    -1;%'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis
        Images\Basis Functions'; %set this to the save location if
        want to update figures, or -1 if not
10 if update_file_path ~= -1
11     keyboard %ensure we have to be very concious about ever
        updating all the images
12 end
13
14 %% Basics of Basis
15 max_cheby = 20; %for demo purposes, maximum number of cheby
    functions to generate
16
17 p = 100; %resolution of the cheby functions
18 cheby_x = linspace(-1, 1, p)'; %define the cheby 'x'
19 cheby_functions = ones(p, max_cheby);
20 cheby_functions(:, 2) = cheby_x;
21 for ndx = 3:max_cheby %this is the recursive cheby generation.
    Could be from a file, but more helpful to see
```

```

22 cheby_functions(:, ndx) = 2 * cheby_x .* cheby_functions(:, 
23     ndx - 1) - cheby_functions(:, ndx - 2); %build cheby out
24 end
25
26 %Plot some of the core chebys
27 core_ndx = [1, 2, 8, 20]; %use 1 as the DC offset, then jst random
28 if plot_all
29     core_cheby_fig = figure('Name', 'Example Cheby Functions');
30     stairs(1:p, cheby_functions(:, core_ndx))
31     xlabel('Chebyshev Step')
32     ylabel('Amplitude')
33     title('Example Chebyshev Functions')
34     legend({'$T_0$', '$T_1$', '$T_7$', '$T_{19}$'}, 'Interpreter',
35         'latex'); %-1 off of matlab ndxs
36     save_figure(update_file_path, core_cheby_fig);
37 end
38
39 %Plot the output signal
40 rng('default'); rng(10);
41 cheby_weights = rand_range(max_cheby, 1, -3, 3);
42 cheby_signal = cheby_functions * cheby_weights;
43 if plot_all
44     fig = figure('Name', 'Example Cheby Signal');
45     stairs(1:p, cheby_signal)
46     xlabel('Chebyshev Step')
47     ylabel('Amplitude')
48     title('Example Chebyshev Composite Signal')
49     save_figure(update_file_path, fig);
50 end

```

```

49 %Plot the Weights
50 if plot_all
51     fig = figure('Name', 'Cheby Weights');
52     temp_plot = plot(1:(max_cheby), cheby_weights, 'Marker', 'o',
53                       'MarkerFaceColor', 'auto');
54     xlabel('Coefficient Number')
55     ylabel('Weight')
56     title('Chebyshev Coefficient Weights')
57     xlim([1, max_cheby])
58     xticks(1:(max_cheby))
59     save_figure(update_file_path, fig);
60 end
61
62 %% System Creation
63 thesis_system = load('Saved Data\thesis_system.mat');
64 A = thesis_system.A;
65 B = thesis_system.B;
66 C = thesis_system.C;
67 D = thesis_system.D;
68 num_inputs = thesis_system.num_inputs;
69 num_outputs = thesis_system.num_outputs;
70 num_states = thesis_system.num_states;
71 x0 = thesis_system.x0;
72
73 %% ILC Parameters
74 num_ilc_states = p * num_outputs;
75 num_ilc_inputs = p * num_inputs;
76 [P, d] = P_from_ABCD(A, B, C, D, p, x0);

```

```

77
78 %% Chebyshev Polynomial Creation
79 max_cheby = 10;
80 cheby_functions = generate_chebyshev(num_ilc_inputs, max_cheby);
81 %In theory, there could be different resolution for inputs and
82 %outputs,
83 %but since num_inputs = num_outputs, we can use the same for both
84
85 %% General Parameters
86 %We have said p = 100
87 %Controller in basis_ilc_sim defaults to 0.5H+
88 %Built chebys out of 10 functions
89
90 %% Setup: u* in Basis Space, y* is not
91 T_u = cheby_functions; %cheby functions is only out of 10, so
92 %grab them all
93 beta_star = [1, 0.2, -0.3, 4, 0, 0, 0, -1, 0, 0]'; %manually set
94 %betas
95 u_star = T_u * beta_star; %from basis to total space
96 y_star = P * u_star + d; %from input to output
97 T_y = T_u; %equal - we do not want T_y to capture y*
98 alpha_star = pinv(T_y) * y_star; %compute what our goal alpha is
99 y_Ty_alpha = T_y * alpha_star;
100 u_star_Tu_u_star_error = norm(y_star - y_Ty_alpha); %if fully
    %captured, should be 0
101 u_star_Tu_b_star_error = norm(u_star - T_u*beta_star); %should be
    %zero since forced to be within
102
103 %% Render: u* in Basis Space, y* is not

```

```

101 if plot_all
102
103     %u1* Plot
104
105     fig = figure('Name', 'Goal Input 1');
106
107     stairs(0:(p-1), u_star(1:2:end))
108
109     title('Goal Input 1')
110
111     %subplot('Defined in $\Phi_u$')
112
113     xlabel('Step Number (k)')
114
115     ylabel('Amplitude')
116
117     save_figure(update_file_path, fig, 'Full input Partial output
118 - Goal Input 1');
119
120
121     %U2* plot
122
123     fig = figure('Name', 'Goal Input 2');
124
125     stairs(0:(p-1), u_star(2:2:end))
126
127     title('Goal Input 2')
128
129     %subplot('Defined in $\Phi_u$')
130
131     xlabel('Step Number (k)')
132
133     ylabel('Amplitude')
134
135     save_figure(update_file_path, fig, 'Full input Partial output
136 - Goal Input 2');
137
138
139     %y1* Plot
140
141     fig = figure('Name', 'Goal Output 1');
142
143     stairs(0:(p-1), y_Ty_alpha(1:2:end))
144
145     title('Goal Output 1 ')
146
147     hold on;
148
149     stairs(0:(p-1), y_star(1:2:end))
150
151     hold off;
152
153     legend({'$\Phi_y \alpha^{*1}$', '$\underline{y}_1^{*}$'},
154           'Interpreter', 'latex');

```

```

127 %subtitle('Defined in $\Phi_u$')
128 xlabel('Step Number (k)')
129 ylabel('Amplitude')
130 save_figure(update_file_path, fig, 'Full input Partial output
- Goal Output 1');

131
132 %y2* Plot
133 fig = figure('Name', 'Goal Output 2');
134 stairs(0:(p-1), y_Ty_alpha(2:2:end))
135 title('Goal Output 2')
136 hold on;
137 stairs(0:(p-1), y_star(2:2:end))
138 hold off;
139 legend({'$\Phi_y \alpha^{ast_2}$', '$\underline{y}_2^{ast}$'},
'Interpreter', 'latex');
140 %subtitle('Defined in $\Phi_u$')
141 xlabel('Step Number (k)')
142 ylabel('Amplitude')
143 save_figure(update_file_path, fig, 'Full input Partial output
- Goal Output 2');

144 end
145
146 %% Application: u* in Basis Space, y* is not
147 num_trials = 20;
148 FIPO_ILC = basis_ilc_sim(P, d, C*x0, T_u, T_y, y_star,
    num_trials);
149 norm(FIPO_ILC(end).output_error) %overall error
150 proj_u_star = T_u * pinv(T_u' * T_u) * T_u' * u_star; %project u*
    onto Tu

```

```

151 norm(proj_u_star - FIPO_ILC(end).input)
152 if plot_all
153     to_plot = [1, 2, 5, 20];
154     plot_ilc_coefficients('Full Input Partial Output', 'Full Input
155         Partial Output', FIPO_ILC, to_plot, beta_star,
156         find(beta_star)', 5, update_file_path);
157     plot_dual_ilc('Full Input Partial Output', 'Full Input Partial
158         Output', FIPO_ILC, y_star, u_star, to_plot, update_file_path);
159 end
160
161 %% Setup: y* in Basis Space, u* is not
162 %% Keep the same Tu and Ty
163 alpha_star = beta_star; %new alpha_star is the old beta_star
164 y_star = T_y * alpha_star;
165 u_star = pinv(P) * (y_star - d); %calcuuate the input that gets us
166     here
167 beta_star = pinv(T_u) * u_star; %compute what our goal alpha is
168 u_Tu_beta = T_u * beta_star;
169 u_star_Tu_u_star_error = norm(u_star - u_Tu_beta) %if fully
170     capured, should be 0
171
172 %% Render: y* in Basis Space, u* is not
173 if plot_all
174     %u1* Plot
175     fig = figure('Name', 'Goal Input 1');
176     stairs(0:(p-1), u_Tu_beta(1:2:end))
177     hold on;
178     stairs(0:(p-1), u_star(1:2:end))
179     hold off;

```

```

175 legend({'$\Phi_u \beta^{\ast_1}$', '$\underline{u}_1^{\ast}$'},
176 'Interpreter', 'latex');
177 title('Goal Input 1')
178 %subtitle('Defined in $\Phi_u$')
179 xlabel('Step Number (k)')
180 ylabel('Amplitude')
181 save_figure(update_file_path, fig, 'Partial input Full output
182 - Goal Input 1');

183 %U2* plot
184 fig = figure('Name', 'Goal Input 2');
185 stairs(0:(p-1), u_Tu_beta(2:2:end))
186 hold on;
187 stairs(0:(p-1), u_star(2:2:end))
188 hold off;
189 legend({'$\Phi_u \beta^{\ast_1}$', '$\underline{u}_1^{\ast}$'},
190 'Interpreter', 'latex');
191 title('Goal Input 2')
192 %subtitle('Defined in $\Phi_u$')
193 xlabel('Step Number (k)')
194 ylabel('Amplitude')
195 save_figure(update_file_path, fig, 'Partial input Full output
196 - Goal Input 2');

197 %y1* Plot
198 fig = figure('Name', 'Goal Output 1');
199 stairs(0:(p-1), y_star(1:2:end))
200 title('Goal Output 1')

```

```

200    %subtitle('Defined in $\Phi_u$')
201    xlabel('Step Number (k)')
202    ylabel('Amplitude')
203    save_figure(update_file_path, fig, 'Partial input Full output
204      - Goal Output 1');
205
206    %y2* Plot
207
208    fig = figure('Name', 'Goal Output 2');
209
210    stairs(0:(p-1), y_star(2:2:end))
211    title('Goal Output 2')
212    %subtitle('Defined in $\Phi_u$')
213    xlabel('Step Number (k)')
214    ylabel('Amplitude')
215    save_figure(update_file_path, fig, 'Partial input Full output
216      - Goal Output 2');
217
218    %% Application: y* in Basis Space, u* is not
219    %% Same number of trials
220
221    PIFO_ILC = basis_ilc_sim(P, d, C*x0, T_u, T_y, y_star,
222      num_trials);
223    norm(PIFO_ILC(end).output_error)
224
225    proj_u_star = T_u * pinv(T_u' * T_u) * T_u' * u_star; %project u*
226      onto Tu
227
228    norm(proj_u_star - PIFO_ILC(end).input)
229
230    if plot_all
231      to_plot = [1, 2, 5, 20]; %same to_plot

```

```

224 plot_ilc_coefficients('Partial Input Full Output', 'Partial
225     Input Full Output', PIFO_ILC, to_plot, -1, 5,
226     find(alpha_star)', update_file_path);
227
228 %% Setup: ny < nu
229 %Keep the same Tu and Ty
230 beta_star = alpha_star; %swap back the coefficients
231 u_star = T_u * beta_star;
232 y_star = P*u_star + d;
233 T_y = y_star; %set our output basis to be pmx1
234
235 %% Render: ny < nu
236 if plot_all
237     %Inputs already rendered
238
239     %y1* Plot
240     fig = figure('Name', 'Goal Output 1');
241
242     stairs(0:(p-1), y_star(1:2:end))
243     title('Goal Output 1 ')
244     %subtitle('Defined in $\Phi_u$')
245     xlabel('Step Number (k)')
246     ylabel('Amplitude')
247     save_figure(update_file_path, fig, 'Full input Single output -
248         Goal Output 1');

```

```

249 %y2* Plot
250 fig = figure('Name', 'Goal Output 2');
251
252 stairs(0:(p-1), y_star(2:2:end))
253 title('Goal Output 2')
254 %subtitle('Defined in $\Phi_u$')
255 xlabel('Step Number (k)')
256 ylabel('Amplitude')
257 save_figure(update_file_path, fig, 'Full input Single output -
    Goal Output 2');
258 end
259
260 %% Application: ny < nu
261 %Same number of trials
262 FISO_ILC = basis_ilc_sim(P, d, C*x0, T_u, T_y, y_star,
    num_trials);
263 norm(FISO_ILC(end).output_error)
264 proj_u_star = T_u * pinv(T_u' * T_u) * T_u' * u_star; %project u*
    onto Tu
265 norm(proj_u_star - FISO_ILC(end).input)
266 if plot_all
267 to_plot = [1, 2, 5, 20]; %same to_plot
268 plot_ilc_coefficients('Full Input Single Output', 'Full Input
    Single Output', FISO_ILC, to_plot, beta_star,
    find(beta_star)', 1, update_file_path);
269 plot_dual_ilc('Full Input Single Output', 'Full Input Single
    Output', FISO_ILC, y_star, u_star, to_plot, update_file_path);
270 end
271

```

```

272 %% Setup: ny > nu
273 T_y = T_u; %set the basis spaces back to both be chebys
274 alpha_star = beta_star; %swap back the coefficients
275 y_star = T_y * alpha_star;
276 u_star = pinv(P) * (y_star - d);
277 T_u = u_star; %set our output basis to be pmx1
278
279 %% Render: ny > nu
280 if plot_all
281     %U1* Plot
282     fig = figure('Name', 'Goal Input 1');
283     stairs(0:(p-1), u_star(1:2:end))
284     title('Goal Input 1')
285     %subtitle('Defined in $\Phi_u$')
286     xlabel('Step Number (k)')
287     ylabel('Amplitude')
288     save_figure(update_file_path, fig, 'Single input Full output -
289 Goal Input 1');

290     %U2* plot
291     fig = figure('Name', 'Goal Input 2');
292     stairs(0:(p-1), u_star(2:2:end))
293     title('Goal Input 2')
294     %subtitle('Defined in $\Phi_u$')
295     xlabel('Step Number (k)')
296     ylabel('Amplitude')
297     save_figure(update_file_path, fig, 'Single input Full output -
298 Goal Input 2');

```

```

299 %Outputs already rendered
300 end
301
302 %% Application: ny > nu
303 %Same number of trials
304 SIFO = basis_ilc_sim(P, d, C*x0, T_u, T_y, y_star, num_trials);
305 norm(SIFO(end).output_error)
306 proj_u_star = T_u * pinv(T_u' * T_u) * T_u' * u_star; %project u*
            onto Tu
307 norm(proj_u_star - SIFO(end).input)
308 if plot_all
309     to_plot = [1, 2, 5, 20]; %same to_plot
310     plot_ilc_coefficients('Single Input Full Output', 'Single
            Input Full Output', SIFO, to_plot, 1, 1, find(alpha_star),
            update_file_path);
311     plot_dual_ilc('Single Input Full Output', 'Single Input Full
            Output', SIFO, y_star, u_star, to_plot, update_file_path);
312 end
313
314 %% Setup: Rolling Tu
315 num_rolling_basis_input = 3; %num_basis - 1
316 num_basis_output = num_rolling_basis_input + 1; %why do anything
            worse than optimal
317 T_u_ndx = 2:(num_rolling_basis_input+1); %initilaize off a littl
318 T_u_full = cheby_functions; %grab max functions we will try
319 T_y_full = T_u_full;
320 T_u = T_u_full(:, [1, T_u_ndx]); %grab the first couple of basis
321 T_y = T_u;
322 %Alpha star assigned rolling

```

```

323 beta_star = [1, 0.2, -0.3, 4, 0, 0, 0, -1, 0, 0]'; %manually set
            betas
324 u_star = T_u_full * beta_star; %from basis to total space
325 y_star = P * u_star + d; %from input to output
326
327 num_loops = 5;%how many loops through we'll do
328 num_rolls_per_loop = max_cheby / num_rolling_basis_input;
329 num_tries = ceil(num_loops * num_rolls_per_loop);
330
331 %% Render: Rolling Tu
332 if plot_all
333     %Inputs already rendered
334
335     %Outputs already rendered
336 end
337
338 %% Application: Rolling Tu
339 %Same number of trials
340 rolling_ILC = [];
341 learned_inputs = zeros(num_ilc_inputs, num_tries + 1);
342
343 for ndx = 1:num_tries
344     %Run the ilc trial with basis spaces
345     temporary_ILC = basis_ilc_sim(P, d, C*x0, T_u, T_y, y_star,
346         num_trials); %sim out a trial
346     rolling_ILC = [rolling_ILC, temporary_ILC]; %stack it on to
            previous results
347
348     %What input did we learn

```

```

349 learned_u = temporary_IILC(end).input;
350 learned_inputs(:, ndx+1) = learned_u;
351 %Take learned input, make it a basis function
352 T_u_ndx = mod(T_u_ndx + num_rolling_basis_input - 1,
353 max_cheby) + 1; %update which new functions we'll add
353 T_u = [learned_u, T_u_full(:, T_u_ndx)];
354 %T_y doesnt matter
355 T_y = T_u;
356 end
357
358 output_error = norm(rolling_IILC(end).output_error)
359 input_error = norm(T_u(:, 1) - u_star)
360
361 if plot_all
362 to_plot = 4;
363 plot_ilc_coefficients('Rolling Input Basis', 'Rolling Input
364 Basis', rolling_IILC, to_plot, -1, num_rolling_basis_input +
1, num_rolling_basis_input + 1, update_file_path);
364 plot_dual_ilc('Rolling Input Basis', 'Rolling Input Basis',
rolling_IILC, y_star, u_star, to_plot, update_file_path);
365 end

```

Listing C.2: Basis Functions Derivation and Use

C.4 Derivation and Demonstration of Conjugate Basis Functions

```
1 %Demonstration and Determination of Conjugate Basis Functions
2 %Noah Dunleavy
3 %Honors Thesis under direction of Professor Minh Q. Phan Thayer
4 %School of
5 %Engineering
6 clc; clear;
7 addpath('Saved Data', 'Functions');
8 setDefaultFigProp();
9 plot_all = true;
10 update_file_path =
11 -1;%'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis
12 Images\Derive and Demo Conjugate Basis'; %set this to the
13 save location if want to update figures, or -1 if not
14 if update_file_path ~= -1
15 keyboard %ensure we have to be very concious about ever
16 updating all the images
17 end
18
19 %% System Creation
20 thesis_system = load('Saved Data\thesis_system.mat');
21 A = thesis_system.A;
22 B = thesis_system.B;
23 C = thesis_system.C;
24 D = thesis_system.D;
25 num_inputs = thesis_system.num_inputs;
26 num_outputs = thesis_system.num_outputs;
```

