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

|  |
| --- |
|  |

An Honors Thesis  
in  
Engineering Sciences  
by Noah Dunleavy  
Thayer School of Engineering  
Dartmouth College  
Hanover, New Hampshire  
March 2025

Advisor (180, 0)110  
Professor Minh Q. Phan  
Author (180, 0)110  
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. 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. 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.

The come 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.

# List of Acronyms

# Introduction

## 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 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 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.

## Introduction to Continuous State Space

A key step for Control Theory is construction of a system model, commonly represented as , , and . captures the impact that the current state will have on the next state and captures how inputs will impact the next state. The matrix captures how states are translated to measured outputs and captures how inputs are directly measured on outputs. Any linear system, regardless of complexity and variations, can be modelled exactly as follows:

where , , , and are the matrices describing the system dynamics, and the terms represent noise (like a residual in a regression). is the state vector of dimensions , where is the number of states. There will be one state for every energy storing element in the system. is the input vector of dimensions , where is the number of inputs. is the output vector of dimensions , where is the number of outputs. As such, is , is , is and is . 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: and the same goes for , , and .

### 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 dampened, two-mass-spring system, as seen in Fig [1.1](#fig:spring_mass_system)

|  |  |
| --- | --- |
|  |  |

Dual Spring-Mass-Damper System

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 , employing Hooke’s and Damping Laws . Recognizing that the derivative of position is velocity, and the derivative of velocity is acceleration, it can be shown that the equations of motion (EoM) for each mass are as shown in Eqs. [[eq:x1\_eom]](#eq:x1_eom) and [[eq:x2\_eom]](#eq:x2_eom).

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:

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

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. [[eq:state\_vector\_derivative]](#eq:state_vector_derivative) below

With all this matrix information, it is now time to construct our continuous state-space model of the form seen in Eq. [[eq:continuous\_state\_space\_model]](#eq:continuous_state_space_model). Through matrix multiplication we arrive upon the following:

Recognizing this simple system is already messy, further substitutions can be made by formatting the masses, spring constants, and damping coefficients into Mass ([[eq:mass\_matrix]](#eq:mass_matrix)), Stiffness ([[eq:stiffness\_matrix]](#eq:stiffness_matrix)), and Damping ([[eq:dampning\_matrix]](#eq:dampning_matrix)) Matrices. These are known as physical parameter matrices.

This allows us to express our equations of motion in a single line as in Eq. [[eq:compact\_eom]](#eq:compact_eom)

and our state-space model can now be written as

This is exactly the format for the state-space model we described earlier in Eq. [[eq:continuous\_state\_space\_model]](#eq:continuous_state_space_model).

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

such that our physical parameter matrices are

Plugging these values into Eq. [[eq:spring\_mass\_state\_space\_continuous\_compact]](#X5f48a84d01e83c43d47227ea342472cac64adeb), we can write our system model as

To explicitly complete the connection to Eq. [[eq:continuous\_state\_space\_model]](#eq:continuous_state_space_model), we can see

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.

Armed with the information to completely model the system, we now decide on our outputs. Recall Eq. [[eq:continuous\_state\_space\_output]](#eq:continuous_state_space_output) relating the current state and inputs to the output , via matrices and . For our system we will choose to only record the block positions ( and ) as our outputs. Monitoring those two states is represented simply in matrix form

where we will once again explicitly note

The resulting outputs from simulating out this system can be seen in Fig. [1.2](#fig:continuous_open_mass_1) and Fig. [1.4](#fig:continuous_open_mass_2)

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Position data from a open-loop continuously-modelled Dual-Spring-Mass system

## 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 (typically in units of seconds). That is, every a new output is collected and input applied. To maintain the exact nature of the above model, matrices and must be ‘discretized’ through the ZOH method. The equations for doing so are shown in Eqs. [[eq:Ac\_to\_A]](#eq:Ac_to_A) and [[eq:Bc\_to\_B]](#eq:Bc_to_B)

What this relationship now constitutes is rather than applying instantaneous inputs, inputs are applied every and held for . The response of the system between samples is not fully captured in the model, but at every discrete time step the model-to-nature relationship is exact. The output collection matrices and do not need to be adjusted. We re-write our continuous state-space models from Eqs. [[eq:continuous\_state\_space\_model]](#eq:continuous_state_space_model) and [[eq:continuous\_state\_space\_output]](#eq:continuous_state_space_output) as

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

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 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.

### Example — Discretization

Continuing with our earlier model, we now seek to discretize it for practical computational techniques. We will set seconds, and apply equations [[eq:Ac\_to\_A]](#eq:Ac_to_A) and [[eq:Bc\_to\_B]](#eq:Bc_to_B). The resulting discrete matrices are

Note that in this perfect information scenario, we can verify our sufficient by examining the continuous-time system matrix . The imaginary components of the eigenvalues are the natural frequencies of the system that 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. [[eq:discrete\_state\_space\_model]](#eq:discrete_state_space_model). Running that out and overlaying it with the results of the continuous system we arrive upon outputs depicted in Figures [1.5](#fig:discrete_open_mass_1) and [1.7](#fig:discrete_open_mass_2).

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

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 [1.8](#fig:zoomed_discrete_open_mass_1) and [1.10](#fig:zoomed_discrete_open_mass_2) 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.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Zoomed-in view of the discrete-continuous model to show that at the discretely modelled 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 . For example, Figure [1.8](#fig:zoomed_discrete_open_mass_1) axis will be marked with intervals 0 through 10, as opposed to time 0 through 0.1 seconds.

## 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 will be of dimensions 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):

Following this control law, the state space equation in Eq. [[eq:discrete\_state\_space\_model]](#eq:discrete_state_space_model) can be re-written as:

A controlled system means that , so as goes to infinity, we would like to go to zero. Any formulation of that causes to be smaller in magnitude than will eventually result in an [[1]](#footnote-39). In the scalar case this is easy to understand; suppose

Where , then . 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 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 and matrices, it is relevant to check if it is controllable. That is – is it possible to actually send all the states to zero, with our selected inputs. This can be checked by examining the Controllability Matrix. Defined as

If the Controllability Matrix 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 another[[2]](#footnote-40). 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 matrix will have rank of 2.