```

21 num_states = thesis_system.num_states;
22 x0 = thesis_system.x0;
23
24 %% ILC Parameters
25 p = 100;
26
27 num_ilc_states = p * num_outputs;
28 num_ilc_inputs = p * num_inputs;
29
30 [P, d] = P_from_ABCD(A, B, C, D, p, x0);
31
32 %% Chebyshev Generation
33 max_cheby = 10;
34 cheby_functions = generate_chebyshev(num_ilc_inputs, max_cheby);
35
36 %% Create u* and y*
37 beta_star = [1, 0.2, -0.3, 4, 0, 0, 0, -1, 0, 0]'; %manually set
            betas
38 u_star = cheby_functions * beta_star; %from basis to total space
39 y_star = P * u_star + d; %from input to output
40
41 %Learning Weights
42 Q = 100*eye(num_ilc_states);
43 R = 0*eye(num_ilc_inputs);
44
45 out1_ndx = (1:2:(num_ilc_states)); %convert stacked output to
            individual positions
46 out2_ndx = (2:2:(num_ilc_states));
47

```

```

48 %% Conjugate Basis Creation (Batch)
49 num_conjugate_basis = max_cheby; %how many of the cheby functions
   we will stimulate the system with / how many conjugate
   functions to make
50 batch_input = cheby_functions; %for batch input (and code
   simplicity)
51 %Batch input must be full rank to ensure wronskian exists
52
53 %Generate Batch output
54 batch_outputs_delta = P * batch_input; %do not include the d
   term, because we want the difference in outputs, which
   excludes d
55
56 W = batch_outputs_delta' * Q * batch_outputs_delta; %W matrix to
   get wronskian from
57
58 rho_batch = chol(W); %cholesky decomposition of W to get the
   optimal coeffecients for the batch
59 T_b_batch = batch_input / (rho_batch); %/ is same as * inv()
60 H_b_batch = batch_outputs_delta / (rho_batch); %
61 beta_batch = H_b_batch' * Q * (y_star - d); %determined optimal
   weights for given basis functions (off of e_0)
62
63 %Generate final output
64 batch_learned_u = T_b_batch * beta_batch;
65 batch_learned_y = P * (batch_learned_u) + d;
66
67 %Verify conjunct condition

```

```

68 batch_conj_cond = T_b_batch' * (R + P' * Q * P) * T_b_batch;
    %should be identity matrix
69
70 %% Conjugate Basis Creation (Iterative)
71 %We already defined Q and R above We have set our input sequences
    through
72 %cheby_functions
73
74 %Initial experiments
75 u0 = zeros(num_ilc_inputs, 1);
76 y0 = d;
77 u1 = cheby_functions(:, 1);
78 y1 = P*u1 + d;
79
80 %Compute del_u1 and del_y1 explicitly (P*del_u also works)
81 Episode(1).del_u = u1 - u0;
82 Episode(1).del_y = y1 - y0;
83
84 %Compute W
85 W = Episode(1).del_u' * R * Episode(1).del_u + Episode(1).del_y'
    * Q * Episode(1).del_y;
86
87 %rho, phi_1, h_1, and beta_1
88 Episode(1).rho = chol(W);
89 phi_1 = Episode(1).del_u * Episode(1).rho^-1;
90 h_1 = Episode(1).del_y * Episode(1).rho^-1;
91 beta_1 = h_1' * Q * (y_star - d); %use e0 for all
92
93 Episode(1).Phi = phi_1;

```

```

94 Episode(1).Hb = h_1;
95 Episode(1).Betas = beta_1;
96
97 %Third experiment for phi_2
98 u2 = cheby_functions(:, 2);
99 y2 = P*u2 + d;
100
101 %Define all our deltas wrt u0 and y0
102 Episode(2).del_u = u2 - u0;
103 Episode(2).del_y = y2 - y0;
104
105 %Episode(2).rho, phi_2, h_2, and beta_2
106 Episode(2).rho(1) = 1/Episode(1).rho(1) * (Episode(1).del_u' * R
107 * Episode(2).del_u + Episode(1).del_y' * Q *
108 Episode(2).del_y);
109 Episode(2).gamma = Episode(2).rho(1)^2;
110 Episode(2).rho(2) = sqrt(Episode(2).del_u' * R * Episode(2).del_u
111 + Episode(2).del_y' * Q * Episode(2).del_y - Episode(2).gamma);
112 phi_2 = (1/Episode(2).rho(2)) * (Episode(2).del_u - phi_1 *
113 Episode(2).rho(1));
114 h_2 = 1/Episode(2).rho(2) * (Episode(2).del_y - (h_1 *
115 Episode(2).rho(1)));
116 beta_2 = h_2' * Q * (y_star - d);
117
118 Episode(2).Phi = [Episode(1).Phi, phi_2];
119 Episode(2).Hb = [Episode(1).Hb, h_2];
120 Episode(2).Betas = [Episode(1).Betas; beta_2];
121
122 %Fourth experiment for phi_3

```

```

118 u3 = cheby_functions(:, 3);
119 y3 = P*u3 + d;
120
121 %Define all our deltas wrt u0 and y0
122 Episode(3).del_u = u3 - u0;
123 Episode(3).del_y = y3 - y0;
124
125 %Episode(3).rho, phi_3, h_3, and beta_3
126 Episode(3).rho(1) = 1/Episode(1).rho(1) * (Episode(1).del_u' * R
    * Episode(3).del_u + Episode(1).del_y' * Q *
    Episode(3).del_y);
127 Episode(3).rho(2) = 1/Episode(2).rho(2) * (Episode(2).del_u' * R
    * Episode(3).del_u + Episode(2).del_y' * Q * Episode(3).del_y
    - (Episode(2).rho(1)*Episode(3).rho(1)));
128 Episode(3).gamma = Episode(3).rho(1)^2 + Episode(3).rho(2)^2;
129 Episode(3).rho(3) = sqrt(Episode(3).del_u' * R * Episode(3).del_u
    + Episode(3).del_y' * Q * Episode(3).del_y - Episode(3).gamma);
130 phi_3 = (1/Episode(3).rho(3)) * (Episode(3).del_u - (phi_1 *
    Episode(3).rho(1) + phi_2 * Episode(3).rho(2)));
131 h_3 = 1/Episode(3).rho(3) * (Episode(3).del_y - ((h_1 *
    Episode(3).rho(1) + h_2 * Episode(3).rho(2))));
132 beta_3 = h_3' * Q * (y_star - d);
133
134 Episode(3).Phi = [Episode(2).Phi, phi_3];
135 Episode(3).Hb = [Episode(2).Hb, h_3];
136 Episode(3).Betas = [Episode(2).Betas; beta_3];
137
138 for b = 3:(max_cheby-1) %for the rest of the trials

```

```

139 Episode = generate_iterative_conjugate(P, d, y_star, Episode,
140 Q, R);
141
142 %% Visualize Approaches
143 if plot_all
144 %Plot control under batch (output, we already have input%
145 fig = figure('Name', 'Batched Learned Input');
146 plot(batch_learned_y(out1_ndx), batch_learned_y(out2_ndx));
147 hold on;
148 plot(y_star(out1_ndx), y_star(out2_ndx));
149 hold off;
150 legend('Learned', 'Goal')
151 xlabel('Mass 1 Position (m)')
152 ylabel('Mass 2 Position (m)')
153 title(sprintf('Batch LQL Output - 10 Conjugate Basis
154 Functions'))
155 axis equal
156
157 %Iterative Plots Plot after 1
158 learned_u = Episode(1).Phi * Episode(1).Betas;
159 learned_y = P * learned_u + d;
160 fig = figure('Name', 'Iterative Learned Input - 1 Basis');
161 plot(learned_y(out1_ndx), learned_y(out2_ndx));
162 hold on;
163 plot(y_star(out1_ndx), y_star(out2_ndx));
164 hold off;
165 legend('Learned', 'Goal')

```

```

166 xlabel('Mass 1 Position (m)')
167 ylabel('Mass 2 Position (m)')
168 title('Output for LQL with 1 Conjugate Basis Function'))
169 axis equal
170 save_figure(update_file_path, fig);

171
172 %Plot after 3
173 learned_u = Episode(3).Phi * Episode(3).Betas;
174 learned_y = P * learned_u + d;
175 fig = figure('Name', 'Iterative Learned Input - 3 Basis');
176 plot(learned_y(out1_ndx), learned_y(out2_ndx));
177 hold on;
178 plot(y_star(out1_ndx), y_star(out2_ndx));
179 hold off;
180 legend('Learned', 'Goal')
181 xlabel('Mass 1 Position (m)')
182 ylabel('Mass 2 Position (m)')
183 title('Output for LQL with 3 Conjugate Basis Functions'))
184 axis equal
185 save_figure(update_file_path, fig);

186
187 %Plot after 8
188 learned_u = Episode(8).Phi * Episode(8).Betas;
189 learned_y = P * learned_u + d;
190 fig = figure('Name', 'Iterative Learned Input - 8 Basis');
191 plot(learned_y(out1_ndx), learned_y(out2_ndx));
192 hold on;
193 plot(y_star(out1_ndx), y_star(out2_ndx));
194 hold off;

```

```

195 legend('Learned', 'Goal')
196 xlabel('Mass 1 Position (m)')
197 ylabel('Mass 2 Position (m)')
198 title('Output for LQL with 8 Conjugate Basis Functions'))
199 axis equal
200 save_figure(update_file_path, fig);
201 end
202 return
203 %% Arbitrary Shape
204 p = 200;
205 loadedShape = load('Saved Data\heart_p200.mat'); %read in the file
206 [P_arb, d_arb] = P_from_ABCD(A, B, C, D, p, x0);
207 num_ilc_states = height(P_arb);
208 num_ilc_inputs = width(P_arb);
209 Q = 100 * eye(num_ilc_states);
210 R = 0 * eye(num_ilc_inputs);
211
212 scale = 10;
213 y_star_x = scale * loadedShape.drawn_x';
214 y_star_y = scale * loadedShape.drawn_y';
215 goal_matrix = [y_star_x, y_star_y]; %stack inputs next to
216 %eachother
217 y_star = reshape(goal_matrix', [], 1); %combine the seperate
218 %goals of each output into one vertical vector, alternating as
219 %necessary
220
221 num_to_try = 100;
222 [arb_conj, arb_betas] = generate_conjugate(num_ilc_inputs,
223 num_to_try, P_arb, Q, R, d_arb, y_star);

```

```

220
221 %% Plot Arbitrary Progression
222 num_basis_to_include = [1, 5, 10, 20, 50, 100, num_to_try];
223
224 if plot_all
225     for num_basis = num_basis_to_include
226         learned_u = arb_conj(:, 1:num_basis) *
227             arb_betas(1:num_basis);
228         learned_y = P_arb * learned_u + d_arb;
229         fig = figure('Name', sprintf('Iterative Learned Input for
230 Specified Shape - %.d Basis', num_basis));
231         plot(learned_y(1:2:end), learned_y(2:2:end));
232         hold on;
233         plot(y_star_x, y_star_y);
234         hold off;
235         legend('Learned', 'Goal')
236         xlabel('Mass 1 Position (m)')
237         ylabel('Mass 2 Position (m)')
238         title(sprintf('Shaped Output for LQL with %.d Conjugate
239 Basis Functions', num_basis))
240         axis equal
241         %save_figure(update_file_path, fig);
242     end
243 end

```

C.5 RL on Conjugate Basis

```
1 %Application of RL to the ILC Problem with Conjugate
2 %Basis Functions
3 %Noah Dunleavy
4 %Honors Thesis under direction of Professor Minh Q. Phan
5 %Thayer School of Engineering
6 clc; clear;
7 addpath('Saved Data', 'Functions');
8 setDefaultFigProp();
9 plot_all = false;
10 update_file_path =
11     -1;%'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis Images\RL
12     on Conjugate'; %set this to the save location if want to
13     update figures, or -1 if not
14
15 if update_file_path ~= -1
16     keyboard %ensure we have to be very concious about ever
17     updating all the images
18 end
19
20 %% System Creation
21 thesis_system = load('Saved Data\thesis_system.mat');
22 A = thesis_system.A;
23 B = thesis_system.B;
24 C = thesis_system.C;
25 D = thesis_system.D;
26 num_inputs = thesis_system.num_inputs;
27 num_outputs = thesis_system.num_outputs;
28 num_states = thesis_system.num_states;
```

```

24 x0 = thesis_system.x0;
25
26 %% ILC Parameters
27 p = 10;
28
29 num_ilc_states = p * num_outputs;
30 num_ilc_inputs = p * num_inputs;
31
32 [P, d] = P_from_ABCD(A, B, C, D, p, x0);
33
34 %% Goal Definition
35 y_star_x = cos(2 * pi * (0:(p-1)) / p)';
36 y_star_y = sin(2 * pi * (0:(p-1)) / p)';
37 goal_matrix = [y_star_x, y_star_y]; %stack inputs next to
            eachother
38 y_star = reshape(goal_matrix', [], 1); %combine the seperate
            goals of each output into one vertical vector, alternating as
            necessary
39
40 %% Conjugate Basis Creation (Batch)
41 %Learning Weights for the Batch Learning
42 Q_batch = 100*eye(num_ilc_states);
43 R_batch = 0 * eye(num_ilc_inputs);
44 max_conj = num_ilc_inputs;
45 [conjugate_basis_functions, conjugate_betas] =
            generate_conjugate(num_ilc_inputs, max_conj, P, Q_batch,
            R_batch, d, y_star);
46
47 %% Conjugate Basis Functions Definition

```

```

48 num_basis_output = max_conj; %full definition / resolution
49 num_basis_input = num_basis_output;
50
51 output_basis_functions = conjugate_basis_functions; %full
      resolution output %conjugate_basis_functions;
52 input_basis_functions = conjugate_basis_functions;
53
54 output_basis_functions_pinv = pinv(output_basis_functions);
      %since done a lot, just compute once and use as constant
55
56 %% Render Goal
57 if (plot_all)
58     fig = figure('Name', 'Goal Output for Arbitrary Basis ILC');
59     plot(y_star(1:2:end), y_star(2:2:(end+1)))
60     axis equal
61     %Should have already saved in RL on ILC
62     %save_figure(update_file_path, fig);
63 end
64
65 %% F_lqr with Varying Basis Functions
66 %RL Parameters
67 Q = 100 * eye(num_ilc_states);
68 R = 10 * eye(num_ilc_inputs);
69 gamma = 0.8;
70
71
72 output_basis = output_basis_functions;
73 A_ilc_basis = eye(num_ilc_states); %fixed - capture the whole
      output

```

```

74 for ndx = 1:3
75     current_basis = ndx;
76     if ndx == 3
77         current_basis = [1, 2];
78     end
79     input_basis = conjugate_basis_functions(:, current_basis);
80     H = -pinv(output_basis) * P * input_basis;
81     F_lqr = discounted_LQR(A_ilc_basis, -H, gamma, Q,
82                             R(current_basis, current_basis));
83 end
84
85 %% Full Resolution Demo
86 FIFO_Trials = basis_ilc_sim(P, d, C*x0,
87                               conjugate_basis_functions, conjugate_basis_functions, y_star,
88                               20);
89 if plot_all
90     plot_dual_ilc('FIFO Conjugate Basis', 'Full Resolution
91                     Conjugate Basis', FIFO_Trials, y_star, -1, 4,
92                     update_file_path);
93 end
94
95 %% Fixed Resolution T_y = I
96 %Fix the output basis
97 num_output_basis = num_ilc_inputs; %assume worst case
98 output_basis_functions = eye(num_output_basis);
99 %Conjugate episodes holder
100 Episode = [];
101 %Setup learning parameters
102 Q = 100 * eye(num_output_basis);

```

```

98 big_R = 1 * eye(num_ilc_inputs);
99 small_R = 1e-6 * eye(num_ilc_inputs);
100 gamma = 0.8;
101 exploration_mag = 10;
102 num_controllers = 5;
103 num_converged = 10;
104 fixed_I_Trials_big_R = [];%hold all the trials
105 fixed_I_Trials_small_R = [];
106 F_big_R = zeros(num_ilc_inputs, num_output_basis);
107 F_small_R = zeros(num_ilc_inputs, num_output_basis);
108 for phi_num = 1:num_ilc_inputs %try every combo possible
109     %Update basis space
110     Episode = generate_iterative_conjugate(P, d, y_star, Episode,
111         Q, 0);
112     phi = Episode(phi_num).Phi(:, phi_num); %get the most recent
113     phi
114     %Learn the controller with big R
115     [temp_Trial, F_phi, ~, ~] = policy_ilc(P, d, C, D, x0, y_star,
116         gamma, Q, big_R(phi_num, phi_num), num_controllers,
117         exploration_mag, phi, output_basis_functions, num_converged);
118     F_big_R(phi_num, :) = F_phi(:, :, end); %save the controller
119     in the stack
120
121     fixed_I_Trials_big_R = [fixed_I_Trials_big_R, temp_Trial];
122
123     %save teh trial info
124
125
126     %Controller with small R
127     [temp_Trial, F_phi, ~, ~] = policy_ilc(P, d, C, D, x0, y_star,
128         gamma, Q, small_R(phi_num, phi_num), num_controllers,
129         exploration_mag, phi, output_basis_functions, num_converged);

```

```

119 F_small_R(phi_num, :) = F_phi(:, :, end); %save the controller
    in the stack
120 fixed_I_Trials_small_R = [fixed_I_Trials_small_R, temp_Trial];
    %save teh trial info
121 end
122 F_lqr_big_R = discounted_LQR(eye(num_output_basis),
    -pinv(output_basis_functions) * P * Episode(end).Phi, gamma,
    Q, big_R);
123 F_lqr_small_R = discounted_LQR(eye(num_output_basis),
    -pinv(output_basis_functions) * P * Episode(end).Phi, gamma,
    Q, small_R);
124
125
126 %Show ow they compare
127 sprintf('F_lqr vs Policy when Fixed Output Basis of I with Big R')
128 lqr = F_lqr_big_R([1, 2, 19, 20], [1, 2, 19, 20])
129 policy = F_big_R([1, 2, 19, 20], [1, 2, 19, 20])
130
131 sprintf('F_lqr vs Policy when Fixed Output Basis of I with Small
    R')
132 lqr = F_lqr_small_R([1, 2, 19, 20], [1, 2, 19, 20])
133 policy = F_small_R([1, 2, 19, 20], [1, 2, 19, 20])
134
135 %Plot the path through learning and straight application
136 if plot_all
137     plot_dual_ilc('Conjugate Phi with Fixed I Output Basis',
        'Learning with Singular Input Basis', fixed_I_Trials_big_R,
        y_star, -1, 4, update_file_path);
138

```

```

139 one_trial = basis_ilc_sim(P, d, C*x0, Episode(end).Phi(:, 1),
140 output_basis_functions, y_star, 10, F_big_R(1, :));
141 plot_dual_ilc('Controller Application of One Conjugate Input
142 Basis with Fixed I Output Basis', 'One-Controller Fixed I
143 Output Basis', one_trial, y_star, -1, 4, update_file_path);
144
145
146 half_trial = basis_ilc_sim(P, d, C*x0, Episode(end).Phi(:, 1:10),
147 output_basis_functions, y_star, 10, F_big_R(1:10, :));
148 plot_dual_ilc('Half Controller Application of Conjugate Input
149 Basis with Fixed I Output Basis', 'Half-Controller Fixed I
150 Output Basis', half_trial, y_star, -1, 4, update_file_path);
151
152
153 seven_five_trial = basis_ilc_sim(P, d, C*x0,
Episode(end).Phi(:, 1:15), output_basis_functions, y_star,
154 10, F_big_R(1:15, :));
155 plot_dual_ilc('Three-Quarter Controller Application of
156 Conjugate Input Basis with Fixed I Output Basis',
157 'Three-Quarter Fixed I Output Basis', seven_five_trial,
158 y_star, -1, 4, update_file_path);
159
160
161 final_Trial = basis_ilc_sim(P, d, C*x0, Episode(end).Phi,
162 output_basis_functions, y_star, 10, F_big_R);
163 plot_dual_ilc('Controller Application of Conjugate Input Basis
164 with Fixed I Output Basis', 'Fixed I Output Basis',
165 final_Trial, y_star, -1, 4, update_file_path);
166
167 end
168
169 %% Fixed Resolution T_y = y*

```

```

154 %Fix the output basis
155 num_output_basis = 1;
156 output_basis_functions = 100 * y_star;
157 %Conjugate episodes holder
158 clear 'Episode';
159 Episode = [];
160 %Setup learning parameters
161 Q = 1000 * eye(num_output_basis);
162 R_y_star = 10 * eye(num_ilc_inputs);
163 gamma = 0.8;
164 exploration_mag = 1;
165 num_controllers = 2;
166 num_converged = 10;
167 y_star_fixed_I_Trials = [];%hold all the trials
168 F = zeros(num_ilc_inputs, num_output_basis);
169 for phi_num = 1:num_ilc_inputs %try every combo possible
170     %Update basis space
171     Episode = generate_iterative_conjugate(P, d, y_star, Episode,
172     Q, 0);
173     phi = Episode(phi_num).Phi(:, phi_num); %get the most recent
174     phi
175     %Learn the controller
176     [temp_Trial, F_phi, ~, ~] = policy_ilc(P, d, C, D, x0, y_star,
177     gamma, Q, R_y_star(phi_num, phi_num), num_controllers,
178     exploration_mag, phi, output_basis_functions, num_converged);
179     F(phi_num, :) = F_phi(:, :, end); %save the controller in the
180     stack
181     y_star_fixed_I_Trials = [y_star_fixed_I_Trials, temp_Trial];
182     %save teh trial info

```

```

177 end
178 F_lqr_y_star = discounted_LQR(eye(num_output_basis),
179     -pinv(output_basis_functions) * P * Episode(end).Phi, gamma,
180     Q, R_y_star);
181
182 %Show how they compare
183 sprintf('F_lqr vs Policy when Fixed Output Basis of y*')
184 lqr = F_lqr_y_star([1, 2, 19, 20])
185 policy = F([1, 2, 19, 20])
186
187 num_sim = 100;
188 y_star_Trial = basis_ilc_sim(P, d, C*x0, Episode(end).Phi,
189     output_basis_functions, y_star, num_sim, F);
190 F_lqr_y_star_mod = discounted_LQR(eye(num_output_basis),
191     -pinv(output_basis_functions) * P * Episode(end).Phi, gamma,
192     Q, R_y_star*(1e-8));
193 y_star_lqr_Trial = basis_ilc_sim(P, d, C*x0, Episode(end).Phi,
194     output_basis_functions, y_star, num_sim, F_lqr_y_star_mod);
195 if plot_all
196     plot_dual_ilc('Controller Application of Conjugate Input Basis
197         with y star Output Basis', 'Fixed y* Output Basis RL',
198         y_star_Trial, y_star, -1, 4, update_file_path);
199     plot_dual_ilc('LQR Controller Application of Conjugate Input
200         Basis with y star Output Basis', 'Fixed y* Output Basis LQR',
201         y_star_lqr_Trial, y_star, -1, 4, update_file_path);
202 end
203
204 %% Fixed Resolution T_y = Phi^b

```

```

196
197 fixed_phi_R = eye(num_ilc_inputs);
198 exploration_mag = 1; %little extra noise to be safe
199 T_y_scale = 10;
200
201 num_converged = 0;
202 num_controllers = 5;
203 %Generate our conjugate basis
204 conjugate_basis_functions = Episode(end).Phi; %use our previous
       iteratively calculated instead of doing a batch -
       computationally faster and more consistent
205 %Fix the output basis
206 num_output_basis = num_ilc_inputs; %assume worst case
207 output_basis_functions = conjugate_basis_functions(:,%
       1:num_output_basis) * T_y_scale;
208 %Setup learning parameters
209 %keep same as before
210 fixed_Phi_b_Trials = []; %hold all the trials
211 F = zeros(num_ilc_inputs, num_output_basis);
212
213 for phi_num = 1:num_ilc_inputs %try every combo possible
214     %Update basis space
215     phi = conjugate_basis_functions(:, phi_num);
216     %Learn the controller
217     [temp_Trial, F_phi, ~, ~] = policy_ilc(P, d, C, D, x0, y_star,
       gamma, Q, fixed_phi_R(phi_num, phi_num), num_controllers,
       exploration_mag, phi, output_basis_functions, num_converged);
218     F(phi_num, :) = F_phi(:, :, end); %save the controller in the
       stack

```

```

219 fixed_Phi_b_Trials = [fixed_Phi_b_Trials, temp_Trial]; %save
220 %teh trial info
221 end
222 F_lqr = discounted_LQR(eye(num_basis_output),
223 -pinv(output_basis_functions) * P *
224 conjugate_basis_functions, gamma, Q, fixed_phi_R);
225 %Show ow they compare
226 sprintf('F_lqr vs Policy when Fixed Conjugate Output Basis')
227 lqr = F_lqr([1, 2, 19, 20], [1, 2, 19, 20])
228 policy = F([1, 2, 19, 20], [1, 2, 19, 20])
229
230
231 %Plot the path
232 if plot_all
233 plot_dual_ilc('c Conjugate Basis', 'Scaled Conjugate Basis',
234 fixed_Phi_b_Trials, y_star, -1, 4, update_file_path);
235
236 F
237 plot_dual_ilc('Scaled Conjugate Basis Application', 'Scaled
238 Conjugate Basis Application', final_conj_Trial, y_star, -1,
239 4, update_file_path);
240 end
241
242 A_fixed = eye(num_ilc_states);
243 B_fixed = -pinv(output_basis_functions) * P *
244 conjugate_basis_functions;
245 poles = eig(A_fixed + B_fixed * F); %A+BF pole formulation
246 mags = abs(poles);
247 if any(mags > 1)
248 sprintf('Unstable')

```

```

241 mags(mags > 1)

242 end

243

244 %% Growing Resolution T_y = Phi^b

245 growing_phi_R = 1 * eye(num_ilc_inputs);

246 Q = 100 * eye(num_ilc_states); %potential for full range

247 exploration_mag = 1;

248 T_y_scale = 10;

249

250 num_converged = 0;

251 num_controllers = 5;

252 %Conjugate episodes holder

253 clear 'Episode';

254 Episode = [];

255 %Setup learning parameters

256 %keep same as before

257 growing_Trials = []; %hold all the trials

258 F = zeros(num_ilc_inputs, num_output_basis);

259

260 for phi_num = 1:num_ilc_inputs %try every combo possible

261 %Update basis space

262 Episode = generate_iterative_conjugate(P, d, y_star, Episode, Q);

263 phi = Episode(phi_num).Phi; %get the most recent phis

264 %Learn the controller

265 output_basis = phi * T_y_scale;

266 input_basis = phi(:, phi_num);

267 [temp_Trial, F_phi, ~, ~] = policy_ilc(P, d, C, D, x0, y_star, gamma, Q(1:phi_num, 1:phi_num), growing_phi_R(phi_num,

```

```

    phi_num), num_controllers, exploration_mag, input_basis,
    output_basis, num_converged);

268 F(phi_num, :) = [F_phi(:, :, end), zeros(1, num_ilc_inputs -
    phi_num)]; %save the controller in the stack
269 growing_Trials = [growing_Trials, temp_Trial]; %save teh trial
    info
270 end
271 Phi = Episode(end).Phi;
272 output_basis = Phi * T_y_scale;
273 input_basis = Phi;
274 F_lqr = discounted_LQR(eye(num_basis_output), -pinv(output_basis)
    * P * input_basis, gamma, Q, growing_phi_R);
275
276 %Show ow they compare (bottom rows should match)
277 sprintf('F_lqr vs Policy when Growing Conjugate Output Basis')
278 lqr = F_lqr([1, 2, 19, 20], [1, 2, 19, 20]);
279 policy = F([1, 2, 19, 20], [1, 2, 19, 20]);
280
281 %Check the poles
282 A_growing = eye(num_ilc_states);
283 B_growing = -pinv(output_basis) * P * input_basis;
284 poles = eig(A_growing + B_growing * F); %A+BF pole formulation
285 mags = abs(poles);
286 if any(mags > 1)
287     sprintf('Unstable')
288     mags(mags > 1)
289 end
290
291 %Plot the path

```

```

292 if plot_all
293     plot_dual_ilc('Growing Basis on IO', 'Growing Basis on IO',
294                   growing_Trials, y_star, -1, 4, update_file_path);
295
296     growing_Trial = basis_ilc_sim(P, d, C*x0, input_basis,
297                                   output_basis, y_star, 100, F);
298
299     plot_dual_ilc('Growing Basis on IO', 'Growing Basis on IO',
300                   growing_Trial, y_star, -1, 4, update_file_path);
301
302 end
303
304
305 %% Rolling Resolution T_y
306
307 rolling_phi_R = 1 * eye(num_ilc_inputs);
308 Q = 100 * eye(num_ilc_states); %potential for full range
309 exploration_mag = 1;
310 T_y_scale = 1;
311
312 num_converged = 0;
313 num_controllers = 5;
314
315 %Conjugate episodes holder
316 clear 'Episode';
317
318 Episode = [];
319
320 %Setup learning parameters
321
322 %keep same as before
323
324 rolling_Trials = []; %hold all the trials
325 F = zeros(num_ilc_inputs, num_output_basis);
326
327
328 for phi_num = 1:num_ilc_inputs %try every combo possible
329     %Update basis space

```

```

317 Episode = generate_iterative_conjugate(P, d, y_star, Episode,
318 Q);
319 phi = Episode(phi_num).Phi; %get the most recent phis
320 %Learn the controller
321 input_basis = phi(:, phi_num);
322 output_basis = input_basis * T_y_scale;
323 [temp_Trial, F_phi, ~, ~] = policy_ilc(P, d, C, D, x0, y_star,
324 gamma, Q(phi_num, phi_num), rolling_phi_R(phi_num, phi_num),
325 num_controllers, exploration_mag, input_basis, output_basis,
326 num_converged);
327 F(phi_num, phi_num) = F_phi(:, :, end); %save the controller
328 in the stack
329 rolling_Trials = [rolling_Trials, temp_Trial]; %save teh trial
330 info
331 end
332 Phi = Episode(end).Phi;
333 output_basis = Phi * T_y_scale;
334 input_basis = Phi;
335 F_lqr = discounted_LQR(eye(num_basis_output), -pinv(output_basis)
336 * P * input_basis, gamma, Q, rolling_phi_R);
337
338 %Show ow they compare (bottom rows should match)
339 sprintf('F_lqr vs Policy when Rolling Conjugate Output Basis')
340 lqr = F_lqr([1, 2, 19, 20], [1, 2, 19, 20]);
341 policy = F([1, 2, 19, 20], [1, 2, 19, 20]);
342
343 %Check the poles
344 A_growing = eye(num_ilc_states);
345 B_growing = -pinv(output_basis) * P * input_basis;

```

```

339 poles = eig(A_growing + B_growing * F); %A+BF pole formulation
340 mags = abs(poles);
341 if any(mags > 1)
342     sprintf('Unstable')
343     mags(mags > 1)
344 end
345
346 %Plot the path
347 if plot_all
348     plot_dual_ilc('Growing Basis on IO', 'Growing Basis on IO',
349         rolling_Trials, y_star, -1, 4, update_file_path);
350
351 growing_Trial = basis_ilc_sim(P, d, C*x0, input_basis,
352     output_basis, y_star, 100, F);
353 plot_dual_ilc('Growing Basis on IO', 'Growing Basis on IO',
354     growing_Trial, y_star, -1, 4, update_file_path);
355 end

```

C.6 RL on ILC

```
1 %Application of Reinforcement Learning to the ILC Problem
2 %Noah Dunleavy
3 %Honors Thesis under direction of Professor Minh Q. Phan
4 %Thayer School of Engineering
5 clc; clear;
6 addpath('Saved Data', 'Functions');
7 setDefaultFigProp();
8 plot_policy = false; %majority of run time goes to figure gen, so
    this speeds up when false
9 plot_decoupled = false;
10 update_file_path =
    -1;%'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis Images\RL
        on ILC'; %set this to the save location if want to update
        figures, or -1 if not
11 if update_file_path ~= -1
12     keyboard %ensure we have to be very concious about ever
        updating all the images
13 end
14
15 %% System Creation
16 thesis_system = load('Saved Data\thesis_system.mat');
17 A = thesis_system.A;
18 B = thesis_system.B;
19 C = thesis_system.C;
20 D = thesis_system.D;
21 num_inputs = thesis_system.num_inputs;
22 num_outputs = thesis_system.num_outputs;
```

```

23 num_states = thesis_system.num_states;
24 x0 = thesis_system.x0;
25
26 %% Goal Definition
27 p = 10;
28
29 % loadedShape = load('Saved Data\heart_p20.mat'); %read in the
30 % file
31 % y_star_x = loadedShape.drawn_x';
32 % y_star_y = loadedShape.drawn_y';
33
34 y_star_x = cos(2 * pi * (0:(p-1)) / p)';
35 y_star_y = sin(2 * pi * (0:(p-1)) / p)';
36 goal_matrix = [y_star_x, y_star_y]; %stack inputs next to
37 %eachother
38 y_star = reshape(goal_matrix', [], 1); %combine the seperate
39 % goals of each output into one vertical vector, alternating as
40 % necessary
41
42 %% ILC System
43 [P, d] = P_from_ABCD(A, B, C, D, p, x0); %y(1:p) = P * u(0:(p-1))
44 % + d
45
46 num_ilc_states = p * num_outputs; %error bar - one for each
47 % output at each time step
48
49 num_ilc_inputs = p * num_inputs; %u bar - each time step gets an
50 % input
51
52 %% ILC State-space so that e(j+1) = A * e(j) + B * del_u_j1 and
53 % y(j) = e(j)

```

```

44 A_ilc = eye(num_ilc_states);
45 B_ilc = -P;
46 C_ilc = eye(num_ilc_states);
47 D_ilc = 0;
48 if (~is_controllable(A_ilc, B_ilc))
49     fprintf('The ILC System is not Controllable!')
50 return
51 end
52
53 %% Policy Iteration RL Parameters, R = 1
54 %Learning Weights
55 Q = 100 * eye(num_ilc_states); %cost of each error
56 R = 1 * eye(num_ilc_inputs); %penalize inputs, or more accurately
      change in input
57 %Setting this too small causes controller to fail, but
      logically 0
58 %should be safe
59 gamma = 0.8;
60 policy_exploration_mag = 1;
61 F_ilc_lqr = discounted_LQR(A_ilc, B_ilc, gamma, Q, R); %perfect
      knowledge controller
62
63
64
65 %Iteration Counts
66 Pj_dim = (num_ilc_inputs + num_ilc_states); %the square diension
      of the Pj matrix
67 num_controllers = 5; %number of controllers to try to create

```

```

68 num_collections_per_controller = Pj_dim^2; %number of datasets
    needed per controller to update
69 num_converged = 1000; %how many trials to then go through once we
    are done 'learning' (so we arent noisy)
70
71 %To be able to set R smaller, increase exploration magnitude ad
    the number
72 %of collections per controller
73
74 total_trial_count = num_controllers *
    num_collections_per_controller + num_converged;
75
76 %Preallocate the space and define structure
77 ILC_Trial(total_trial_count).output = [];
78 ILC_Trial(total_trial_count).input = [];
79 ILC_Trial(total_trial_count).output_error = [];
80 ILC_Trial(total_trial_count).del_u = [];
81
82 %% Policy Iteration Learning Process, R = 1
83 rng('default'); rng(10); %ensure same randomization each time
84 F_ilc = zeros(num_ilc_inputs, num_ilc_states, num_controllers +
    1); %start with no controller
85
86 %Prepopulate the first trial
87 trial_num = 1;
88 ILC_Trial(trial_num).input = zeros(num_ilc_inputs, 1); %start
    with open-loop / no input
89 ILC_Trial(trial_num).output = [C*x0; d]; %open loop response is
    IC and then d term

```