### Example — Basic Control with Pole Placement

There is an infinite number of controllers that could send our system to stability. The closer the poles of 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 [1.11](#fig:simple_poles). 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.

|  |
| --- |
|  |

Pole locations for a Dual-Spring-Mass system when manually placed at locations and

The poles, placed at and result in controller

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 as shown in Eq. [[eq:initial\_conditions\_real]](#eq:initial_conditions_real). The first row of the controller dictates how our first input ( on ) is computed given current states (see Eq. [[eq:initial\_conditions\_real]](#eq:initial_conditions_real)). In this case

The same process can be repeated for and for any sample number . 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 is (recall ‘control’ is a zero state), will generate -40,364 units of input. The net input will be the summed effects of each state. When the controller in Eq. [[eq:simple\_pole\_controller]](#eq:simple_pole_controller) is applied to our dual-mass system, our system produces outputs seen in Figs. [1.12](#fig:simple_pole_mass_1) and [1.13](#fig:simple_pole_mass_2), under the inputs shown in Figs. [1.14](#fig:simple_pole_input_1) and [1.16](#fig:simple_pole_input_2)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Position of Mass 1 and Mass 2 for the Dual-Spring-Mass system under closed-loop, state-feedback controller . is designed such that the poles of () are at and . Inputs 1 and 2 are generated as

 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 origin[[3]](#footnote-47) and produces the time-optimal solution. Once again, the system poles are shown in Figure [1.17](#fig:deadbeat_poles)

|  |
| --- |
|  |

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

and produces controller

Its application results in the outputs and inputs shown in Figures [1.18](#fig:deadbeat_mass_1) - [1.22](#fig:deadbeat_input_2)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Position of Mass 1 and Mass 2 for our Dual-Spring-Mass system under a deadbeat closed-loop, state-feedback controller . is designed such that the poles of () are at . Inputs 1 and 2 are generated as . Under deadbeat control, it can be seen that control is achieved under steps

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.

## 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: , , and . is an matrix which applies relative ‘costs’ (or rewards as desired) to states of the system. Similarly, is a matrix which weights the inputs. 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’).

and 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 ( and ). We will only use the identity approach. What each component of the cost matrices and 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 and will be the exact same as one defined by and .

Each sample, , 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 which produces such a scalar value.

In our journey to control, we will work through multiple time steps, each with their own , so in the whole process we will incur some net cost, . The cost can be viewed as the summation of all these utilities along the infinite horizon.

Bringing back the aforementioned discount factor , we modify the cost function such that we can adjust the time horizon of consequence. So long as , we can introduce it such that as goes to , the impact of the infinite horizon utility reduces to zero[[4]](#footnote-56). This is additionally useful as we want to be able to induce stability in finite-time. This presents us with our discounted cost function

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

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 [1.23](#fig:principle_optimality) 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 contradicted.

|  |
| --- |
|  |

Illustration of Principle of Optimality, with nodes and with paths between labelled with their associated costs. Even though to go from is only a cost of , costs making the total path cost of . Also see that the path of may have a final step of cost , but the first step has a cost pf . The central path through then could result in costs of or . Clearly going through is the optimal way to proceed. It can then further be seen that the optimal way to go from is a subset of the optimal path of .

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. 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:

Where ,, and is the solution to the algebraic Riccati equation associated with the un-discounted LQR problem

### Example — LQR

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

Using the attached function *discount\_LQR*[[5]](#footnote-58) will find the correct controller for the cost function (and discount factor) we use. Applying that function to our discrete system, we find

Examining where that places the poles (Figure [1.24](#fig:big_Q_poles)) and the input-output data (Figures [1.25](#fig:big_Q_mass_1) - [1.29](#fig:big_Q_input_2)), we see that control takes almost two seconds, but inputs are magnitudes smaller than that seen by the pole placement controllers.

|  |
| --- |
|  |

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

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 by a factor of ten (or by a factor of 0.1), then the following controller (Eq. [[eq:big\_R\_lqr\_ex]](#eq:big_R_lqr_ex)), poles (Figure [1.30](#fig:big_R_poles)), and IO data (Figures [1.31](#fig:big_R_mass_1) - [1.35](#fig:big_R_input_2)) would result:

|  |
| --- |
|  |

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

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 , the closer our poles get to the origin of the complex plane. It is important to note that does not result in a deadbeat controller. Given that every state is weighted equally, and velocity is a state, a 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 with no velocity – incurring a cost of . A deadbeat controller would attempt to bring shift the mass such that in one step , over a period of seconds. A change is position of m over seconds implies a velocity, or , of . No controller that views position and velocity as equally expensive would make this decision.

## 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 known[[6]](#footnote-74).

Iterative Learning Control employs a system representation that is expanded to factor in the temporal element of control steps. Instead of each step () trying to send the states to zero, we now want each trial () to send the error on our outputs to zero. The first step is to define our output, denoted as , occurring over p time steps. That is, is a column vector. Similarly, there is then a sequence of inputs, denoted as that when applied, will get us here – it will be .

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

where captures disturbances and initial conditions. All the above matrices are formulated as such

See that the vector factors in both how the initial conditions propagate through the system and how any additional noise play a role. Also note that in the ILC process we do not try to control 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 . From Eq. [[eq:y\_Pu\_d]](#eq:y_Pu_d), there is a sequence of inputs that gets us our goal output. Call this .

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

Applying this operator to Eq. [[eq:y\_Pu\_d]](#eq:y_Pu_d)

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

Next, we define error. Each trial () will produce an output that will be off from our goal out of by an error denoted as

Applying the operator to this equation

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

which expands to

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

Through re-arrangement of Eq. [[eq:e\_j\_1\_e\_law]](#eq:e_j_1_e_law) and substitution of Eq, [[eq:del\_y\_P\_del\_u]](#eq:del_y_P_del_u), we arrive upon the ILC Equation

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

Where is (or ). As we have already explored the ideas of controllers, it logically follows there are an infinite number of these controllers, all facing tradeoffs.

### Example — ILC

The first step in setting up an ILC problem is to establish the goal, or . For simplicity, we will work through this example trying to draw a circle. That is, would ideally trace out one period of a cosine, and will follow one period of a sine wave. We can set the resolution of this circle by our choice of . Supposed we set , meaning we want to draw a circle over discrete time steps. We will define the goal for our first output (the position of ) as and the second output (position of ) as :

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

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 determining the location of our poles however, it is . By selecting a controller to be (where + denotes the pseudo inverse operation and ), we can guarantee such pole placement. For the presented system, we select , which we will apply for just 10 trials

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 [1.36](#fig:ilc_error) that the error rapidly drops to zero.

|  |
| --- |
|  |

Error Progression of an ILC problem when using a perfect knowledge controller such that the poles of the system under () are guaranteed to be within the unit circle and relatively close to the origin for rapid convergence.