```

90 ILC_Trial(trial_num).output_error = y_star - d; %relevant error
91
92 trial_num = 2; %start at second trial now
93 for iteration = 1:num_controllers
94     Uk_stack = zeros(num_collections_per_controller, 1);
95     Xk_stack = zeros(num_collections_per_controller, (Pj_dim)^2);
96
97     %Simulate the necessary trials
98     for trial = 1:num_collections_per_controller %number of trials
99         to collect before updatin controller
100             %ILC / Real Controller Process
101             %Inputs
102             exploration_term = rand_range(num_ilc_inputs, 1,
103             -policy_exploration_mag, policy_exploration_mag); %jiggle to
104             learn
105
106             ILC_Trial(trial_num).del_u = F_ilc(:, :, iteration) *
107             ILC_Trial(trial_num - 1).output_error + exploration_term;
108
109             ILC_Trial(trial_num).input = ILC_Trial(trial_num - 1).input
110             + ILC_Trial(trial_num).del_u;
111
112             %Simulate Reality
113
114             relevant_output = P * ILC_Trial(trial_num).input + d; %y(1)
115             -> y(p)
116
117             ILC_Trial(trial_num).output = [C*x0 +
118             D*ILC_Trial(trial_num).input(1:num_inputs); relevant_output];
119
120             %total output y(0) -> y(p) for completeness
121
122             %Calculate Error
123
124             ILC_Trial(trial_num).output_error = y_star -
125             relevant_output;

```

```

109      %error_next_law = ILC_Trial(trial_num - 1).output_error - P
110      * ILC_Trial(trial_num).del_u; %verify that this matches the
111      produced error, that is:
112
113      %e(j+1) = e(j) - P*del_u(j+1) when del_u(j+1) = L * e(j)
114      = u(j) - u(j-1)
115
116      %RL Translation
117
118      state = ILC_Trial(trial_num - 1).output_error; %analogous
119      state, x(k) -> e_(j-1)
120
121      input = ILC_Trial(trial_num).del_u;%analogous input, u(k)
122      -> del_u_(j) = L * e_(j-1)
123
124      next_state = ILC_Trial(trial_num).output_error; %x(k+1) =
125      e(j)
126
127      next_input = F_ilc(:, :, iteration) * next_state; %no
128      exploration term here
129
130
131      xu_stack = [state; input];
132
133      xu_next_stack = [next_state; next_input];
134
135
136      Xk_stack(trial, :) = kron(xu_stack', xu_stack') - gamma *
137      kron(xu_next_stack', xu_next_stack');
138
139      Uk_stack(trial, :) = input' * R * input + state' * Q *
140      state;
141
142
143      trial_num = trial_num + 1;
144
145  end
146
147
148  %Calculate P and new controller
149  % [U, S, V] = svd(Xk_stack);

```

```

129 % rank(S)
130 PjS = pinv(Xk_stack) * Uk_stack;
131 Pj = reshape(PjS, Pj_dim, Pj_dim);
132 Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
133 %reduces error)
134 Pjuu = Pj((num_ilc_states+1):end, (num_ilc_states+1):end);
135 PjxuT = Pj((num_ilc_states+1):end, 1:num_ilc_states);
136 new_F = -pinv(Pjuu) * PjxuT;
137 F_ilc(:, :, iteration + 1) = new_F;
138 end
139 for ndx = 1:num_converged
140 %Inputs
141 ILC_Trial(trial_num).del_u = F_ilc(:, :, end) *
142 ILC_Trial(trial_num - 1).output_error;
143 ILC_Trial(trial_num).input = ILC_Trial(trial_num - 1).input +
144 ILC_Trial(trial_num).del_u;
145 %Simulate Reality
146 relevant_output = P * ILC_Trial(trial_num).input + d; %y(1) ->
147 y(p)
148 ILC_Trial(trial_num).output = [C*x0 +
149 D*ILC_Trial(trial_num).input(1:num_inputs); relevant_output];
%total output y(0) -> y(p) for completeness
150 %Calculate Error
151 ILC_Trial(trial_num).output_error = y_star - relevant_output;
152 trial_num = trial_num + 1;
153 end
154 controller_error = norm(F_ilc(:, :, end) -
F_ilc_lqr)/numel(F_ilc_lqr)

```

```

151
152 %% Visualize Learning
153 if plot_policy
154     to_plot = 4;
155     plot_dual_ilc('Policy Iteration on ILC', 'Policy Iteration for
156     ILC', ILC_Trial, y_star, -1, to_plot, update_file_path);
157     plot_controller_history('Policy Iteration ILC Controller',
158     'Policy Iteration ILC Controller Weights', F_ilc, 4, 5,
159     update_file_path); %we cannot plot *all* the IOs because
160     there are so many, pick a few
161 end
162
163 %% Input Decoupled Learning
164 %Iteration Counts
165 decoupled_exploration_mag = policy_exploration_mag;
166 Pj_dim = (1 + num_ilc_states);
167 num_controllers = 5;
168 num_collections_per_controller = Pj_dim^2;
169 num_converged_decoupled = num_collections_per_controller;
170
171 total_trial_count_decoupled = num_ilc_inputs * num_controllers *
172     num_collections_per_controller + num_converged_decoupled;
173     %now need a * num inputs
174
175 %Preallocate the space and define structure
176 ILC_Trial_decoupled(total_trial_count_decoupled).output = [];
177 ILC_Trial_decoupled(total_trial_count_decoupled).input = [];
178 ILC_Trial_decoupled(total_trial_count_decoupled).output_error =
179     [];

```

```

173 ILC_Trial_decoupled(total_trial_count_decoupled).del_u = [];
174
175 %% Input Decoupled Learning Process
176 rng('default'); rng(10); %ensure same randomization each time
177 F_ilc_decoupled = zeros(num_ilc_inputs, num_ilc_states, 1);
178
179 %Prepopulate the first trial
180 trial_num = 1;
181 ILC_Trial_decoupled(trial_num).input = zeros(num_ilc_inputs, 1);
    %start with open-loop / no input
182 ILC_Trial_decoupled(trial_num).output = [C*x0; d]; %open loop
    response is IC and then d term
183 ILC_Trial_decoupled(trial_num).output_error = y_star - d;
    %relevant error
184
185 trial_num = 2; %start at second trial now
186 for iteration = 1:(num_controllers)
187     for input_num = 1:num_ilc_inputs
188         Uk_stack = zeros(num_collections_per_controller, 1);
189         Xk_stack = zeros(num_collections_per_controller,
(Pj_dim)^2);
190         F_i = F_ilc_decoupled(input_num, :, end);
191         %Simulate the necessary trials
192         for trial = 1:num_collections_per_controller %number of
trials to collect before updatin controller
193             %ILC / Real Controller Process
194             %Inputs
195             exploration_term = rand_range(1, 1,
-decoupled_exploration_mag, decoupled_exploration_mag);

```

```

%jiggle to learn

196      ILC_Trial_decoupled(trial_num).del_u =
F_ilc_decoupled(:, :, iteration) *
ILC_Trial_decoupled(trial_num - 1).output_error;

197

198      ILC_Trial_decoupled(trial_num).del_u(input_num) =
ILC_Trial_decoupled(trial_num).del_u(input_num) +
exploration_term; %difference between PI - only learn on one
input

199

200      ILC_Trial_decoupled(trial_num).input =
ILC_Trial_decoupled(trial_num - 1).input +
ILC_Trial_decoupled(trial_num).del_u;

201      %Simulate Reality

202      relevant_output = P *
ILC_Trial_decoupled(trial_num).input + d; %y(1) -> y(p)

203      ILC_Trial_decoupled(trial_num).output = [C*x0 +
D*ILC_Trial_decoupled(trial_num).input(1:num_inputs);
relevant_output]; %total output y(0) -> y(p) for completeness

204      %Calculate Error

205      ILC_Trial_decoupled(trial_num).output_error = y_star -
relevant_output;

206

207

208      %RL Translation

209      state = ILC_Trial_decoupled(trial_num - 1).output_error;
%analogous state, x(k) -> e_(j-1)

210      full_input = ILC_Trial_decoupled(trial_num).del_u; %Uk
vs Xk use different inputs

```

```

211     input = full_input(input_num);
212
213     next_state =
214         ILC_Trial_decoupled(trial_num).output_error; %x(k+1) = e(j)
215
216     next_input = F_i * next_state; %no exploration term here
217
218     xu_stack = [state; input];
219
220     xu_next_stack = [next_state; next_input];
221
222     Xk_stack(trial, :) = kron(xu_stack', xu_stack') - gamma
223     * kron(xu_next_stack', xu_next_stack');
224
225     Uk_stack(trial, :) = full_input' * R * full_input +
226     state' * Q * state;
227
228     trial_num = trial_num + 1;
229
230     end
231
232
233     %Calculate P and new controller
234
235     PjS = pinv(Xk_stack) * Uk_stack;
236
237     Pj = reshape(PjS, Pj_dim, Pj_dim);
238
239     Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
240     reduces error)
241
242     Pjuu = Pj((num_ilc_states+1):end, (num_ilc_states+1):end);
243
244     PjxuT = Pj((num_ilc_states+1):end, 1:num_ilc_states);
245
246     new_F_i = -pinv(Pjuu) * PjxuT;
247
248     F_ilc_decoupled(:, :, end + 1) = F_ilc_decoupled(:, :, end);
249
250     F_ilc_decoupled(input_num, :, end) = new_F_i;
251
252     end
253
254 end

```

```

236 for ndx = 1:num_converged_decoupled
237     %Inputs
238     ILC_Trial_decoupled(trial_num).del_u = F_ilc_decoupled(:, :, end) * ILC_Trial_decoupled(trial_num - 1).output_error;
239     ILC_Trial_decoupled(trial_num).input =
240         ILC_Trial_decoupled(trial_num - 1).input +
241         ILC_Trial_decoupled(trial_num).del_u;
242     %Simulate Reality
243     relevant_output = P * ILC_Trial_decoupled(trial_num).input +
244     d; %y(1) -> y(p)
245     ILC_Trial_decoupled(trial_num).output = [C*x0 +
246     D*ILC_Trial_decoupled(trial_num).input(1:num_inputs);
247     relevant_output]; %total output y(0) -> y(p) for completeness
248     %Calculate Error
249     ILC_Trial_decoupled(trial_num).output_error = y_star -
250     relevant_output;
251     trial_num = trial_num + 1;
252 end
253 controller_error = norm(F_ilc_decoupled(:, :, end) -
254     F_ilc_lqr)/numel(F_ilc_lqr)
255
256 %% Visualize Learning
257 if plot_decoupled
258     to_plot = 4;
259     plot_dual_ilc('Input Decoupling on ILC', 'Input Decoupling for
260     ILC', ILC_Trial_decoupled, y_star, -1, to_plot,
261     update_file_path);
262     plot_controller_history('Input Decoupling ILC Controller',
263     'Input Decoupling ILC Controller Weights', F_ilc_decoupled,

```

```

        4, 5, update_file_path, true);

254 end

255

256 %% Policy Iteration RL Parameters, R = 1e-6

257 %Learning Weights

258 Q = 100 * eye(num_ilc_states); %cost of each error

259 small_R = 1e-6 * eye(num_ilc_inputs); %penalize inputs, or more
    accurately change in input

260 %Setting this too small causes controller to fail, but
    logically 0

261 %should be safe

262 gamma = 0.8;

263 small_R_policy_exploration_mag = 1000;

264 F_ilc_lqr_small_R = discounted_LQR(A_ilc, B_ilc, gamma, Q,
    small_R); %perfect nowledge controller

265

266 %Iteration Counts

267 Pj_dim = (num_ilc_inputs + num_ilc_states); %the square diension
    of the Pj matrix

268 num_controllers = 5; %number of controllers to try to create

269 num_collections_per_controller = Pj_dim^2; %number of datasets
    needed per controller to update

270 num_converged = 100; %how many trials to then go through once we
    are done 'learning' (so we arent noisey)

271

272 %To be able to set R smaller, increase exploration magnitude ad
    the number

273 %of collections per controller

274

```

```

275 total_trial_count = num_controllers *
276     num_collections_per_controller + num_converged;
277
278 %Preallocate the space and define structure
279 ILC_Trial_small_R(total_trial_count).output = [];
280 ILC_Trial_small_R(total_trial_count).input = [];
281 ILC_Trial_small_R(total_trial_count).output_error = [];
282 ILC_Trial_small_R(total_trial_count).del_u = [];
283
284 %% Policy Iteration Learning Process, R = 1e-6
285 rng('default'); rng(10); %ensure same randomization each time
286 F_ilc_policy_small_R = zeros(num_ilc_inputs, num_ilc_states,
287     num_controllers + 1); %start with no controller
288
289 %Prepopulate the first trial
290 trial_num = 1;
291 ILC_Trial_small_R(trial_num).input = zeros(num_ilc_inputs, 1);
292     %start with open-loop / no input
293 ILC_Trial_small_R(trial_num).output = [C*x0; d]; %open loop
294     response is IC and then d term
295 ILC_Trial_small_R(trial_num).output_error = y_star - d; %relevant
296     error
297
298 trial_num = 2; %start at second trial now
299 for iteration = 1:num_controllers
300     Uk_stack = zeros(num_collections_per_controller, 1);
301     Xk_stack = zeros(num_collections_per_controller, (Pj_dim)^2);
302
303     %Simulate the necessary trials

```

```

299    for trial = 1:num_collections_per_controller %number of trials
      to collect before updatin controller
300
301        %ILC / Real Controller Process
302
303        %Inputs
304
305        exploration_term = rand_range(num_ilc_inputs, 1,
306        -small_R_policy_exploration_mag,
307        small_R_policy_exploration_mag); %jiggle to learn
308
309        ILC_Trial_small_R(trial_num).del_u =
310        F_ilc_policy_small_R(:, :, iteration) *
311        ILC_Trial_small_R(trial_num - 1).output_error +
312        exploration_term;
313
314        ILC_Trial_small_R(trial_num).input =
315        ILC_Trial_small_R(trial_num - 1).input +
316        ILC_Trial_small_R(trial_num).del_u;
317
318        %Simulate Reality
319
320        relevant_output = P * ILC_Trial_small_R(trial_num).input +
321        d; %y(1) -> y(p)
322
323        ILC_Trial_small_R(trial_num).output = [C*x0 +
324        D*ILC_Trial_small_R(trial_num).input(1:num_inputs);
325
326        relevant_output]; %total output y(0) -> y(p) for completeness
327
328        %Calculate Error
329
330        ILC_Trial_small_R(trial_num).output_error = y_star -
331        relevant_output;
332
333        %error_next_law = ILC_Trial(trial_num - 1).output_error - P
334        * ILC_Trial(trial_num).del_u; %verify that this matches the
335        produced error, that is:
336
337            %e(j+1) = e(j) - P*del_u(j+1) when del_u(j+1) = L * e(j)
338            = u(j) - u(j-1)
339
340
341
342

```

```

313     %RL Translation
314
315     state = ILC_Trial_small_R(trial_num - 1).output_error;
316
317     %analogous state, x(k) -> e_(j-1)
318
319     input = ILC_Trial_small_R(trial_num).del_u;%analogous
320
321     input, u(k) -> del_u_(j) = L * e_(j-1)
322
323     next_state = ILC_Trial_small_R(trial_num).output_error;
324
325     %x(k+1) = e(j)
326
327     next_input = F_ilc_policy_small_R(:, :, iteration) *
328
329     next_state; %no exploration term here
330
331
332     xu_stack = [state; input];
333
334     xu_next_stack = [next_state; next_input];
335
336
337     Xk_stack(trial, :) = kron(xu_stack', xu_stack') - gamma *
338
339     kron(xu_next_stack', xu_next_stack');
340
341     Uk_stack(trial, :) = input' * small_R * input + state' * Q
342
343     * state;
344
345
346     trial_num = trial_num + 1;
347
348 end
349
350
351 %Calculate P and new controller
352
353 % [U, S, V] = svd(Xk_stack);
354
355 % rank(S)
356
357 PjS = pinv(Xk_stack) * Uk_stack;
358
359 Pj = reshape(PjS, Pj_dim, Pj_dim);
360
361 Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
362
363 reduces error)
364
365 Pjuu = Pj((num_ilc_states+1):end, (num_ilc_states+1):end);

```

```

335 PjxuT = Pj((num_ilc_states+1):end, 1:num_ilc_states);
336 new_F = -pinv(Pjuu) * PjxuT;
337 F_ilc_policy_small_R(:, :, iteration + 1) = new_F;
338 end
339
340 for ndx = 1:num_converged
341 %Inputs
342 ILC_Trial_small_R(trial_num).del_u = F_ilc_policy_small_R(:, :, end) * ILC_Trial_small_R(trial_num - 1).output_error;
343 ILC_Trial_small_R(trial_num).input =
344 ILC_Trial_small_R(trial_num - 1).input +
ILC_Trial_small_R(trial_num).del_u;
345 %Simulate Reality
346 relevant_output = P * ILC_Trial_small_R(trial_num).input + d;
%y(1) -> y(p)
347 ILC_Trial_small_R(trial_num).output = [C*x0 +
D*ILC_Trial_small_R(trial_num).input(1:num_inputs);
relevant_output]; %total output y(0) -> y(p) for completeness
348 %Calculate Error
349 ILC_Trial_small_R(trial_num).output_error = y_star -
relevant_output;
350 trial_num = trial_num + 1;
351 end
352 controller_error = norm(F_ilc_policy_small_R(:, :, end) -
F_ilc_lqr_small_R)/numel(F_ilc_lqr_small_R)
353
354 %% Visualize Learning
355 if plot_policy
356 to_plot = 4;

```

```
356 plot_dual_ilc('Small R Policy Iteration on ILC', 'Policy  
Iteration for ILC - Reduced R', ILC_Trial_small_R, y_star,  
-1, to_plot, update_file_path);  
357 plot_controller_history('Small R Policy Iteration ILC  
Controller', 'Policy Iteration ILC Controller Weights -  
Reduced R', F_ilc_policy_small_R, 4, 5, update_file_path);  
%we cannot plot *all* the IOs because there are so many, pick  
a few  
358 end
```

Appendix D

Matlab Functions

D.1 Controllability Check

```
1 function [controllable] = is_controllable(A, B, precision)
2 %Check whether or not a matrix specified by A and B matrices is
3 %controllable
4 %Inputs:
5 %A: matrix - state dynamics
6 %B: matrix - input dynamics
7 %precision: scalar - set how precise matlab is with its
8 %rounding
9 %Outputs:
10 %controllable: bool - whether or not the system is controllable
11 if exist('precision', 'var') %ability to set precision
12     A = vpa(A, precision);
13     B = vpa(B, precision);
14 end
15
16 controllability_matrix = ctrb(A, B);
17 controllable = (rank(controllability_matrix) == height(A));
18
19 end
```

Listing D.1: Check the Controllability of a System

D.2 Decoupled Learning for Iterative Learning Control

```
1 function [ILC_Trial, F_decoupled, controller_error, F_lqr] =
2     decoupled_ilc(P, d, C, D, x0, y_star, gamma, Q, R,
3         num_controllers, exploration_mag, input_basis_functions,
4         output_basis_functions, num_converged)
5 %Perform the RL input decoupled learning on an ILC system since
6     done so much in
7 %thesis
8 %Inputs:
9     %P: matrix - inputs u(0->(p-1)) to outputs y(1->p)
10    %d: vector - noise/initial conditions matrix
11    %C: matrix - state to output descriptor
12    %D: matrix - input to output descriptor
13    %x0: vector - initial state
14    %y_star: vector - goal output
15    %gamma: scalar - discount factor
16    %Q: matrix - cost of states (errors)
17    %R: matrix - cost of inputs (change in inputs)
18    %num_controllers: scalar - number of controllers to learn
19    %exploration_mag: scalar - range around 0 to explore (defaults
        to 1)
20    %input_basis_functions: matrix - basis functions on the inputs
        (defaults to identity)
21    %output_basis_functions: matrix - basis functions on the
        outputs (defaults to identity)
22    %num_converged: scalar - number of trials to simulate out
        without exploration (defaults to 0)
23 %Outputs:
```

```

20 %ILC_Trial: structure with indexed by trial number, contains
21     %inputs
22     %betas (input basis weights)
23     %del_betas (change in input betas)
24     %outputs
25     %alphas (output basis weights)
26     %output error (y* - y)
27     %alpha error
28 %F_decoupled: matrix - controller learning history
29 %controller_error: scalar - normalized error from LQR
30 %F_lqr: matrix - lqr controller to compare to
31
32 %Required Parameters to System Info
33 num_inputs = width(D);
34
35 %Default paramters
36 if ~exist('num_converged', 'var')
37     num_converged = 0; %default to no converged trials
38 end
39 if ~exist('exploration_mag', 'var')
40     exploration_mag = 1;
41 end
42 if ~exist('input_basis_functions', 'var')
43     input_basis_functions = eye(width(P));
44 end
45 if ~exist('output_basis_functions', 'var')
46     output_basis_functions = eye(height(P));
47     output_basis_functions_pinv = output_basis_functions;%save on
        compute time

```

```

48 else
49     output_basis_functions_pinv = pinv(output_basis_functions);
50 end
51 num_ilc_states = width(output_basis_functions);
52 num_ilc_inputs = width(input_basis_functions);
53
54 alpha_star = output_basis_functions_pinv * y_star;
55
56 %Calculate optimal controller
57 F_lqr = discounted_LQR(eye(num_ilc_states),
58                         -output_basis_functions_pinv * P * input_basis_functions,
59                         gamma, Q, R);
60
61 %Iteration Counts
62 Pj_dim = num_ilc_states + 1;
63 num_collections_per_controller = Pj_dim^2;
64 total_trial_count = num_ilc_inputs * num_controllers *
65                         num_collections_per_controller + num_converged;
66
67 %Preallocate structure
68 ILC_Trial(total_trial_count).input = []; %input
69 ILC_Trial(total_trial_count).betas = []; %basis representation of
70                               input
71 ILC_Trial(total_trial_count).del_beta = [];
72 ILC_Trial(total_trial_count).output = []; %output
73 ILC_Trial(total_trial_count).alphas = []; %basis representation
74                               of output
75 ILC_Trial(total_trial_count).output_error = []; %output error

```

```

72 ILC_Trial(total_trial_count).alpha_error = []; %alpha error
73
74 F_decoupled = zeros(num_ilc_inputs, num_ilc_states,
75                         num_controllers + 1); %start with no controller
76 %Prepopulate the first trial
77 trial_num = 1;
78 ILC_Trial_decoupled(trial_num).betas = zeros(num_ilc_inputs, 1);
79             %start with no basis guessed
80
81 ILC_Trial_decoupled(trial_num).input = input_basis_functions *
82             ILC_Trial_decoupled(trial_num).betas;
83
84 ILC_Trial_decoupled(trial_num).output = [C*x0; d]; %open loop
85             response is IC and then d term
86 ILC_Trial_decoupled(trial_num).alphas =
87             output_basis_functions_pinv * d;
88 ILC_Trial_decoupled(trial_num).output_error = y_star - d;
89             %relevant error
90 ILC_Trial_decoupled(trial_num).alpha_error = alpha_star -
91             ILC_Trial_decoupled(trial_num).alphas;
92
93 trial_num = 2; %start at second trial now
94
95 for iteration = 1:num_controllers
96
97     for input_num = 1:num_ilc_inputs
98
99         Uk_stack = zeros(num_collections_per_controller, 1);
100
101         Xk_stack = zeros(num_collections_per_controller,
102                         (Pj_dim)^2);
103
104         F_i = F_decoupled(input_num, :, end);
105
106         %Simulate the necessary trials

```

```

93     for trial = 1:num_collections_per_controller %number of
94         trials to collect before updatin controller
95             %ILC / Basis Controller Process
96             %Beta Coeffecients
97             exploration_term = rand_range(1, 1, -exploration_mag,
98             exploration_mag); %jiggle to learn
99
100
101             ILC_Trial_decoupled(trial_num).del_beta = F_decoupled(:, :
102             end) * ILC_Trial_decoupled(trial_num - 1).alpha_error;
103             ILC_Trial_decoupled(trial_num).del_beta(input_num) =
104             ILC_Trial_decoupled(trial_num).del_beta(input_num) +
105             exploration_term;
106
107             ILC_Trial_decoupled(trial_num).betas =
108             ILC_Trial_decoupled(trial_num - 1).betas +
109             ILC_Trial_decoupled(trial_num).del_beta;
110
111             %Beta to Inputs
112             ILC_Trial_decoupled(trial_num).input =
113             input_basis_functions * ILC_Trial_decoupled(trial_num).betas;
114
115             %Simulate Reality
116             relevant_output = P *
117             ILC_Trial_decoupled(trial_num).input + d; %y(1) -> y(p)
118             ILC_Trial_decoupled(trial_num).alphas =
119             output_basis_functions_pinv * relevant_output;
120
121             ILC_Trial_decoupled(trial_num).output = [C*x0 +
122             D*ILC_Trial_decoupled(trial_num).input(1:num_inputs);
123             relevant_output]; %total output y(0) -> y(p) for completeness
124
125             %Calculate Error

```

```

109         ILC_Trial_decoupled(trial_num).output_error = y_star -
110             relevant_output;
111
112         %RL Translation
113         state = ILC_Trial_decoupled(trial_num - 1).alpha_error;
114         %analogous state
115         full_input = ILC_Trial_decoupled(trial_num).del_beta;
116         %analogous input
117         input = full_input(input_num);
118
119         next_state = ILC_Trial_decoupled(trial_num).alpha_error;
120         %x(k+1) = e_alpha(j)
121
122         next_input = F_i * next_state; %no exploration term here
123
124         xu_stack = [state; input];
125         xu_next_stack = [next_state; next_input];
126
127         Xk_stack(trial, :) = kron(xu_stack', xu_stack') - gamma
128             * kron(xu_next_stack', xu_next_stack');
129         Uk_stack(trial, :) = full_input' * R * full_input +
130             state' * Q * state;
131
132         trial_num = trial_num + 1;
133
134     end
135
136
137     %Calculate P and new controller
138     PjS = pinv(Xk_stack) * Uk_stack;
139     Pj = reshape(PjS, Pj_dim, Pj_dim);

```

```

131      Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
132      %reduces error)
133
134      Pjuu = Pj((num_ilc_states+1):end, (num_ilc_states+1):end);
135      PjxuT = Pj((num_ilc_states+1):end, 1:num_ilc_states);
136      new_F_i = -pinv(Pjuu) * PjxuT;
137      F_decoupled(:, :, end + 1) = F_decoupled(:, :, end);
138      F_decoupled(input_num, :, end) = new_F_i;
139
140    end
141  end
142
143  for ndx = 1:num_converged
144
145    %ILC / Basis Controller Process
146
147    %Beta Coeffecients
148
149    ILC_Trial_decoupled(trial_num).del_beta = F_decoupled(:, :, iteration) * ILC_Trial_decoupled(trial_num - 1).alpha_error;
150
151    ILC_Trial_decoupled(trial_num).betas =
152      ILC_Trial_decoupled(trial_num - 1).betas +
153      ILC_Trial_decoupled(trial_num).del_beta;
154
155    %Beta to Inputs
156
157    ILC_Trial_decoupled(trial_num).input = input_basis_functions *
158      ILC_Trial_decoupled(trial_num).betas;
159
160    %Simulate Reality
161
162    relevant_output = P * ILC_Trial_decoupled(trial_num).input +
163      d; %y(1) -> y(p)
164
165    ILC_Trial_decoupled(trial_num).alphas =
166      output_basis_functions_pinv * relevant_output;
167
168    ILC_Trial_decoupled(trial_num).output = [C*x0 +
169      D*ILC_Trial_decoupled(trial_num).input(1:num_inputs);
170
171      relevant_output]; %total output y(0) -> y(p) for completeness

```

```
151 %Calculate Error
152 ILC_Trial_decoupled(trial_num).output_error = y_star -
    relevant_output;
153 ILC_Trial_decoupled(trial_num).alpha_error = alpha_star -
    ILC_Trial_decoupled(trial_num).alphas;
154
155 trial_num = trial_num + 1;
156 end
157 controller_error = norm(F_decoupled(:, :, end) -
    F_lqr)/numel(F_lqr);
158
159 end
```

Listing D.2: Input Decoupled ILC

D.3 Default Figure Properties

```
1 function [] = setDefaultFigProp()
2 %Text size scaling set in save_figure
3 %Set some global parameters for general figure properties
4
5 %Set global properties for plotting for the executed script
6 set(groot, 'DefaultFigureColor', 'w'); %Default background color
    for figures
7
8 set(groot, 'DefaultAxesFontName', 'Arial'); %Font
9 set(groot, 'DefaultAxesLineWidth', 1.8);
10 set(groot, 'DefaultLegendBackgroundAlpha', 0.5);
11
12 setappdata(groot, 'DefaultSubtitleFontSize', 10); %just a temp
    small subtitle font (save_figure should overwrite)
13
14 %Default Line
15 set(groot, 'DefaultLineLineWidth', 4);
16 set(groot, 'DefaultLineMarkerSize', 16);
17
18 %Default stair properties (inherit most from line)
19 set(groot, 'DefaultStairLineWidth', 4);
20 set(groot, 'DefaultStairLineStyle', '-');
21
22 %Default scatter properties
23 %set(groot, 'DefaultScatterMarkerSize', 50);
24 %set(groot, 'DefaultScatterFilled', '1');
25 set(groot, 'DefaultScatterMarker', 'o');
```

```
26 set(groot, 'DefaultScatterSizeData', 200); % Set default marker  
    size  
27 set(groot, 'DefaultScatterLineWidth', 4);  
28  
29  
30 end
```

Listing D.3: Default Figure Properties

D.4 Discounted LQR Solution

```
1 function [F] = discounted_LQR(A, B, discount_factor, Q, R,
2     verbose)
3 %From system specifications, determine the optimal feedback
4 % controller
5 %(Linear quadratic regulator). Of form u(k) = F*x(k) optimally
6 %Inputs:
7 %A: matrix - system dynamics
8 %B: matrix - input dynamics
9 %discount_factor: scalar - discount factor
10 %Q: matrix - state costs
11 %R: matrix - input costs
12 %verbose: bool - system dynamics
13 %Outputs:
14 %F: matrix - found controller
15
16 %Check if the verbose provided
17 if nargin < 6
18     verbose = false; %Default false
19 end
20
21 %Solve for P - the solution to Algebraic Riccati Equation
22 % (Discrete - DARE):
23 R_gamma = R/discount_factor;
24 A_gamma = sqrt(discount_factor)*A;
25 [P, ~, ~] = idare(A_gamma, B, Q, R_gamma); %solve the riccati
26 %equation
27
```

```

24 %Verify it is a solution (text output for user)
25 if verbose
26     test_P = A_gamma' * P * A_gamma - A_gamma' * P * B / ((R_gamma +
27         B' * P * B) * B' * P * A_gamma + Q; %verify P is a solution
28     sprintf('Solved Ps suitability to solve the Riccati equation
29         is %g', norm(test_P - P)) %names the answer this way - want
30         this small
31 end
32
33
34 %Calculate F
35 F = (-1/sqrt(discount_factor)) * inv((transpose(B) * P * B +
36     R_gamma) * transpose(B) * P * A_gamma); %LQR solution
37
38 end

```

Listing D.4: Discounted LQR Controller Generation

D.5 Draw a Goal

```
1 function [x, y] = draw_to_XY(resolution)
2 %Written by ChatGPT with prompt: 'I would like a matlab
3 function that
4 %opens up a figure that I can draw on with my mouse, and then
5 %it returns
6 %two arrays of the x positions and y positions of the thing I
7 %drew. I
8 %would like to set the resoltuon (aka number of xy pairs) in
9 %the
10 %function call'
11 %
12 % draw_to_XY - Opens a figure to draw with the mouse and
13 % returns x, y coordinates.
14 %
15 % Syntax: [x, y] = draw_with_mouse(resolution)
16 %
17 % Inputs:
18 % resolution - Number of points to return in the output arrays.
19 %
20 % Outputs:
21 % x - Array of x positions of the drawn curve.
22 % y - Array of y positions of the drawn curve.
23 %
24 if nargin < 1
25     resolution = 100; % Default resolution if not specified
26 end
```

```

23 % Create a new figure
24 figure;
25 hold on;
26 grid on
27 axis([0 1 0 1]); % Set axis limits (can be adjusted as needed)
28 title('Draw with your mouse (click and drag). Press Enter when
done.');
29 set(gca, 'Position', [0.1, 0.1, 0.8, 0.8]); % Adjust axis
position for aesthetics
30 set(gcf, 'WindowButtonMotionFcn', @mouse_move_callback); %
Enable mouse motion tracking
31 set(gcf, 'WindowButtonDownFcn', @mouse_down_callback); % Start
drawing on mouse press
32 set(gcf, 'WindowButtonUpFcn', @mouse_up_callback); % Stop
drawing on mouse release
33 pan off; zoom off; % Disable default interactions
34
35 % Variables to store points
36 drawing = false; % Indicates whether the mouse is pressed
37 points_x = [];
38 points_y = [];
39
40 % Callback to track mouse movements while pressed
41 function mouse_move_callback(~, ~)
42     if drawing
43         current_point = get(gca, 'CurrentPoint');
44         points_x(end + 1) = current_point(1, 1);
45         points_y(end + 1) = current_point(1, 2);
46         plot(points_x, points_y, 'b-');

```

```

47         drawnow;
48
49     end
50
51 % Callback to start drawing
52 function mouse_down_callback(~, ~)
53
54     drawing = true;
55
56 % Callback to stop drawing
57 function mouse_up_callback(~, ~)
58
59     drawing = false;
60
61 % Wait for the user to press Enter
62 pause;
63 close(gcf); % Close the figure
64
65 % Interpolate to match the desired resolution
66 t_raw = linspace(0, 1, length(points_x));
67 t_interp = linspace(0, 1, resolution);
68
69 x = interp1(t_raw, points_x, t_interp, 'linear');
70 y = interp1(t_raw, points_y, t_interp, 'linear');
71 end

```

Listing D.5: Draw XY Goals

D.6 Figure Saving

```
1 function [file_path] = save_figure(path, fig_list,
2     name_overwrite, shaped)
3 %Save figures to specified path following naming convention
4 %Inputs:
5     %path: string - where to save the file to,
6     %fig_list: list - figure handles
7     %name_overwrite: string - if you wish to overwrite the naming
8     %convention
9     %shaped: bool - whether or not it is shaped (should be square)
10 %Outputs:
11     %file_path: string - file save location
12
13 %Set save settings
14 title_size = 45;
15 subtitle_size = 30;
16 axes_size = 40;
17 tick_size = 35;
18 legend_size = 30;
19
20 %Default condition
21 if ~exist('name_overwrite', 'var')
22     name_overwrite = -1;
23 end
24 if ~exist('shaped', 'var')
25     shaped = false;
26 end
```

```

26 if path == -1 %condition for if we are not updating the figures
27     return
28 end
29
30 %Dimension params
31 aspect_ratio = 16/12;
32 figure_width = 16;
33 figure_height = figure_width / aspect_ratio;
34 if shaped
35     figure_width = figure_height;
36 end
37
38 for fig = fig_list
39     %Determine what name to give the file
40     if name_overwrite == -1
41         file_name = fig.Name;
42     else
43         file_name = name_overwrite;
44     end
45
46     %Set figure sizes
47     fig.Units = 'inches';
48     fig.Position = [0, 0, figure_width, figure_height];
49
50     % Ensure the axes is square
51     ax = findobj(fig, 'Type', 'axes');
52     for a = ax'
53         %Set font sizes

```

```

54     a.FontSize = tick_size; %tick size - do this one first
55
56     because otherwise it overwrites others
57
58     title(a, a.Title.String, 'FontSize', title_size,
59             'FontWeight', 'bold');
60
61     xlabel(a, a.XLabel.String, 'FontSize', axes_size);
62
63     ylabel(a, a.YLabel.String, 'FontSize', axes_size);
64
65     %adjust subtitles
66
67     if isprop(a, 'Subtitle') && ~isempty(a.Subtitle.String)
68
69         subtitle(a, a.Subtitle.String, 'FontSize',
70                  subtitle_size, 'FontWeight', 'normal');
71
72     end
73
74
75     %adjust legend
76
77     lgd = findobj(fig, 'Type', 'legend');
78
79     for l = lgd'
80
81         l.FontSize = legend_size;
82
83         l.Location = 'northeast';
84
85     end
86
87
88     set(a, 'PlotBoxAspectRatio', [figure_width, figure_height,
89          1]); %ensure plot matched to our desired scales
90
91     if shaped
92
93         set(a, 'DataAspectRatio', [1, 1, 1]); %force square
94
95         axis(a, 'equal'); %dont distort scaling
96
97     else
98
99         set(a, 'DataAspectRatioMode', 'auto');
100
101         axis(a, 'tight'); % Ensure scaling is not distorted
102
103     end

```