Further showing the progression of the individual outputs and inputs (Figures [1.37](#fig:ilc_mass_1) - [1.41](#fig:ilc_input_2)) as well as the shaped output in Figure [1.42](#fig:ilc_shaped_circle) (where ’s position is the x-axis and ’s position is the y-axis), you can see as the system ‘learns’ to draw the circle. Trial 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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

The progression of Input-Output trials under our controller of . 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

|  |
| --- |
|  |

Progression of shaped outputs (where Mass 1 Position is the x-coordinate and Mass 2 Position is the y-coordinate) under our ILC controller

 To demonstrate ILC’s ability to learn arbitrary shapes[[7]](#footnote-82), a point resolution, mouse-drawn ‘Dartmouth’ is introduced as the . Utilizing the same controller shown in Eq. [[eq:ilc\_controller]](#eq:ilc_controller), 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 and going for the green play/triangle symbol at the top of the ‘D’ where the goal output begins.

|  |
| --- |
|  |

Application of ILC Controller 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

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of Mass 1 and 2 Positions under controller , 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

## Reinforcement Learning

Until now we have been assuming that we always know the , and 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. [[eq:discounted\_cost\_function]](#eq:discounted_cost_function). We can choose to restrict our time horizon down from infinity, and now look steps ahead. We create a cost-to-go function :

As from our principle of optimality, it similarly follows that whatever controller minimizes will then also minimize . This agrees with an important feature of the cost-to-go function, and that is recurrence. If we multiply Eq. [[eq:cost\_to\_go]](#eq:cost_to_go) by and increment by 1, we get

Substituting Eq. [[eq:gamma\_inc\_cost\_to\_go]](#eq:gamma_inc_cost_to_go) into [[eq:cost\_to\_go]](#eq:cost_to_go), the relationship between and can be shown

This is known as the recurrence equation. So long as and is sufficiently large[[8]](#footnote-89),

It can then be useful to express in a supervector format for later descriptions and derivations. Substituting the utility function described in Eq. [[eq:utility\_function]](#eq:utility_function) into Eq. [[eq:expanded\_recurrence]](#eq:expanded_recurrence)

And we can define supervectors for state and input histories

It is further beneficial to define matrices and , where Q and R are and block-diagonal matrices comprised of of the cost-defining matrices and (respectively)

and the matrices are defined as

Now Eq. [[eq:supervector\_expanded\_ctg]](#eq:supervector_expanded_ctg) can be re-written as

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 . Its logic is as follows: suppose we are at state and have just made input – both can be arbitrary. However, from time k+1 to infinity, our next state will be a function of our previous state and input, as described in Eq. [[eq:discrete\_state\_space\_model]](#eq:discrete_state_space_model). And our inputs will follow the control law described in Eq. [[eq:control\_law]](#eq:control_law). Thus the vector can be expressed in terms of the current state , the current input , and all future inputs to as

where

Substituting these into our cost-to-go expression of Eq. [[eq:supervector\_ctg]](#eq:supervector_ctg), we get a cost-to-go defined only in terms of present state and all inputs from now until

If we define the symmetric matrix as

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

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

Given our system model and control law, all future inputs can be tracked back to and . By repeated substitution, it can be shown

We can now rewrite [[eq:u\_s\_and\_u\_s\_1]](#eq:u_s_and_u_s_1) as

where

Then

Defining **R**

We can now write

If we create matrix[[9]](#footnote-90) P = **SR**, we can substitute Eq. [[eq:xus\_to\_Rxu]](#eq:xus_to_Rxu) into Eq. [[eq:xus\_cost\_to\_go]](#eq:xus_cost_to_go) to get a new cost-to-go function

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 , and we constructed 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 will produce that minimizes our cost-to-go, now defined in Eq. [[eq:q\_function]](#eq:q_function), starting from . That is, it will give us input defined as:

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 that will minimize the cost-to-go function is returned. Therefore it follows that the Q-function that utilizes this minimizing input will equal the minimum possible value for the Q-function

So we can now write the recurrence equation for the deterministic Q-Learning-based RL method. By taking Eq. [[eq:recurrence]](#eq:recurrence), substituting in our new cost-to-go defined in Eq. [[eq:q\_function]](#eq:q_function) and the logic in Eq. [[eq:q\_function\_min\_recurrence]](#eq:q_function_min_recurrence), we arrive upon the relationship that the optimal Q-function (as defined by the optimal controller) must satisfy:

Any Q-function will satisfy

but only Eq. [[eq:optimal\_q\_recurrence]](#eq:optimal_q_recurrence) is satisfied by the optimal Q-function, as defined by the optimal controller .

To extract the controller from a given Q-function, we return to Eq. [[eq:q\_function]](#eq:q_function). Recall matrix **P** is symmetric. We can re-write it as

Where the and subscript indicate the size of each **P** component. **P** is , so refers to the top-left portion, and the same logic follows for the other components. We can extract controller as

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.

### Policy Iteration

The first method which we will demonstrate is that of Policy Iteration. To do this, we need data triplets of and . By manipulating enough of these triplets, we can episodically solve for the **P** that parametrizes a Q-function, while simultaneously optimizing the Q-function. We begin with Eq. [[eq:q\_recurrence]](#eq:q_recurrence), but re-arrange it to

It is next necessary to find a way to re-write . We will start by defining the stack operator for a matrix. Given an arbitrary matrix that is , the stack operator creates that is a single column, making our matrix . So if

Where each is , then

It is important to recognize that the stack operator is not the transpose operation, as each is a vector, not a scalar. Our next operator is the Kronecker product, marked . 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 matrix , and a matrix

The Kronecker product between the two will create an matrix

Now to see how these operators will be useful, we return to Eq. [[eq:q\_function]](#eq:q_function), re-written below as

$$\begin{gathered}
Q\left(x\left(k\right),u\left(k\right)\right)={\left[\begin{matrix}x\left(k\right)\\u\left(k\right)\\\end{matrix}\right]}^T\textbf{P}\left[\begin{matrix}x\left(k\right)\\u\left(k\right)\\\end{matrix}\right] \tag{\ref{eq:q\_function}}
\end{gathered}$$

To better demonstrate the process, we will assume and are scalar, and **P** is thus a matrix.

Manually working out Eq. [[eq:q\_function]](#eq:q_function), we can re-write it as

It can be shown that the results of Eq. [[eq:manual\_Q\_ex]](#eq:manual_Q_ex) can then be expressed by stacking **P** as

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

Putting it all together, we can re-write Eq. [[eq:manual\_Q\_ex]](#eq:manual_Q_ex) as

This was just demonstrated in the scalar case, but can be similarly proven for when is and is . Thus we can write our Q-Function as

For controller , we have a given Q-function parametrized by . Recall that and can be arbitrary[[10]](#footnote-91), will be produced by nature/the system, and all inputs from onward are defined by our control law. Thus we can re-write Eq. [[eq:q-gamma\_q]](#eq:q-gamma_q) as

To simplify equations, write

will then be dimensions . From each , and triplet, we can produce an and pair. Stacking those we can construct

Where do not need to be consecutive (but logically will be). With sufficient data samples, we can then solve for as

In the noise free scenario, we need collections to solve for . 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 , we can update controller using Eq. [[eq:F\_from\_P]](#eq:F_from_P) such that iteratively

Since Eq. [[eq:XjPjU]](#eq:XjPjU) is derived from the optimal condition outlined in Eq. [[eq:optimal\_q\_recurrence]](#eq:optimal_q_recurrence), the controller we derived must also be optimal. It may take several iterations, but this process will alternate updating **P** and until the optimal controller is found.