```

79      %Fix offset caused by labels
80
81      insets = get(a, 'TightInset'); %space needed for labels
82
83      left_offset = insets(1); %padding due to ylabel
84
85      bottom_offset = insets(2); %padding due to xlabel
86
87      right_offset = insets(3); %padding due to xlabel
88
89      top_offset = insets(4); %padding due to title
90
91
92      % Adjust outer position to fit tightly around the plot
93
94      % Ensure no unnecessary margins are included
95
96      a.OuterPosition = [left_offset/figure_width,
97
98          bottom_offset/figure_height, (figure_width - left_offset -
99          right_offset)/figure_width, (figure_height - top_offset -
100         bottom_offset)/figure_height];

```

```
101 end  
102  
103 end
```

Listing D.6: Figure Saving

D.7 Generate Chebyshev Polynomials

```
1 function [cheby_functions] = generate_chebyshev(cheb_resolution,
2 num_cheby)
3 %Generate a matrix of 'num_cheby' functions, with
4 % depth/resolution of
5 %'cheb_resolution'
6 %Input:
7 % cheb_resolution: scalar - resolution of each functions,
8 % height' of matrix
9 %num_cheby: scalar - number of chebyshevs to generate
10 %Output:
11 %cheby_functions: matrix - chebyshev functions, type 1
12
13 cheby_x = linspace(-1, 1, cheb_resolution); %define the cheby 'x'
14
15 cheby_functions = ones(cheb_resolution, num_cheby);
16 cheby_functions(:, 2) = cheby_x;
17
18 for ndx = 3:num_cheby %this is the recursive cheby generation.
19 Could be from a file, but more helpful to see
20 cheby_functions(:, ndx) = 2 * cheby_x .* cheby_functions(:, ndx - 1) - cheby_functions(:, ndx - 2); %build cheby out
21
22 end
23
24 end
```

Listing D.7: Recursive Chebyshev Polynomial Generation

D.8 Batch Generate Conjugate Basis Functions

```
1 function [conjugate_basis_functions, conjugate_betas] =
2     generate_conjugate(basis_resolution, num_basis, P, Q, R, d,
3     y_star)
4 %Create conjugate basis functions for a system, derived from
5 %chebyshevs
6 %Input:
7 %height: scalar - height/resolution of the conjugate functions
8 %to generate
9 %num_basis: scalar - num_basis/number of functions to generate
10 %P: matrix - system matrix relating
11 %Q: matrix - cost of states
12 %d: vector - IC vector
13 %y_star: vector - goal output
14 %Output:
15 %conjugate_basis_functions: matrix - functions that satify
16 %conjunctionality for the system P
17 %conjuagte_betas: vector - optimal weighting to minimize error
18
19
20 if nargin < 7
21     y_star = zeros(height(P), 1); %default goal to zeros
22 end
23 if nargin < 6
24     d = zeros(height(P), 1); %default noise to zeros
25 end
26 if nargin < 5
27     R = 0 * eye(height(P)); %default input cost to 0
28 end
```

```

23 if nargin < 4
24     Q = 100 * eye(width(P)); %default state costs to 100
25 end
26
27 %Create chebys
28 batch_input = generate_chebyshev(basis_resolution, num_basis);
29
30 %Generate Batch output
31 batch_outputs_delta = P * batch_input; %do not include the d
   term, because we want the difference in outputs, which
   excludes d
32
33 W = batch_input' * R * batch_input + batch_outputs_delta' * Q *
   batch_outputs_delta; %W matrix
34
35 while rank(W) < min(basis_resolution, num_basis) %if dont get a
   full rank W
36     batch_input = rand_range(basis_resolution, num_basis, -10,
   10); %go different input approach
37
38 %Generate Batch output
39 batch_outputs_delta = P * batch_input; %do not include the d
   term, because we want the difference in outputs, which
   excludes d
40
41 W = batch_input' * R * batch_input + batch_outputs_delta' * Q *
   batch_outputs_delta; %W matrix
42 sprintf('Using random inputs')
43 end

```

```

44
45 rho_batch = chol(W); %cholesky decomposition of W to get the
46 % optimal coeffecients for the batch
47
48 T_b = batch_input / (rho_batch); %/ is same as * inv()
49
50 conjugate_basis_functions = T_b;
51
52 H_b = batch_outputs_delta / (rho_batch);
53 conjugate_betas = H_b' * Q * (y_star - d); %determined optimal
54 % weights for given basis functions (off of e_0)
55
56 end

```

Listing D.8: Batch Conjugate Function Generation

D.9 Iteratively Generate Conjugate Basis Functions

```
1 function [Episode] = generate_iterative_conjugate(P, d, y_star,
2     old_episodes, Q, R)
3 %From an existing conjugate basis space, apply a new episode to
4     generate a
5 %new basis
6 %Input:
7     %P: matrix -descriptie matrix to map u to y (y = Pu + d)
8     %d: vector - handle noise in the relation to map u to y
9     %y_star: vector - goal output
10    %old_episodes: struct - structure holding old iterative data
11    %Q: matrix - cost of states
12    %R: matrix - cost of inputs
13
14    %Output:
15        %Episode: struct - all the previous trials and info, plus the
16        new one
17
18 num_ilc_inputs = width(P);
19 num_ilc_states = height(P);
20
21 if nargin < 6%if no R
22     R = zeros(num_ilc_inputs);
23 end
24 if nargin < 5%if no Q
25     Q = 100 * eye(num_ilc_states);
26 end
27
28 if ((nargin < 4) || isempty(old_episodes)) %if we have no
```

```

    episodes to start, do the first trial
24 Episode(1).del_u = ones(num_ilc_inputs, 1); %first chebyshev
      is all ones
25 Episode(1).del_y = P * Episode(1).del_u;
26
27 %Compute W
28 W = Episode(1).del_u' * R * Episode(1).del_u +
      Episode(1).del_y' * Q * Episode(1).del_y;
29
30 %rho_1, phi_1, h_1, and beta_1
31 Episode(1).rho = chol(W);
32 phi_1 = Episode(1).del_u * Episode(1).rho^-1;
33 h_1 = Episode(1).del_y * Episode(1).rho^-1;
34 beta_1 = h_1' * Q * (y_star - d); %use e0 for all
35
36 Episode(1).Phi = phi_1;
37 Episode(1).Hb = h_1;
38 Episode(1).Betas = beta_1;
39
40 return
41
42 end
43
44
45 %If this is not our first trial
46 Episode = old_episodes;
47
48 b = length(Episode);
49
50
51 %Generate our dels
52 tried_inputs = zeros(num_ilc_inputs, b);
53
54 for ndx = 1:b

```

```

50    tried_inputs(:, ndx) = Episode(ndx).del_u;
51 end
52
53 gen_cheby = generate_chebyshev(num_ilc_inputs, b + 1); %there is
54     likely a more effecient way, but to ensure all the
55     generations are exactly the same
56 next_input = gen_cheby(:, b+1); %next input is our final cheby
57 while (rank([tried_inputs, next_input]) < (b+1)) %ensure that our
58     added input does not ruin the rank of the system
59
60 next_input = rand_range(num_ilc_inputs, 1, -1, 1);
61
62 return
63
64 end
65
66 Episode(b + 1).del_u = next_input; %-u0, but open loop
67 Episode(b + 1).del_y = P * Episode(b + 1).del_u;
68
69 %Compute first component
70 Episode(b + 1).rho(1) = 1/Episode(1).rho(1) * (Episode(1).del_u'
71     * R * Episode(b + 1).del_u + Episode(1).del_y' * Q *
72     Episode(b + 1).del_y);
73
74 %Middle Components
75 if b >= 2
76
77     for i = 2:b
78
79         sum_term = 0;
80
81         for j = 1:(i-1)
82
83             sum_term = sum_term + Episode(i).rho(j) * Episode(b +
84                 1).rho(j); %hard to vector format across structs
85
86         end
87
88         Episode(b+1).rho(i) = (1 / Episode(i).rho(i)) *

```

```

        (Episode(i).del_u' * R * Episode(b+1).del_u +
    Episode(i).del_y' * Q * Episode(b+1).del_y - sum_term);

73 end

74 end

75

76 %Last Component

77 Episode(b + 1).gamma = Episode(b + 1).rho(1:b) * Episode(b +
    1).rho(1:b)';

78 Episode(b + 1).rho(b+1) = sqrt(Episode(b + 1).del_u' * R *
    Episode(b + 1).del_u + Episode(b + 1).del_y' * Q * Episode(b +
    1).del_y - Episode(b + 1).gamma);

79

80 %Phi calculation

81 new_basis = (1/Episode(b + 1).rho(b+1)) * (Episode(b + 1).del_u -
    Episode(b).Phi * Episode(b + 1).rho(1:b)');

82 Episode(b+1).Phi = [Episode(b).Phi, new_basis];

83 %Hb

84 new_h = (1/Episode(b + 1).rho(b+1)) * (Episode(b + 1).del_y -
    Episode(b).Hb * Episode(b + 1).rho(1:b)');

85 Episode(b+1).Hb = [Episode(b).Hb, new_h];

86 %Betas

87 new_beta = new_h' * Q * (y_star - d);

88 Episode(b+1).Betas = [Episode(b).Betas; new_beta];

89

90 end

```

Listing D.9: Recursive Conjugate Polynomial Generation

D.10 ILC Simulation

```
1 function [ILC_Trial] = basis_ilc_sim(P, d, y0, T_u, T_y, y_star,
2     num_trials, error_controller)
3 %Generate the ILC trials and representative data for a given
4     system (P, d)
5 %with initial output y0, and basis description (T_u, T_y) to move
6     to a goal (y_star) over specified
7 %trial count (num_trials)
8 %Inputs:
9     %P: matrix - inputs u(0->(p-1)) to outputs y(1->p)
10    %d: vector - noise/initial conditions matrix
11    %y0: vector - initial output (since y(0) skipped by P)
12    %T_u: matrix - Input basis functions (default to identity)
13    %T_y: matrix - Output basis functions (default to identity)
14    %y_star: vector - goal output
15    %num_trials: scalar - how many trials to simulate
16    %error_controller: matrix - controller of the ILC system
17        (defaults to
18        %0.8)
19 %Outputs:
20     %ILC_Trial: structure with indexed by trial number, contains
21         %inputs
22         %betas (input basis weights)
23         %del_betas (change in input betas)
24         %outputs
25         %alphas (output basis weights)
26         %output error (y* - y)
```

```

23      %alpha error
24
25 if T_u == -1
26     T_u = eye(width(P));
27 end
28 if T_y == -1
29     T_y = eye(height(P));
30 end
31
32 num_basis_input = width(T_u); %basis functions are pr x n_u
33 num_basis_output = width(T_y); %pm x n_y
34
35 T_y_pinv = pinv(T_y); %most common form of T_y we will use (to go
            from real to basis land)
36 alpha_star = T_y_pinv * y_star;
37
38 H = T_y_pinv * P * T_u; %descriptive IO controller, such that
            alpha = H * beta + T_y+ * d
39
40 if (~is_controllable(eye(num_basis_output), -H))
41     fprintf('The ILC System is not Controllable with %.d basis
            inputs and %.d basis outputs!\n', num_basis_input,
            num_basis_output)
42 end
43
44 if ~exist('error_controller', 'var')
45     error_controller = 0.5 * pinv(H); %perfect knowledge
            controller as default
46 end

```

```

47
48 %Preallocate structure
49 ILC_Trial(num_trials).input = []; %input
50 ILC_Trial(num_trials).betas = []; %basis representation of input
51 ILC_Trial(num_trials).del_betas = [];
52 ILC_Trial(num_trials).output = []; %output
53 ILC_Trial(num_trials).alphas = []; %basis representation of output
54 ILC_Trial(num_trials).output_error = []; %output error
55 ILC_Trial(num_trials).alpha_error = []; %alpha error
56
57 %Initial trial
58 trial = 1;
59 %Inputs
60 ILC_Trial(trial).del_betas = zeros(num_basis_input, 1); %no
    'input'
61 ILC_Trial(trial).betas = zeros(num_basis_input, 1);
62 ILC_Trial(trial).input = T_u * ILC_Trial(trial).betas;
63
64 %Outputs
65 relevant_output = P * ILC_Trial(trial).input + d; %y(1) -> y(p)
66 ILC_Trial(trial).output = [y0; relevant_output];
67 ILC_Trial(trial).alphas = T_y_pinv * relevant_output; %alpha(j) =
    T_y+ * y(1:p)
68
69 %Errors
70 ILC_Trial(trial).output_error = y_star - relevant_output;
71 ILC_Trial(trial).alpha_error = alpha_star -
    ILC_Trial(trial).alphas;
72

```

```

73 trial = trial + 1;

74

75 %Simulate Rest

76 for trial_num = 2:num_trials

77 %Inputs

78 ILC_Trial(trial).del_betas = error_controller *

    ILC_Trial(trial - 1).alpha_error; %del_B(j) = L * e_alpha(j-1)

79 ILC_Trial(trial).betas = ILC_Trial(trial - 1).betas +

    ILC_Trial(trial).del_betas; %B(j) = B(j-1) + del_B(j)

80 ILC_Trial(trial).input = T_u * ILC_Trial(trial).betas; %u(j) =

    T_u * B(j)

81

82 %Outputs

83 relevant_output = P * ILC_Trial(trial).input + d; %y(1) -> y(p)

84 ILC_Trial(trial).output = [y0; relevant_output];

85 ILC_Trial(trial).alphas = T_y_pinv * relevant_output;

86 %alpha(j) = T_y+ * y(1:p)

87

88 %Errors

89 ILC_Trial(trial).output_error = y_star - relevant_output;

90 ILC_Trial(trial).alpha_error = alpha_star -

    ILC_Trial(trial).alphas;

91

92 trial = trial + 1;

93 end

94 end

```

Listing D.10: ILC System Simulation

D.11 P Matrix from ABCD Model

```
1 function [P, d] = P_from_ABCD(A, B, C, D, p, x0)
2 %Construct the P matrix and d matrix for given matrix values
3 %Will satisfy the equations y_bar = P * u_bar + d
4 %y is y(1) -> y(p)
5 %u is u(0) -> u(p-1)
6 %d captures initial conditions and noise
7 %Inputs:
8     %A: matrix - state dynamics matrix
9     %B: matrix - input dynamics matrix
10    %C: matrix - state to output
11    %D: matrix - input to output
12    %p: scalar - number of steps to map out
13    %x0: vector - initial conditions
14 %Outputs:
15     %P: matrix - ILC system mapping matrix for u(0->(p-1)) ->
16     %y(1->p)
17     %d: vector - captures the 'disturbance' caused by initial
18     %conditions
19
20 num_states = width(A);
21 num_inputs = width(B);
22 num_outputs = height(C);
23
24 %Construct 'True' P
25 P = zeros(num_states, num_inputs);
26 ic_matrix = zeros(num_outputs, num_states); %matrix which governs
27     %the impact of the Ics (for sim)
```

```

25 for row = 1:p
26     row_start = ((row - 1) * num_outputs) + 1;
27     row_end = row_start + num_outputs - 1;
28     for col = 1:row
29         col_start = ((col - 1) * num_inputs) + 1;
30         col_end = col_start + num_inputs - 1;
31         if (row + 1 == col)
32             P(row_start:row_end, col_start:col_end) = D;
33         else
34             mat_pow = row - col;
35             P(row_start:row_end, col_start:col_end) = C *
(A^mat_pow) * B;
36         end
37
38     end
39     ic_matrix(row_start:row_end, :) = C * A^row; %construct the IC
matrix
40 end
41 d = ic_matrix * x0;
42
43 %Verify P is accurate
44 demo_in = rand_range(num_inputs * p, 1, -2, 2);
45 demo_P_out = P * demo_in + d; %y = Pu + d
46
47 demo_dlsim_in = reshape(demo_in, num_inputs, []); %dlsim takes a
pxr matrix, whereas P is a pr x 1
48 demo_dlsim_out_matrix = dlsim(A, B, C, D, [demo_dlsim_in;
zeros(1, num_inputs)], x0);
49 demo_dlsim_out = reshape(demo_dlsim_out_matrix(2:end, :)', [], 1);

```

```
50
51 if (norm(demo_P_out - demo_dlsim_out)/numel(demo_P_out) > 1e-6)
52 fprintf('P does not capture output accurately')
53 end
54
55
56 end
```

Listing D.11: Descriptive Matrix P from ABCD

D.12 Plot Basis Coefficients

```
1 function [alpha_error_fig, beta_error_fig, alpha_prog_fig,
2     beta_prog_fig] = plot_ilc_coefficients(figure_name,
3     graph_title, ILC_Trial, to_plot, beta_star, betas_to_plot,
4     alphas_to_plot, save_path)
5 %Note: beta star error is sometimes illogical
6 %Plot out the progression of coefficients in an ILC problem
7 %described in a
8 %basis space
9 %Inputs:
10 %figure_name: string - name of the figure that is opened, or
11 %precursor to saved image
12 %graph_title: string - displayed title header on plot
13 %ILC_Trial: structure - containing iterations of coefficients
14 %to_plot: scalar/vector - indicate which trials or how many to
15 %plot
16 %beta_star: vector - sometimes illogical, but when there is a
17 %defined beta weights, use to compute the error
18 %betas_to_plot: scalar/vector - indicate which beta values (or
19 %how many) to plot
20 %alphas_to_plot: scalar/vector - see above
21 %save_path: string - where to save the generated plots to
22 %Outputs:
23 %alpha_error_fig: figure - handle to alpha errors
24 %beta_error_fig: figure - handle to beta errors
25 %alpha_prog_fig: figure - handle to showing alpha evolve
26 %beta_prog_fig: figure - handle to showing betas evolve
```

```

20 max_coef = 5; %default maximum # of coefficients to plot
21 if nargin < 8 %no save path
22     save_path = -1;
23 end
24 if nargin < 7 %if no alpha to plot
25     alphas_to_plot = max_coef;
26 end
27 if nargin < 6
28     betas_to_plot = max_coef;
29 end
30 if nargin < 5
31     beta_star = -1; %ensure we have a matrix for matching
32 end
33 beta_star_passed = true;
34 if isscalar(beta_star)
35     if(beta_star == -1)
36         beta_star_passed= false;
37     end
38 end
39
40 num_trials = length(ILC_Trial);
41 marker_color_scale = 0.75; %how to scale the color difference
    from line to point
42 marker_size = 20;
43 subtitle_size = getappdata(groot, 'DefaultSubtitleFontSize');
44
45 if isscalar(to_plot) %check if we are passing in a count or
    specific trials to plot
46     trials_to_plot = floor(linspace(1, num_trials, to_plot));

```

```

47     trials_to_plot(end) = num_trials;
48 else
49     trials_to_plot = to_plot;
50 end
51
52 %Process ILC Strcture
53 num_betas = length(ILC_Trial(1).betas);
54 num_alphas = length(ILC_Trial(1).alphas);
55 num_plot = length(trials_to_plot);
56
57 %Determine which coefficients to plot
58 if isscalar(alphas_to_plot)
59     if(alphas_to_plot == -1) %default plot
60         alphas_to_plot = max_coef;
61     end
62     alphas_to_plot = floor(linspace(1, num_alphas,
63                             alphas_to_plot)); %space out the plots
64     alphas_to_plot(end) = num_alphas; %ensure the last one is shown
65 end
66 if isscalar(betas_to_plot)
67     if(betas_to_plot == -1) %default plot
68         betas_to_plot = max_coef;
69     end
70     betas_to_plot = floor(linspace(1, num_alphas, betas_to_plot));
71     %space out the plots
72     betas_to_plot(end) = num_betas; %ensure the last one is shown
73 end
74
75 %Progression of errors on coefficients

```

```

74 alpha_error_norm = zeros(num_trials, 1);
75 beta_error_norm = zeros(num_trials, 1);
76 for ndx = 1:num_trials
77     alpha_error_norm(ndx) = norm(ILC_Trial(ndx).alpha_error) /
78         num_alphas;
79     if beta_star_passed
80         beta_error_norm(ndx) = norm(beta_star -
81             ILC_Trial(ndx).betas) / num_betas;
82     end
83 end
84
85 alpha_error_fig = figure('Name', sprintf('%s - Alpha Error Norm
86 Progression', figure_name));
87 semilogy(1:num_trials, alpha_error_norm, 'r*', 'LineStyle', ':');
88 set(gca, 'YScale', 'log'); %force log scale (sometimes linear if
89 small range)
90 title('Alpha Error', graph_title);
91 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
92 xlabel('Trial Number')
93 ylabel('Normalized Amplitude')
94
95 if beta_star_passed %if we were given a beta_star
96     beta_error_fig = figure('Name', sprintf('%s - Beta Error Norm
97 Progression', figure_name));
98     semilogy(1:num_trials, beta_error_norm, 'r*', 'LineStyle', ':');
99     set(gca, 'YScale', 'log'); %force log scale (sometimes linear
100    if small range)
101    title('Beta Error', graph_title)
102    subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)

```

```

97 xlabel('Trial Number')
98 ylabel('Normalized Amplitude')
99 end
100
101 %Transfer structure to array for plotting
102 %For progression smoothing
103 full_betas = zeros(num_betas, num_trials);
104 full_alphas = zeros(num_alphas, num_trials);
105 for ndx = 1:num_trials
106     full_betas(:, ndx) = ILC_Trial(ndx).betas;
107     full_alphas(:, ndx) = ILC_Trial(ndx).alphas;
108 end
109
110 %Progression of Alphas
111 alpha_prog_fig = figure('Name', sprintf('%s - Alpha Coefficient
112 Progression', figure_name));
112 legend_list = "[REMOVE ME]";
113 hold on;
114 for alpha = alphas_to_plot
115     temp_plot = plot(0:(num_trials-1), full_alphas(alpha, :),
116 'MarkerSize', marker_size, 'LineStyle', '-', 'Marker', 'o',
117 'MarkerIndices', trials_to_plot);
118     temp_plot.MarkerFaceColor = temp_plot.Color *
119     marker_color_scale;%make the marker colors slightly darker
than lines
117     temp_plot.MarkerEdgeColor = temp_plot.Color *
marker_color_scale;
118     legend_list(end + 1) = sprintf('Alpha %.d', alpha);
119 end

```

```

120 hold off;
121
122 title('Alpha Coefficients')
123 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
124 xlabel('Trial Number')
125 ylabel('Alpha Value')
126 xticks(trials_to_plot - 1);
127 legend(legend_list(2:end))
128
129 %Progression of Beats
130 beta_prog_fig = figure('Name', sprintf('%s - Beta Coefficient
    Progression', figure_name));
131 legend_list = "[REMOVE ME]";
132 hold on;
133 for beta = betas_to_plot
134     temp_plot = plot(0:(num_trials-1), full_betas(beta, :),
    'MarkerSize', marker_size, 'LineStyle', '-', 'Marker', 'o',
    'MarkerIndices', trials_to_plot);
135     temp_plot.MarkerFaceColor = temp_plot.Color *
    marker_color_scale; %make the marker colors slightly darker
    than lines
136     temp_plot.MarkerEdgeColor = temp_plot.Color *
    marker_color_scale;
137     legend_list(end + 1) = sprintf('Beta %.d', beta);
138 end
139 hold off;
140
141 title('Beta Coefficients')
142 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)

```

```
143 xlabel('Trial Number')
144 ylabel('Beta Value')
145 xticks(trials_to_plot - 1);
146 legend(legend_list(2:end))
147
148 %Save images if possible
149 if exist('save_path', 'var') %if a save path was provided
150     save_figure(save_path, [alpha_error_fig, alpha_prog_fig,
151                             beta_prog_fig]); %save all the figures to the path
152     if beta_star_passed
153         save_figure(save_path, beta_error_fig);
154     end
155 end
156 end
```

Listing D.12: Plot Basis Space Coefficients

D.13 Plot Controller History

```
1 function [figure_list] = plot_controller_history(figure_name,
2                                                 graph_title, controllers, inputs_of_interest,
3                                                 states_of_interest, save_path, decoupled)
4 %Plot the progression of a given controller parameters for
5 %indicated inputs
6 %Inputs:
7 %figure_name: string - name of the figure that is opened, or
8 %precursor to saved image
9 %graph_title: string - displayed title header on plot
10 %controllers: matrix - r x n x 1 matrix, showing the learning
11 %of r controllers from n states over 1 iterations
12 %inputs_of_interest: scalar/vector - for when there are lots
13 %of inputs, which ones to plot
14 %states_of_interest: scalar/vector - see above
15 %save_path: string - where to save the generated plots to
16 %decoupled: bool - indicates whether or not it was learned via
17 %input decoupling (only show trials where input changes)
18 %Outputs:
19 %figure_list: list - figure handles generated
20
21 %Plot settings
22 marker_color_scale = 0.75; %how to scale the color difference
23 %from line to point
24 marker_size = 20;
25 subtitle_size = getappdata(groot, 'DefaultSubtitleFontSize');
26 %default set in fig properties
27
```

```

19 num_inputs = height(controllers);
20 num_states = width(controllers);
21
22 %Assign defaults
23 if nargin < 7 %if flag of whether or not it was input decoupled
24     decoupled = false;
25 end
26 if nargin < 6
27     save_path = -1;
28 end
29 if nargin < 5 %if no states index passed
30     states_of_interest = floor(linspace(1, num_states,
31                                         min(num_states, 4))); %do not overclutter sith state points
31 end
32 if nargin < 4 %if no passed input indexes
33     inputs_of_interest = [1, num_inputs]; %default to showing 2 of
34                                         them (sometimes there are a lot)
34 end
35
36 %Shortcuts
37 if (states_of_interest == -1) %if -1, plot all components
38     states_of_interest = 1:num_states;
39 elseif (isscalar(states_of_interest))
40     states_of_interest = floor(linspace(1, num_states,
41                                         states_of_interest));
41     states_of_interest(end) = num_states;
42 end
43
44 if (inputs_of_interest == -1)

```

```

45     inputs_of_interest = 1:num_inputs;
46 elseif (isscalar(inputs_of_interest))
47     inputs_of_interest = floor(linspace(1, num_inputs,
48         inputs_of_interest));
49     inputs_of_interest(end) = num_inputs;
50 end
51
52 figure_list = [];
53 for input_num = inputs_of_interest
54     figure_list(end + 1) = figure('Name', sprintf('%s - Input %.d
55         Controller Weights', figure_name, input_num));
56
57 if decoupled
58     trial_ndx = [1, (input_num+1):num_inputs:size(controllers,
59         3)]; %if input decoupled, alternate to only show points when
60         updated
61 else
62     trial_ndx = 1:size(controllers, 3);
63 end
64
65 legend_list = "[REMOVE ME]";
66 hold on;
67 for state = states_of_interest
68     temp_plot = plot((trial_ndx-1),
69         squeeze(controllers(input_num, state, trial_ndx)),
70         'MarkerSize', marker_size, 'Marker', 'o');
71     temp_plot.MarkerFaceColor = temp_plot.Color *
72         marker_color_scale;%make the marker colors slightly darker
73         than lines

```

```

66     temp_plot.MarkerEdgeColor = temp_plot.Color *
marker_color_scale;
67     legend_list(end + 1) = sprintf('Weight on State %.d',
state);
68 end
69 hold off;
70
71 legend(legend_list(2:end), "BackgroundAlpha", 0.5)
72 title(sprintf('Input %.d Controller Weights', input_num));
73 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
74 xlabel('Controller Number')
75 ylabel('Controller Weight')
76 xticks(trial_ndx-1);
77
78
79 end
80
81 %Save images if possible
82 if exist('save_path', 'var') %if a save path was provided
83 for fig_num = figure_list
84     save_figure(save_path, figure(fig_num)); %save all the
figures to the path
85 end
86 end
87
88
89 end

```

Listing D.13: Plot Controller Histories

D.14 Plot Dual-Mass System

```
1 function [mass1_fig, mass2_fig, input1_fig, input2_fig] =
2     plot_two_mass(figure_name, graph_title, outputs, inputs,
3     save_path)
4 %For the dual spring-mass system repeatedly used, plot positions
5 %and inputs
6 %Inputs:
7 %figure_name: string - name of the figure that is opened, or
8 %precursor to saved image
9 %graph_title: string - displayed title header on plot
10 %outputs: matrix - 2 x k matrix, where row 1 is mass 1
11 %position, row 2 is mass 2 position
12 %inputs: matrix - 2 x k, row 1 is input1 and row 2 is input 2
13 %save_path: string - where to save the generated plots to
14 %Outputs:
15 %mass1_fig: figure - handle to position of mass 1
16 %mass2_fig: figure - handle to position of mass 2
17 %input1_fig: figure - handle to input 1
18 %input2_fig: figure - handle to input 2
19
20 if (height(outputs) > width(outputs)) %Ensure wide
21     outputs = outputs';
22 end
23
24 if (height(inputs) > width(inputs)) %Ensure wide
25     inputs = inputs';
26 end
```

```

23
24 num_samples = width(outputs);
25 subtitle_size = getappdata(groot, 'DefaultSubtitleFontSize');
26
27 %Output Style
28 output_color = [0 0.4471 0.7412];
29 output_size = 1.2;
30 output_style = '-';
31
32 %Mass 1 Position
33 mass1_fig = figure('Name', sprintf('%s - Mass 1 Position',
34 figure_name));
34 stairs(0:(num_samples-1), outputs(1, :), 'Color', output_color,
35 'LineStyle', output_style, 'LineWidth', output_size);
35 stairs(0:(num_samples-1), outputs(1, :), 'Color', output_color);
36 title('Mass 1 Position');
37 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
38 xlabel('Sample Number (k)')
39 ylabel('Position (m)')
40
41 %Mass 2 Position
42 mass2_fig = figure('Name', sprintf('%s - Mass 2 Position',
43 figure_name));
43 stairs(0:(num_samples-1), outputs(2, :), 'Color', output_color);
44 title('Mass 2 Position')
45 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
46 xlabel('Sample Number (k)')
47 ylabel('Position (m)')
48

```

```

49 %Input Style
50 input_color = [1, 0, 1];
51 input_size = 1.2;
52 input_style = '-';
53
54 %Input 1
55 input1_fig = figure('Name', sprintf('%s - Input 1 Magnitude',
56 figure_name));
56 stairs(0:(num_samples-1), inputs(1, :), 'Color', input_color);
57 title('Input 1')
58 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
59 xlabel('Sample Number (k)')
60 ylabel('Force (N)')
61
62 %Input 2
63 input2_fig = figure('Name', sprintf('%s - Input 2 Magnitude',
64 figure_name));
64 stairs(0:(num_samples-1), inputs(2, :), 'Color', input_color);
65 title('Input 2')
66 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
67 xlabel('Sample Number (k)')
68 ylabel('Force (N)')
69
70 %Save images if possible
71 if exist('save_path', 'var') %if a save path was provided
72 save_figure(save_path, [mass1_fig, mass2_fig, input1_fig,
73 input2_fig]); %save all the figures to the path
74 end

```

75 | end

Listing D.14: Plot a Dual-Mass System

D.15 Plot Iterative Learning Control Problem

```
1 function [error_fig, out1_fig, out2_fig, dual_out_fig,
2     input1_fig, input2_fig] = plot_dual_ilc(figure_name,
3     graph_title, ILC_Trial, goal_output, goal_input, to_plot,
4     save_path)
5 %Plot the ILC Structure information, showing the input and output
6 %history
7 %for a two-spring-mass system through ILC trials
8 %Inputs:
9     %figure_name: string - name of the figure that is opened, or
10    %precursor to saved image
11    %graph_title: string - displayed title header on plot
12    %ILC_Trial: structure - containing iterations of coefficients
13    %goal_output: vector - y*, what we want to generate
14    %goal_input: vector - u*, what input gets us there
15    %to_plot: scalar/vector - indicate which trials or how many to
16    %plot
17    %save_path: string - where to save the generated plots to
18 %Outputs:
19     %error_fig: figure - handle to the error progression
20     %out1_fig: figure - handle to position of mass 1
21     %out2_fig: figure - handle to position of mass 2
22     %dual_out_fig: figure - handle to figure showing a 'shaped'
23     %output
24     %input1_fig: figure - handle to input 1
25     %input2_fig: figure - handle to input 2
26
27 %legend_alpha = 0.5;
```

```

21 subtitle_size = getappdata(groot, 'DefaultSubtitleFontSize');
22
23 if nargin < 7 %if no file save path
24     save_path = -1;
25 end
26 if nargin < 6
27     to_plot = 5; %default to plotting 5 trials
28 end
29 if nargin < 5
30     goal_input = -1;
31 end
32 if nargin < 4
33     goal_output = -1;
34 end
35
36
37 %Process ILC Strcture
38 num_trials = length(ILC_Trial);
39 if isscalar(to_plot) %check if we are passing in a count or
    specific trials to plot
40     trials_to_plot = floor(linspace(1, num_trials, to_plot));
41     trials_to_plot(end) = num_trials;
42 else
43     trials_to_plot = to_plot;
44 end
45
46 %There is potential to clean up the code here, but readability is
    more
47 %important

```

```

48 num_inputs = length(ILC_Trial(1).input);
49 num_outputs = length(ILC_Trial(1).output);
50
51 out1_ndx = 1:2:num_outputs;
52 out2_ndx = 2:2:num_outputs;
53
54 in1_ndx = 1:2:num_inputs;
55 in2_ndx = 2:2:num_inputs;
56
57 %Error Progression
58 error_progression = zeros(num_trials, 1);
59 for ndx = 1:num_trials %convert structure to array
60     error_progression(ndx) =
61         norm(ILC_Trial(ndx).output_error)/num_outputs; %normalize the
62         errors
63 end
64 error_fig = figure('Name', sprintf('%s - Error Progression',
65                     figure_name));
66 semilogy(1:num_trials, error_progression, 'r*', 'LineStyle', ':');
67 %semi log to show better convergance rate for small values
68 set(gca, 'YScale', 'log'); %force log scale (sometimes linear if
69         small range)
70 title('Error Magnitude');
71 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
72 xlabel('Trial Number')
73 ylabel('Normalized Amplitude')
74
75 %Mass 1 Position

```

```

71 out1_fig = figure('Name', sprintf('%s - Mass 1 Position',
72 figure_name));
73 legend_list = "[REMOVE ME]"; %teach it we're building strings
74 hold on;
75 for ndx = 1:length(trials_to_plot)
76     trial_num = trials_to_plot(ndx); %which trial of the process
77     is being plotted
78     stairs(0:(num_outputs/2-1),
79             ILC_Trial(trial_num).output(out1_ndx));
80     legend_list(end + 1) = sprintf('Trial %.d', trial_num);
81 end
82 if ~isscalar(goal_output) && (goal_output == -1)
83     stairs(1:((num_outputs-1)/2),
84             goal_output(out1_ndx(1:(end-1))), 'Color', [1, 0, 0],
85             'LineStyle', '--');
86     legend_list(end + 1) = 'Goal Output';
87 end
88 hold off;
89 legend(legend_list(2:end));%, "BackgroundAlpha", legend_alpha);
90 title('Mass 1 Position')
91 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
92 xlabel('Sample Number (k)')
93 ylabel('Position (m)')

94 %Mass 2 Position

95 out2_fig = figure('Name', sprintf('%s - Mass 2 Position',
96 figure_name));
97 legend_list = "[REMOVE ME]"; %reset legend list
98 hold on;

```