#### Example — Policy Iteration

We now return to our earlier spring-mass system shown in Figure [1.1](#fig:spring_mass_system). Operating off the same parameters outlined in Eqs. [[eq:LQR\_params\_SMD]](#eq:LQR_params_SMD) which produced the shown in Eq. [[eq:F\_lqr]](#eq:F_lqr) as

$$F\_{LQR}^\gamma=\left[\begin{matrix}9.9546&-1.8952&-2.6156&-0.3501\\-25.2185&29.3267&-0.8026&-3.767\\\end{matrix}\right]
\tag{\ref{eq:F\_lqr}}$$

Our system has four states and two inputs . As such, each will be . That means that each controller ‘update’ marked by the process shown in Eq. [[eq:PjXjU]](#eq:PjXjU) requires 36 triplets of state, input, and next state data . 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 . When learning, it is not enough to just use our classic ; we must add some random excitation term for exploration. The mathematical reason for this is the ensure that we construct sufficient linear-independent s to allow for a proper solving of . Intuitively - how can one expect to learn by not trying something new every now and again?

Where is some exploration term intentionally added to the classic . 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

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 .

Whichever input approach we chose, we then apply it to the system in state . Nature then produces for us. We now have our , , triplet. Following Eq. [[eq:Xj]](#eq:Xj) we formulate . Note that the term does not include an exploration term. At the same time, we will compute the utility as defined in Eq. [[eq:utility\_function]](#eq:utility_function)

For our system, we will repeat this process 35 more times before we can update the controller once. We compute from Eq. [[eq:PjXjU]](#eq:PjXjU), and undo the stack operator by reshaping it into a matrix. The way in which we do this does not matter (rows to column or column to rows) as is symmetric. Numerical operators are not exact, however, and we can accelerate the learning process by imposing symmetry. That is, after computing a which is semi-symmetric, we set as

Now we refer to Eq. [[eq:F\_from\_P]](#eq:F_from_P) to extract the components to solve for our next controller as shown in Eq. [[eq:F\_from\_P\_iterative]](#eq:F_from_P_iterative). In this example, we grab the bottom right block of as , and the bottom left block as . 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

which matches our exactly to at least 4 decimal places. Its application can be seen in Figures [1.47](#fig:policy_mass_1) - [1.51](#fig:policy_input_2)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

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 . After 5 controllers, the learning stops and exploration is no longer applied to the input for the final 20 trials.

Notice how even for the first 36 trials there is variation on the input due to the exploration term, and the final 20 trials are much smoother as they follow a strict control law without exploration. In Figures [1.52](#fig:policy_F1_history) and [1.54](#fig:policy_F2_history) 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 .

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

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

### 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 to , but at the cost of needing to complete times as many learning trials. It can be easily shown that when , 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. [[eq:q\_function]](#eq:q_function))

$$Q\left(x\left(k\right),u\left(k\right)\right)={\left[\begin{matrix}x\left(k\right)\\u\left(k\right)\\\end{matrix}\right]}^T\textbf{P}\left[\begin{matrix}x\left(k\right)\\u\left(k\right)\\\end{matrix}\right]
\tag{\ref{eq:q\_function}}$$

For our -input problem, we can express as a stack of each individual input

Which can be produced by our similarly-represented stacked controller

Where each is a vector. In general, for input , we can re-write Eq. [[eq:stacked\_inputs]](#eq:stacked_inputs) and [[eq:stacked\_controllers]](#eq:stacked_controllers) as

Where subscripts and represent the top elements and bottom elements of and . , by definition. Defining matrix

We can write the state-input stack as:

By defining a **P** akin to the one from Eq. [[eq:sym\_P\_rl]](#eq:sym_P_rl)

We can write an input-decoupled Q-function for as

Once again we have a function of current state , but now the only concern is a single input variable . In our previous Q-function the controller that was built in was ; now we are looking for a controller . Just as we did for Policy Iteration, we now define a recurrence equation like that seen in Eq. [[eq:q\_recurrence]](#eq:q_recurrence).

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 equation. That is

The analogies to Policy Iteration continue where instead of Eq. [[eq:q\_function]](#eq:q_function) we now have

and instead of Eq. [[eq:sym\_P\_rl]](#eq:sym_P_rl)

Where the associated with is captured as it is in Eq. [[eq:F\_from\_P]](#eq:F_from_P)

A similar stacking computation as seen in Eq. [[eq:stacked\_Xj\_jU]](#eq:stacked_Xj_jU) can be done, except now we use , defined as

Note that we still compute in the complete form, using all the inputs on the system not just the current one of interest (). We can solve for in the exact same manner as we do in Eq. [[eq:F\_from\_P\_iterative]](#eq:F_from_P_iterative), with the iterative equation

#### Example — Input Decoupling

Once again we turn to the system in Figure [1.1](#fig:spring_mass_system), Eqs. [[eq:LQR\_params\_SMD]](#eq:LQR_params_SMD) which produced the shown in Eq. [[eq:F\_lqr]](#eq:F_lqr)

$$F\_{LQR}^\gamma=\left[\begin{matrix}9.9546&-1.8952&-2.6156&-0.3501\\-25.2185&29.3267&-0.8026&-3.767\\\end{matrix}\right]
\tag{\ref{eq:F\_lqr}}$$

As before, our system has four states and two inputs but we only learn one input at a time now. So each will be 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 , but only learn on . So , but then we modify input as

Where is some random value. We could also purely randomize our input as

We then apply it to the system to produce . With our , triplet, following Eq. [[eq:Xij]](#eq:Xij) we formulate . At the same time, we will compute the utility as defined in Eq. [[eq:utility\_function]](#eq:utility_function). For our system, we will repeat this process 24 more times before we can update the controller once. We compute from Eq. [[eq:PjXjU]](#eq:PjXjU) (using in place of ), and undo the stack operator by reshaping it into a matrix. Symmetry is imposed once again.

Refer to Eq. [[eq:Pi\_matrix]](#eq:Pi_matrix) to extract the components to solve for our next controller as shown in Eq. [[eq:Fi\_from\_Pi\_iterative]](#eq:Fi_from_Pi_iterative). In input decoupling, we grab the bottom right scalar of as , and the bottom left block as . 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. [[eq:F\_input\_decoupled]](#eq:F_input_decoupled)

Which once again matches our LQR controller exactly. The learning process and application can be seen in Figures [1.55](#fig:id_mass_1) - [1.59](#fig:id_input_2)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

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.

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 [1.60](#fig:id_F1_history) and [1.62](#fig:id_F2_history) 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.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

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.