```

94 for ndx = 1:length(trials_to_plot)
95     trial_num = trials_to_plot(ndx); %which trial of the process
96     is being plotted
97     stairs(0:(num_outputs/2-1),
98            ILC_Trial(trial_num).output(out2_ndx));
99     legend_list(end + 1) = sprintf('Trial %.d', trial_num);
100 end
101
102 if ~isscalar(goal_output) && (goal_output == -1)
103     stairs(1:((num_outputs-1)/2),
104            goal_output(out2_ndx(1:(end-1))), 'Color', [1, 0, 0],
105            'LineStyle', '--');
106     legend_list(end + 1) = 'Goal Output';
107 end
108 hold off;
109 legend(legend_list(2:end));%, "BackgroundAlpha", legend_alpha);
110 title('Mass 2 Position')
111 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
112 xlabel('Sample Number (k)')
113 ylabel('Position (m)')
114
115 %Combined Outputs (for fun 2D shapes)
116 dual_out_fig = figure('Name', sprintf('%s - Shaped Output',
117                         figure_name));
118 legend_list = "[REMOVE ME]"; %reset legend list
119 hold on;
120 for ndx = 1:length(trials_to_plot)
121     trial_num = trials_to_plot(ndx); %which trial of the process
122     is being plotted

```

```

117 plot(ILC_Trial(trial_num).output(out1_ndx),
118 ILC_Trial(trial_num).output(out2_ndx));
119 legend_list(end + 1) = sprintf('Trial %.d', trial_num);
120 end
121 if ~isscalar(goal_output) && (goal_output == -1))
122 plot(goal_output(out1_ndx(1:(end-1))),
123 goal_output(out2_ndx(1:(end-1))), 'Color', [1, 0, 0],
124 'LineStyle', '--');
125 legend_list(end + 1) = 'Goal Output';
126 %Add a start/stop position indicator
127 scatter(goal_output(1), goal_output(2), 'filled', '>',
128 'MarkerFaceColor', [0, 1, 0])
129 scatter(goal_output(end - 1), goal_output(end), 'filled',
130 'hexagram', 'MarkerFaceColor', [1, 0, 0])
131 legend([legend_list(2:end), 'Start', 'Stop']);%
132 %BackgroundAlpha", legend_alpha);
133 end
134 hold off;
135 title('Shaped Outputs')
136 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
137 xlabel('Mass 1 Position (m)')
138 ylabel('Mass 2 Position (m)')
139 axis equal %shapes should be equally scaled
140
141
142 %Input 1
143 input1_fig = figure('Name', sprintf('%s - Input 1', figure_name));
144 legend_list = "[REMOVE ME]"; %reset legend list
145 hold on;

```

```

140 for ndx = 1:length(trials_to_plot)
141     trial_num = trials_to_plot(ndx); %which trial of the process
142         is being plotted
143         stairs(0:(num_inputs/2-1),
144             ILC_Trial(trial_num).input(in1_ndx));
145         legend_list(end + 1) = sprintf('Trial %.d', trial_num);
146 end
147 if ~isscalar(goal_input) && (goal_input == -1)
148     stairs(0:((num_inputs/2)-1), goal_input(in1_ndx), 'Color', [1,
149         0, 0], 'LineStyle', '--');
150     legend_list(end + 1) = 'Goal Input';
151 end
152 hold off;
153 legend(legend_list(2:end));%, "BackgroundAlpha", legend_alpha);
154 title('Input 1')
155 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
156 xlabel('Sample Number (k)')
157 ylabel('Input (N)')
158
159 %Input 2
160 input2_fig = figure('Name', sprintf('%s - Input 2', figure_name));
161 legend_list = "[REMOVE ME]"; %reset legend list
162 hold on;
163 for ndx = 1:length(trials_to_plot)
164     trial_num = trials_to_plot(ndx); %which trial of the process
165         is being plotted
166         stairs(0:(num_inputs/2-1),
167             ILC_Trial(trial_num).input(in2_ndx));
168         legend_list(end + 1) = sprintf('Trial %.d', trial_num);

```

```

164 end
165 if ~isscalar(goal_input) && (goal_input == -1)
166 stairs(0:((num_inputs/2)-1), goal_input(in2_ndx), 'Color', [1,
167 0, 0], 'LineStyle', '--');
168 legend_list(end + 1) = 'Goal Input';
169 end
170 hold off;
171 legend(legend_list(2:end));%, "BackgroundAlpha", legend_alpha);
172 title('Input 2')
173 subtitle(sprintf('%s', graph_title), 'FontSize', subtitle_size)
174 xlabel('Sample Number (k)')
175 ylabel('Input (N)')
176 %Save images if possible
177 if exist('save_path', 'var') %if a save path was provided
178 save_figure(save_path, [error_fig, out1_fig, out2_fig,
179 input1_fig, input2_fig]); %save all the figures to the path
180 save_figure(save_path, dual_out_fig, -1, true); %this one is
181 shapede
182 end
183
184 end

```

Listing D.15: Plot Histories of a Dual-Mass ILC System

D.16 Policy Learning for Iterative Learning Control

```
1 function [ILC_Trial, F_policy, controller_error, F_lqr] =
2     policy_ilc(P, d, C, D, x0, y_star, gamma, Q, R,
3     num_controllers, exploration_mag, input_basis_functions,
4     output_basis_functions, num_converged, existing_controller,
5     existing_T_u, existing_T_y)
6 %Perform the RL policy learning on an ILC system since done so
7 %much in
8 %thesis
9 %Inputs:
10 %P: matrix - inputs u(0->(p-1)) to outputs y(1->p)
11 %d: vector - noise/initial conditions matrix
12 %C: matrix - state to output descriptor
13 %D: matrix - input to output descriptor
14 %x0: vector - initial state
15 %y_star: vector - goal output
16 %gamma: scalar - discount factor
17 %Q: matrix - cost of states (errors)
18 %R: matrix - cost of inputs (change in inputs)
19 %num_controllers: scalar - number of controllers to learn
20 %exploration_mag: scalar - range around 0 to explore (defaults
21 to 1)
22 %input_basis_functions: matrix - basis functions on the inputs
23 (defaults to identity)
24 %output_basis_functions: matrix - basis functions on the
25 outputs (defaults to identity)
26 %num_converged: scalar - number of trials to simulate out
27 without exploration (defaults to 0)
```

```

19 %Outputs:
20
21     %ILC_Trial: structure with indexed by trial number, contains
22
23         %inputs
24
25         %betas (input basis weights)
26
27         %del_betas (change in input betas)
28
29         %outputs
30
31         %alphas (output basis weights)
32
33         %output error ( $y^*$  -  $y$ )
34
35         %alpha error
36
37     %F_policy: matrix - controller learning history
38
39     %controller_error: scalar - normalized error from LQR
40
41     %F_lqr: matrix - goal LQR controller
42
43
44 %Required Parameters to System Info
45
46 num_inputs = width(D);
47
48
49 %Default paramters
50
51 if ~exist('num_converged', 'var')
52
53     num_converged = 0; %default to no converged trials
54
55 end
56
57 if ~exist('exploration_mag', 'var')
58
59     exploration_mag = 1;
60
61 end
62
63 if ~exist('input_basis_functions', 'var')
64
65     input_basis_functions = eye(width(P));
66
67 end
68
69 if ~exist('output_basis_functions', 'var')
70
71     output_basis_functions = eye(height(P));
72
73 end

```

```

47     output_basis_functions_pinv = output_basis_functions;%save on
48     %compute time
49
50 else
51     output_basis_functions_pinv = pinv(output_basis_functions);
52 end
53
54 %Calculate optimal controller
55 F_lqr = discounted_LQR(eye(num_ilc_states),
56                         -output_basis_functions_pinv * P * input_basis_functions,
57                         gamma, Q, R);
58
59 alpha_star = output_basis_functions_pinv * y_star;
60
61 %Iteration Counts
62 Pj_dim = num_ilc_states + num_ilc_inputs;
63 num_collections_per_controller = Pj_dim^2;
64 total_trial_count = num_controllers *
65                         num_collections_per_controller + num_converged;
66
67 %Preallocate structure
68 ILC_Trial(total_trial_count).betas = [];%input
69 ILC_Trial(total_trial_count).betas = [];%basis representation of
    input
70 ILC_Trial(total_trial_count).del_beta = [];
71 ILC_Trial(total_trial_count).output = [];%output
72 ILC_Trial(total_trial_count).alphas = [];%basis representation
    of output

```

```

70 ILC_Trial(total_trial_count).output_error = []; %output error
71 ILC_Trial(total_trial_count).alpha_error = []; %alpha error
72
73 F_policy = zeros(num_ilc_inputs, num_ilc_states, num_controllers
74     + 1); %start with no controller
75
76 %Prepopulate the first trial
77 trial_num = 1;
78 ILC_Trial(trial_num).betas = zeros(num_ilc_inputs, 1); %start
79     with no basis guessed
80 ILC_Trial(trial_num).input = input_basis_functions *
81     ILC_Trial(trial_num).betas;
82
83 ILC_Trial(trial_num).output = [C*x0; d]; %open loop response is
84     IC and then d term
85 ILC_Trial(trial_num).alphas = output_basis_functions_pinv * d;
86 ILC_Trial(trial_num).output_error = y_star - d; %relevant error
87 ILC_Trial(trial_num).alpha_error = alpha_star -
88     ILC_Trial(trial_num).alphas;
89
90 trial_num = 2; %start at second trial now
91 for iteration = 1:num_controllers
92     Uk_stack = zeros(num_collections_per_controller, 1);
93     Xk_stack = zeros(num_collections_per_controller, (Pj_dim)^2);
94
95     %Simulate the necessary trials
96     for trial = 1:num_collections_per_controller %number of trials
97         to collect before updatin controller
98             %ILC / Basis Controller Process

```

```

93     %Beta Coeffcients
94
95     exploration_term = rand_range(num_ilc_inputs, 1,
96     -exploration_mag, exploration_mag); %jiggle to learn
97
98     ILC_Trial(trial_num).del_beta = F_policy(:, :, iteration) *
99     ILC_Trial(trial_num - 1).alpha_error + exploration_term;
100
101    ILC_Trial(trial_num).betas = ILC_Trial(trial_num - 1).betas
102    + ILC_Trial(trial_num).del_beta;
103
104    %Beta to Inputs
105
106    ILC_Trial(trial_num).input = input_basis_functions *
107    ILC_Trial(trial_num).betas;
108
109    %Simulate Reality
110
111    relevant_output = P * ILC_Trial(trial_num).input + d; %y(1)
112    -> y(p)
113
114    ILC_Trial(trial_num).alphas = output_basis_functions_pinv *
115    relevant_output;
116
117    ILC_Trial(trial_num).output = [C*x0 +
118    D*ILC_Trial(trial_num).input(1:num_inputs); relevant_output];
119
120    %total output y(0) -> y(p) for completeness
121
122    %Calculate Error
123
124    ILC_Trial(trial_num).output_error = y_star -
125    relevant_output;
126
127    ILC_Trial(trial_num).alpha_error = alpha_star -
128    ILC_Trial(trial_num).alphas;
129
130
131    %RL Translation
132
133    state = ILC_Trial(trial_num - 1).alpha_error; %analogous
134    state
135
136    input = ILC_Trial(trial_num).del_beta; %analogous input

```

```

110     next_state = ILC_Trial(trial_num).alpha_error; %x(k+1) =
111         e_alpha(j)
112
113     next_input = F_policy(:, :, iteration) * next_state; %no
exploration term here
114
115
116     Xu_stack = [state; input];
117     Xu_next_stack = [next_state; next_input];
118
119     Xk_stack(trial, :) = kron(xu_stack', xu_stack') - gamma *
120         kron(xu_next_stack', xu_next_stack');
121
122     Uk_stack(trial, :) = input' * R * input + state' * Q *
123         state;
124
125     trial_num = trial_num + 1;
126
127 end
128
129 %Calculate P and new controller
130 PjS = pinv(Xk_stack) * Uk_stack;
131 Pj = reshape(PjS, Pj_dim, Pj_dim);
132 Pj = 0.5 * (Pj + Pj'); %to impose symmetry (significantly
reduces error)
133 Pjuu = Pj((num_ilc_states+1):end, (num_ilc_states+1):end);
134 PjxuT = Pj((num_ilc_states+1):end, 1:num_ilc_states);
135 new_F = -pinv(Pjuu) * PjxuT;
136 F_policy(:, :, iteration + 1) = new_F;
137
138 end
139
140 controller_error = norm(F_policy(:, :, end) - F_lqr)/numel(F_lqr);
141
142

```

```

134
135 for ndx = 1:num_converged
136 %ILC / Basis Controller Process
137 %Beta Coeffecients
138 ILC_Trial(trial_num).del_beta = F_policy(:, :, iteration) *
139 ILC_Trial(trial_num - 1).alpha_error;
140 ILC_Trial(trial_num).betas = ILC_Trial(trial_num - 1).betas +
141 ILC_Trial(trial_num).del_beta;
142 %Beta to Inputs
143 ILC_Trial(trial_num).input = input_basis_functions *
144 ILC_Trial(trial_num).betas;
145 %Simulate Reality
146 relevant_output = P * ILC_Trial(trial_num).input + d; %y(1) ->
147 y(p)
148 ILC_Trial(trial_num).alphas = output_basis_functions_pinv *
149 relevant_output;
150 ILC_Trial(trial_num).output = [C*x0 +
151 D*ILC_Trial(trial_num).input(1:num_inputs); relevant_output];
152 %total output y(0) -> y(p) for completeness
153 %Calculate Error
154 ILC_Trial(trial_num).output_error = y_star - relevant_output;
155 ILC_Trial(trial_num).alpha_error = alpha_star -
156 ILC_Trial(trial_num).alphas;
157
158 trial_num = trial_num + 1;
159
160 end
161
162
163 end

```

Listing D.16: Policy Iteration for ILC

D.17 Generate Random Numbers in a Range

```
1 function [num] = rand_range(height, width, lower, upper)
2 %Return a random number in a range
3 %Inputs:
4     %height: scalar - height (num rows) of random vector
5     %width: scalar - width (num columns)
6     %lower: scalar - lower bound of numbers to generate
7     %upper: scalar - upper bound
8 %Outputs:
9     %num: matrix - height x width matrix of random numbers in range
10
11 num = lower + rand(height, width)*(upper - lower); %random times
12     %the amplitude, shift by lower bound
13
14 end
```

Listing D.17: Generate Random Matrices in a Range

D.18 Plot Poles

```
1 function [fig_pole] = plot_pole_placement(figure_name,
2                                         graph_title, pole_locations, goal_poles, save_path)
3 %Render the pole placements (or more accurately, plot eigen
4                                         values)
5 %Inputs:
6 %figure_name: string - name of the figure that is opened, or
7 %precursor to saved image
8 %graph_title: string - displayed title header on plot
9 %pole_locations: vector - location of poles
10 %goal_poles: vector - where we hoped/wanted poles (not
11 %rendered if excluded)
12 %save_path: string - location to save the figure
13 %Outputs:
14 %fig_pole: figure - handle to showing poles
15
16 if ((nargin < 4) || (all(goal_poles == -1))) %if there is no goal
17   poles
18   goal_poles = false;
19 end
20
21 fig_pole = figure('Name', figure_name);
22 circle_resolution = 200; %how many points to make up the circle
23 circle_domain = linspace(0, 2*pi, circle_resolution); %trace out
24                                         a full period
25 unit_x = cos(circle_domain);
26 unit_y = sin(circle_domain);
27 plot(unit_x, unit_y);
```

```

22 hold on;
23 scatter(real(pole_locations), imag(pole_locations), 'filled',
24     'o', 'LineWidth', 3, 'SizeData', 200);
25
26 xline(0)
27 yline(0)
28 if (goal_poles == false) %yes I know usually you can just do
29     ~goal_poles, but if this is a matrix that breaks it
30     legend('Unit Circle', 'Placed Poles')
31 else
32     scatter(real(goal_poles), imag(goal_poles), 'o', 'SizeData',
33         200);
34     legend('Unit Circle', 'Pole Locations', '', '', 'Goal Poles')
35     %skip the x and y line
36 end
37 hold off;
38
39 xlabel('Real')
40 ylabel('Imaginary')
41 title('Pole Locations')
42 subtitle(graph_title, 'FontSize', getappdata(groot,
43     'DefaultSubtitleFontSize'));
44 axis equal %ensure the circle looks like a circle
45
46 %Save images if possible
47 if exist('save_path', 'var') %if a save path was provided
48     save_figure(save_path, fig_pole, -1, true); %save all the
49     figures to the path, marking it as shaped

```

```
45 end  
46  
47 end
```

Listing D.18: Plot the Poles of a System

Bibliography

- Bristow, D.A., M. Tharayil, and A.G. Alleyne (2006). “A survey of iterative learning control”. In: *IEEE Control Systems Magazine* 26.3, pp. 96–114. DOI: 10.1109/MCS.2006.1636313.
- Frueh, J.A. and M.Q. Phan (1998). “Linear quadratic optimal learning control (LQL)”. In: *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*. Vol. 1, 678–683 vol.1. DOI: 10.1109/CDC.1998.760762.
- Moore, Kevin L., YangQuan Chen, and Hyo-Sung Ahn (2006). “Iterative Learning Control: A Tutorial and Big Picture View”. In: *Proceedings of the 45th IEEE Conference on Decision and Control*, pp. 2352–2357. DOI: 10.1109/CDC.2006.377582.
- Nguyen, Nhan (2013). “Least-Squares Model-Reference Adaptive Control with Chebyshev Orthogonal Polynomial Approximation”. In: *Journal of Aerospace Information Systems* 10.6, pp. 268–286. DOI: 10.2514/1.I010037. eprint: <https://doi.org/10.2514/1.I010037>. URL: <https://doi.org/10.2514/1.I010037>.
- Phan, M.Q. (n.d.). “Optimal State-Space System Identification and Control”. Writing in Process.
- Phan, M.Q. and J.A. Frueh (1996). “Learning control for trajectory tracking using basis functions”. In: *Proceedings of 35th IEEE Conference on Decision and Control*. Vol. 3, 2490–2492 vol.3. DOI: 10.1109/CDC.1996.573465.

Saab, Samer Said et al. (2022). “Iterative Learning Control: Practical Implementation and Automation”. In: *IEEE Transactions on Industrial Electronics* 69.2, pp. 1858–1866. DOI: 10.1109/TIE.2021.3063866.

Zhang, Yueqing, Bing Chu, and Zhan Shu (2019). “A Preliminary Study on the Relationship Between Iterative Learning Control and Reinforcement Learning”. In: *IFAC-PapersOnLine* 52.29. 13th IFAC Workshop on Adaptive and Learning Control Systems ALCOS 2019, pp. 314–319. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2019.12.669>. URL: <https://www.sciencedirect.com/science/article/pii/S2405896319326187>.