## 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.

# Methods and Experimentation

## 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 approach. How do these two methods behave when combined?

Recall our earlier definition of the ILC system:

$$e\_{j+1}=Ie\_j-P\delta\_{j+1}\underline{u}
\tag{\ref{eq:ILC\_law}}$$

And its controller

$$\delta\_{j+1}\underline{u}=\mathcal{L}e\_j
\tag{\ref{eq:del\_u\_L\_e\_j}}$$

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

$$x\left(k\ +\ 1\right)=Ax\left(k\right)+Bu\left(k\right)
\tag{\ref{eq:discrete\_state\_space\_model}}$$

$$y\left(k\right)=Cx\left(k\right)+Du\left(k\right)
\tag{\ref{eq:discrete\_state\_space\_output}}$$

Where our ‘state’ is now the error and our ‘input’ is the change in inputs . 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’ () is now a vector, and our ‘ILC Input’ () is . 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 demarked , that sends our error state to zero. Thus our utility function that defines the cost function that the found controller will minimize is

### Example — Policy Iteration on ILC

We first begin by setting the number of steps in our process. We will work with a (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

$$y\_1^\ast=\cos\left(\frac{2\pi k}{p}\right)
\quad
y\_2^\ast=\sin\left(\frac{2\pi k}{p}\right)
\tag{\ref{eq:y1\_y2\_star}}$$

stacked as in Eq. [[eq:stacked\_y\_star]](#eq:stacked_y_star)

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

Using the complete knowledge afforded to us in simulation, we can find the controller

Given our ILC state-input dimensions, will be . This does not mean 16 seconds of data (from our systems seconds), but it is 1,600 trials of 10 steps (), 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 is equal to the noise and initial conditions parameter (from Eq. [[eq:y\_Pu\_d]](#eq:y_Pu_d)). For each trial, we compute the change in inputs,

where is our exploration term. In this case, is normally distributed around 0 and covers ranges [-1, 1]. We then compute and apply the system input as . Keep in mind the distinction between the input to the system, , and the input to the ILC learning of .

Upon applying , our system will produce a sequence of outputs. Recall that as we cannot control , we exclude it from our model such that

Computing from , 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

So our is

We repeat the ILC process until we have enough trials to solve for our , unstack it, impose symmetry, and update our controller as

Where is now the bottom-right matrix in , and the bottom-left . For the listed example, we repeat this process 4 more times (for a total of 5 controllers). Our end controller is

With an error magnitude of , computed as

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 [2.1](#fig:policy_ilc_F1)- [2.3](#fig:policy_ilc_F20).

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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

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 – each controller update requires 1600 ILC trials

|  |
| --- |
|  |

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Input-Output Progressions through Policy Iteration Trials of . Observe Trial 1 to be the open-loop response, and subsequent trials to be progressing towards the goal output.

|  |
| --- |
|  |

Shaped Output through Policy Iteration Trials of . 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 does not remove the existence of an LQR controller but introduces issues when computing the pseudo-inverse of . It is a requirement for to be well-conditioned and composed of sufficient linearly independent equations (full rank). When learning in ILC with a small , we approach a point where both and are near-zero and we lose the ability to even generate linearly-independent 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 to zero, meaning it no longer needs to generate a change in inputs – our is then essentially all zeros.

If one wishes to reduce , 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 , , triplets – this can also be accmplished with purely randomized inputs. The final option, and the one shown below, is to increase the magnitude of your exploration term. Whereas previous 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. [[eq:y1\_y2\_star]](#eq:y1_y2_star), but redefining our cost-matrices as

We have the new LQR Controller

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

With an error magnitude of , computed as in Eq. [[eq:controller\_error\_calc]](#eq:controller_error_calc). We see a much sharper error progression in Figure [2.4](#fig:reduced_R_ilc_error) 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 , 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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Select Controller Weights on Select Inputs progression through Policy Iteration Trials when . Due to the necessary amplified exploration , observe the tendency for weights to converge much less smoothly than before, displaying behaviors of overshoot and lag.

|  |
| --- |
|  |

Error Magnitude of Output through Policy Iteration Trials with . Observe how initial errors creep up much further than the earlier shown 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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Progression of Input-Output Data in a Dual-Spring-Mass system under ILC derived from RL when

|  |
| --- |
|  |

Shaped Output through Policy Iteration Trials when

 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 [2.5](#fig:shaped_rl_small_R). 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 . What if we were to increase this to 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 would go from to . just went from having elements - already a lot - to . Monetarily and computationally this makes the RL approach very costly. In fact, if we set , 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 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.

### Example — Input Decoupling on ILC

We have already shown that Input Decoupling can be used to reduce the dimensions of from to , 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 ).

However while it will take iterating through all 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. [[eq:y1\_y2\_star]](#eq:y1_y2_star) and [[eq:rl\_ilc\_params]](#eq:rl_ilc_params)), we predictably find the same controller (see Eq. [[eq:ilc\_lqr\_controller]](#eq:ilc_lqr_controller)).

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 , we modify only one of the terms with our exploration, such that

Now our ILC to Input Decoupling analogies are

So our is

We repeat the ILC process until we have enough trials to solve for our , unstack it, impose symmetry, and update our controller as

Where is now the bottom-right scalar in , and the bottom-left . 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

Which has an error of , as computed in Eq. [[eq:controller\_error\_calc]](#eq:controller_error_calc). We show the learning process below, once again only showing a few select weights and inputs of the controller.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Select Controller Weights on Select Inputs through Input Decoupling Trials to learn the ILC Controller when . Notice how each controller update takes ILC trials, and each input controller is only updated at that rate.

|  |
| --- |
|  |

Error Magnitude of Output through Input Decoupling Trials

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Input-Output Data progression through Input Decoupled Learning trials when

|  |
| --- |
|  |

Shaped Output through Input Decoupling Trials when .

 The input decoupling approach ends up suffering from the same issues as the policy iteration approach, and then some. We could solve the same 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.

### 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.

## 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. - Learning Control for Trajectory Tracking using Basis Functions - provides the logic from which we launch our approach.

### What are Basis Functions

Basis Functions offer the ability to represent a signal as a weighting of composite signals. Suppose you had the vector

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

you could capture Eq. [[eq:example\_vector\_basis]](#eq:example_vector_basis) exactly as .

This is the premise behind basis functions. For any signal of length (or ‘resolution’) , we can describe it as a composite of functions that are defined for points. We then create a ‘basis space’ out of basis functions , as shown in Eq. [[eq:basis\_space\_phi]](#eq:basis_space_phi)

where each is a () vector. The only condition on each basis functions is that it is independent of / orthogonal to any other basis functions.

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

To represent a signal in terms of basis functions, we then use a weighting vector .

In the ideal scenario, a signal can be captured with a single . This happens when the signal we are representing is a scalar of the chosen basis function, as shown in Eq. [[eq:example\_vector\_basis]](#eq:example_vector_basis). 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 identity matrix is implicitly being used, and thus basis functions. If we were to re-express Eq. [[eq:example\_vector\_basis]](#eq:example_vector_basis) in the worst case, then it would at most take five basis weights

The Basis Space also does not necessarily need to be . So long as it is full rank, a basis space of 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 . 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.

To summarize, any signal can be expressed exactly as a product of a basis space and basis coefficients , so long as exists in the basis space.

The trick then comes down to picking the right basis functions, and ensuring 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.

### Chebyshev Polynomials

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

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 shown in Eq. [[eq:cheby\_recurrence]](#eq:cheby_recurrence)

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 , that is where we will restrict ourselves for their generation. Recalling we wish to capture a signal of points, we will then define our from in steps of . This can be accomplished in Matlab by setting our to x = linspace(-1, 1, )’[[11]](#footnote-128) – this is our (Eq. [[eq:cheby\_T1]](#eq:cheby_T1)). We must similarly create a of all 1s to serve as our (Eq. [[eq:cheby\_T0]](#eq:cheby_T0)). To generate functions, we then iterate as described in Eq. [[eq:cheby\_recurrence]](#eq:cheby_recurrence). The process can also be seen in Code Appendix [8.7](#code:gen_cheby)

In the presented example, we set and , generating a that is . Figure [2.6](#fig:example_chebys) shows our , , and some select iteratively generated functions. You will see that the x-axis units are ‘Chebyshev Steps’ that go from , even though we numerically said they had to go from . Numerically the functions must be defined in this range, but for application the can be viewed purely as data points along a signal of any length, and the real values of and no longer hold any meaning.

|  |  |
| --- | --- |
|  |  |

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 [2.7](#fig:example_cheby_signal). This is a 100 time step signal created with just 20 chebyshev polynomials, weighted as shown in Figure [2.8](#fig:example_cheby_weights)

|  |  |
| --- | --- |
|  |  |

Example Signal constructed from Chebyshev Polynomials of length

|  |  |
| --- | --- |
|  |  |

Example Chebyshev Weights to Generate Signal in Fig. [2.7](#fig:example_cheby_signal)

### 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:

$$\underline{y} = P\underline{u} + \underline{d}
\tag{\ref{eq:y\_Pu\_d}}\\$$

For complete flexibility, we will assign separate basis spaces to the output () and our inputs (). The basis space will be and the will be . The will get to keep the basis coefficients convention of shown in Eq. [[eq:basis\_definition]](#eq:basis_definition), but the space will be expressed in terms of weights. We will have basis coefficients on the output, and basis coefficients on the input. So will be and will be .

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

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

It is important to note that when our basis spaces are not well-defined, these reverse operations become approximations. gives us the projection of into the basis space defined by – the same goes for and . 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 onto tells us how much of can be described in . However this ‘shadow’ does not tell us if there is some data behind our proverbial tree.This allows multiple s and s to produce the same and , so long as the differences in the s and s occur in the null space (outside) of their respective basis spaces.

Substituting these identities into Eq. [[eq:y\_Pu\_d]](#eq:y_Pu_d), we can re-write it as

We then multiply both sides on the left by , and given the property in Eq. [[eq:identity\_of\_basis\_spaces]](#eq:identity_of_basis_spaces), we isolate as

To simplify matters moving forward, we introduce a new matrix system dynamics which, just as captured the impact of inputs on output, will describe the impacts of s on s

We then apply the operator and recognizing as a constant, drop it out as done in Eq. [[eq:del\_y\_P\_del\_u]](#eq:del_y_P_del_u)

Referring back to our section on ILC ([[sec:ILC]](#sec:ILC)), this will all look very similar and one can properly assume our next step is to define our goal output. Recall our goal output ; given the identity of Eq. [[eq:a\_pinv\_Ty\_y]](#eq:a_pinv_Ty_y) we can write a new goal of

meaning that each trials , now marked with the subscript to indicate trials, has an associated error. Calling this error , we have

Once again applying the 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 and

Our new model, while still adhering to the format of Eq. [[eq:discrete\_state\_space\_model]](#eq:discrete_state_space_model) and the ILC format of Eq. [[eq:ILC\_law]](#eq:ILC_law), now has controllable dimensions. Just as going from state-space to ILC took our number of states from , in the transition to basis space we have taken our state count from , where . The same can be shown for our inputs. It can also be shown at every step of the derivation process that if and 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. [[eq:del\_u\_L\_e\_j]](#eq:del_u_L_e_j)

$$\delta\_{j+1}\underline{u}=\mathcal{L}e\_j
\tag{\ref{eq:del\_u\_L\_e\_j}}$$

Substituting and

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

Similar to how we defined , we can now define the controller

So that

Armed with this exact model and controller, we can now explore the the relationships between , , , , , and any resultant error.

#### Demonstration of Requirements

Recall the assumptions (Eqs. [[eq:y\_Ty\_a]](#eq:y_Ty_a) and [[eq:u\_Tu\_b]](#eq:u_Tu_b)) 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

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 to set the 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 [1.42](#fig:ilc_shaped_circle) or the word ‘Dartmouth’ in Figure [1.43](#fig:ilc_shaped_dartmouth).

##### FIPO

The first assumption we will relax is that the output basis space captures . That is our will fully be described in , but will only partially be in . In other words, we will have FIPO describing their respective spaces.

We can always construct a basis space to capture by setting one of the functions of equal to , 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 polynomials[[12]](#footnote-138), using the method shown in Eq. [[eq:cheby\_recurrence]](#eq:cheby_recurrence).

and we will construct as done in Eq. [[eq:u\_Tu\_b]](#eq:u_Tu_b), using a defined as

which produces the input signals shown in Figures [2.9](#fig:FIPO_input_1) and [2.11](#fig:FIPO_input_2). Recall that in our ILC problem, and are stacks of input/output data, rotating through the different components of each (see Eq. [[eq:stacked\_y\_star]](#eq:stacked_y_star)), 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.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Goal Inputs deconstructed from an explicitly constructed from basis functions to be in

We apply this to our system to create . For convenience, we will set = . It can be demonstrated that our output basis space does not capture our by computing our and attempting to go back to . For our system and given input,

$$\begin{aligned}
\alpha^\ast &= {\Phi\_y}^+ \underline{y}^\ast \tag{\ref{eq:a\_pinv\_Ty\_y}} \\
&= \begin{bmatrix}
0.1364 \\
-0.2043 \\
0.2136 \\
-0.2672 \\
-0.1991 \\
0.1067 \\
-0.0212 \\
-0.0241 \\
-0.0025 \\
-0.0320
\end{bmatrix}
\end{aligned}$$

If were in , then should be a zero vector[[13]](#footnote-142). However, we see both numerically in Eq. [[eq:y\*\_Ty\_a\*\_error]](#eq:y*_Ty_a*_error) and visually in Figures [2.12](#fig:FIPO_output_1) and [2.14](#fig:FIPO_output_2) that this is not the case.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  |  | | --- | --- | |  |  | |

Deconstructed s vs the goal constructed from . is found by inverting the the in the space . By then attempting to go back, this highlights the inability of to perfectly capture .

Now we have a 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 . If we set , then and . Just as before, we can compute – but now we must convert into the output basis space before we can proceed onto the next trial, by [[14]](#footnote-147).

Armed with our first error, we now iteratively apply our controller. Using the previous trials error in the space and controller , we compute our change in betas . We use that to compute our new , convert it from the space to get . Applying our new sequence of inputs, our system will produce new outputs to calculate an error with and calculate our . Repeat this process as needed.

Through these trials, an amazing property emerges. Even though we cannot capture perfectly in , we are still able to inform our controller with error data that it enables it to send the error to zero. Figures [2.15](#fig:FIPO_alpha_error_progression) and [2.17](#fig:FIPO_beta_error_progression) show that through trials, the error on both and are sent to zero.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of Errors on coefficients through perfect-knowledge controller trials when but . Even though , the associated coefficient errors can still go to zero.

We can additionally see the progression of the coefficients through trials in Figures [2.18](#fig:FIPO_alpha_progression) and [2.20](#fig:FIPO_beta_progression). We intentionally only plot a few of the coefficients to prevent a cluttered plot.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of Coefficients through perfect-knowledge controller trials when but

Figures [2.21](#fig:FIPO_mass_1_position) through [2.23](#fig:FIPO_shaped_output) show that the zero-error in the basis space translates to zero-error in the full-dimension space.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Progression of outputs through ILC trials when but

##### PIFO

Obviously the next thing to check is what if is fully defined in but is not in ? Or in our terminology, there is a set of PIFO describing those spaces. It is incredibly easy to ensure our is captured because, if you recall our ideal basis space condition, we can simply set one of our basis functions at the same time that we set .

So now our will only partially be described in , but will be fully in .

Ensuring we capture is understandably much harder without perfect system knowledge. With our current understanding, the only way to guarantee is in is to expand it to be and full rank. Obviously this high dimension hurts us, So let us test if it is completely necessary for .

We will repeat a similar process as the above example, except now we define first. For consistency, we will re-use the same parameters as those in Eq. [[eq:beta\_star\_in\_basis]](#eq:beta_star_in_basis), except now for

Remembering to split and plotting the goal outputs separately, we see our goals in Figures [2.24](#fig:PIFO_output_1) and [2.26](#fig:PIFO_output_2). 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.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Goal Outputs deconstructed from a explicitly constructed from basis functions to be in

We can back out our input

From there we calculate

$$\begin{aligned}
\beta^\ast &= \Phi\_u^+ \underline{u}^\ast \tag{\ref{eq:b\_pinv\_Tu\_u}} \\
&= \begin{bmatrix}
1,555.4 \\ 4,385.8 \\ 2,871.3 \\ 4,453.6 \\ 2,478.4 \\ 4,508.5 \\ 1,871.0 \\ 4,618.8 \\ 1,124.2 \\ 3,643.1
\end{bmatrix}
\end{aligned}$$

This can be numerically confirmed to not capture our input in Eq. [[eq:u\*\_Tu\_b\*\_error]](#eq:u*_Tu_b*_error) and visually in Figures [2.27](#fig:PIFO_input_1) and [2.29](#fig:PIFO_input_2)

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Deconstructed Goal s from , where was backed out of with perfect knowledge.

Having confirmed we have a 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 . Unsurprisingly, we follow the exact same steps as before[[15]](#footnote-164).

Here we begin to see the limits of our basis functions. While figure [2.30](#fig:PIFO_alpha_error_progression) shows that we can send to zero, no other parameter does so. Figure [2.32](#fig:PIFO_error_progression) shows that through trials, we reach a steady state error. Recall earlier from Eq. [[eq:a\_pinv\_Ty\_y]](#eq:a_pinv_Ty_y) that multiple s can produce the same . We are able to generate , but there is some additional that exists outside of . So we can capture fully in our basis, but cannot see the error that exists outside of it in the null space. Our input is not producing our , since our goes to zero, our betas stop learning.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of coefficient errors through trials when but . Contrast this with the earlier example where , we see now that the error of can still go to zero, but the error on reaches a non-zero steady state.

The progression of the coefficients through trials can be seen Figures [2.33](#fig:PIFO_alpha_progression) and [2.35](#fig:PIFO_beta_progression). Once again, only select coefficients are plotted for cleanliness.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of Coefficients through trials when but

Just how drastically off we are can be seen in Figures [2.36](#fig:PIFO_mass_1_position) through [2.38](#fig:PIFO_shaped_output)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Progression of Outputs through ILC trials when but

##### FIPO 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, 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 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 that produces (Eq. [[eq:y\*\_Pu\*\_d]](#eq:y*_Pu*_d)), and does this by changing inputs between trials, so while to goal may be within , 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 . We start with , apply it to produce , and compute . We follow our control law of Eq. [[eq:basis\_ilc\_control\_law]](#eq:basis_ilc_control_law) to compute , such that . Given that was a zero-vector, we can ignore it, and since we are saying that is deadbeat, we can say that is our best attempt at , meaning our best attempt at must be in .

If one were to repeat this example with a non-deadbeat controller, they would find that any found would always produce a in the space . So while it is possible to always achieve a that captures our goal of zero error, the space defining the inputs must be selected carefully to include . It is also important to note that the learned input is **not** the projection of onto [[16]](#footnote-175).

So our matters, but does not.[[17]](#footnote-176). Given that, why not define such that . Even if we then leave as the identity matrix, then thinking ahead to our RL problem, we would have the ability to infinitely reduce our ‘state’ dimensions down to . Then input-decoupling could be employed such that each controller could be updated in just trials, and that would only then need to be done times.

##### FISO

To test this theory, we will setup our problem very much like we did the FIPO example. will once again be defined as it is in Eq. [[eq:beta\_star\_in\_basis]](#eq:beta_star_in_basis), with our similarly following Eq. [[eq:Tu\_in\_cheby]](#eq:Tu_in_cheby). We will define . So we have a set of FISO describing those spaces.

We have already seen this input sequence in Figures [2.9](#fig:FIPO_input_1) and [2.11](#fig:FIPO_input_2), and show the outputs in Figures [2.39](#fig:FISO_output_1) and [2.41](#fig:FISO_output_2).

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Deconstructed Goal Outputs for and

However, the learning history is quite different. Even 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 point[[18]](#footnote-181) is 0.9823

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of coefficient errors through trials when and

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of coefficients through trials when and

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Progression of Outputs through ILC trials when and . Even though could in theory be fully described by , the controller is unable to capture

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

##### SIFO

We now check the other scenario, where the spaces can be described with a set of SIFO. This is an extreme version of our POFI example, so we set it up the same. The difference being we extract in Eq. [[eq:u\*\_from\_Py\*]](#eq:u*_from_Py*), we also use that to set our input basis space such that . The goals can be seen in Figures [2.24](#fig:PIFO_output_1) and [2.26](#fig:PIFO_output_2), and the inputs that get us there in Figures [2.51](#fig:SIFO_input_1) and [2.53](#fig:SIFO_input_2) – nothing new here.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Deconstructed Goal Inputs for and

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 - within our ‘zero’ range.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of coefficient errors through trials when and

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |

Progression of coefficients through trials when and

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

Progression of Positions through ILC trials when and . 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 , and lack thereof on , we must meet certain criteria when selecting our dimensions. It can be shown that condition is .

If we refer back to our controller in Eq. [[eq:basis\_ilc\_control\_law]](#eq:basis_ilc_control_law), this makes sense.[[19]](#footnote-205) 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 exists, our input basis space must include it.

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.

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

### Creating a Dynamic Basis Space from Learned Inputs

To effectively learn, we are limited by our ability to capture . For every reduction we can make to , we can similarly reduce . 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 , we will at most need basis functions. To start, we define a complete basis space to draw from that we know will capture the entire input. As we now know that the distinction between and does not matter, we will now use the same notation of for both[[20]](#footnote-211).

Next we define our s. As per the condition stated in Eq. [[eq:basis\_ilc\_num\_basis\_condition]](#eq:basis_ilc_num_basis_condition), we will set and use 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 basis space , and will learn an input [[21]](#footnote-212).

After each episode, we take the learned input and set that as a basis function. We then update the remaining basis functions with new s from , as shown in Eq [[eq:rolling\_basis]](#eq:rolling_basis).

where , marking what basis function we draw from , and ‘rolls-over’ as needed. So if , a -sequence would be .

One would hope that after sufficient episodes, there would be a found such that is included.

#### Example – Rolling Basis Space

First we set up the problem as done in the FIPO example, with shown in Figures [2.9](#fig:FIPO_input_1) and [2.11](#fig:FIPO_input_2) and in Figures [2.39](#fig:FISO_output_1) and [2.41](#fig:FISO_output_2).

Although we have a signal of length [[22]](#footnote-213), we do not need basis functions to ensure we capture our . Recall for our FIPO example, we defined in the space of Chebyshev Polynomials , so we can define as we define in Eq. [[eq:Tu\_in\_cheby]](#eq:Tu_in_cheby) to ensure we capture . If we did not know this to be the case, we would have to define to be a full rank matrix to ensure we spanned the whole input space.

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

Recall that our is defined with weights on , but also . If the learned input for a given basis space was the projection of onto that basis space, we would expect . As previously stated, this is not the case and so we end up with

The controller found a way to send (as shown in Figure. [2.63](#fig:rolling_basis_alpha_error)) 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 with the learned input as one of the basis functions

and repeat the process. Since we are using , make sure to update both basis spaces. If is left fixed, although it does not matter, learning will be sub-optimal. This is because our first pass finds the that sends , and so if we do not update , then that same learned input will be applied again with the same results - the system will see no added benefit from including new inputs.

We do this enough times to try every every basis function in multiple times and see that each time while we are able to send , we never find nor send .

|  |  |
| --- | --- |
|  |  |

Progression of through rolling basis space episodes

It can be seen that this approach does come close to producing our goal output in Figure [2.68](#fig:rolling_basis_shaped), 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 [2.65](#fig:rolling_basis_input1) and [2.66](#fig:rolling_basis_input2), and seeing they do not match . Additionally, we have a final error magnitude [[23]](#footnote-216).

|  |
| --- |
|  |

Progression of through rolling basis space episodes

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | | |  | | --- | |  | |

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 . Failure to capture can best be seen on the fringes of the input signals.

|  |
| --- |
|  |

Progression of through rolling basis space episodes. After sufficient trials have been made to cover all of , it can be seen that 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 .

Figure [2.64](#fig:rolling_basis_error) shows that after our first pass through all of the input basis we have a ‘petty good’ input. This is logical, as it has now taken into consideration every possible basis function that makes up our . However, Figure [2.69](#fig:rolling_basis_betas) 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 .

In Figure [2.70](#fig:rolling_basis_betas_forced) we ‘force’ the learned input that goes on to make up the next to be exactly . It can be seen that the system, if it finds the proper in its cycle will properly stay converged on that solution, so the prior example clearly did not find our goal input.

|  |
| --- |
|  |

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

#### Summary of Dyanmic Spaces

While it is possible to start with a finite number, , of basis functions to create a dynamic basis space which better captures , there is no guarantee to create one that captures .

There are two main reasons for the failings of this approach. Recall that there are numerous s that can produce the same same for a given . Additionally, while our basis functions may be independent of one another, as stated in Eq. [[eq:orthogonal\_basis]](#eq:orthogonal_basis), they are not necessarily independent in the response they illicit from the system in the basis space. So in addition to the different s that can get us , there are also multiple s to get us to . 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

### 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 , and that our input basis space must capture – no such condition exists for and . We have additionally shown that while it is possible to work with a dynamic basis space , the possibility that different inputs can create the same – or projection of on – makes exact control impossible.

## 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.

### 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 the produce and the cost they incur.

Conjugate Basis Functions have all the same properties shown in **What are Basis Functions** (Section  [[sub:what\_are\_basis]](#sub:what_are_basis)), and then some. As Frueh and Phan describe in their paper on LQL, the goal is to define basis functions such that when they are added to pre-existing basis space , 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 episodes worth of data, and we run one more episode to improve our and . The properties of LQL dictate that

See that if was the optimal input for , then will be optimal for and

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

### Deriving Conjugate Basis Functions

To derive our Conjugate Basis Functions, we must first define a cost function. Recall we want to send our error to zero via control of . In the converged ILC state, both and are 0. We will define our cost-per-episode the same way we defined our utility function for the ILC problem in Eq. [[eq:ilc\_utility]](#eq:ilc_utility), except expressing it as instead of

where is some cost matrix, and a one.

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

and substitute them into Eq. [[eq:conjugate\_cost]](#eq:conjugate_cost)

For reasons that will be seen shortly, as well as in the efforts of minimizing page space, we will define matrices and

By setting , we can solve for the which minimizes our cost function as

By the definition of the operator, we can write

We will make one final step to simplify future logic, and that is assume every attempt is a deadbeat attempt. Meaning always. This enables us to substitute as our open loop error , and do away with episodic notation:

where can be collected from an open-loop response (). The complete derivation can be seen in Appendix [6.1](#sec:lql_derivation).

Suppose we set . Our basis space would then have 2 functions and we would have 2 s:

where and are rows 1 and 2 of

We would like it that if we expanded to include , that would be generated without impacting and – as by the goal property of conjugate basis functions shown in Eq. [[eq:conjugate\_subspace]](#eq:conjugate_subspace). Yet is a function of so we run the risk of our previous s changing when changes - that is unless 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’:

which when written in terms of our basis space

or if explicitly written by

So long as the condition in Eq. [[eq:conjuct\_cond\_I]](#eq:conjuct_cond_I) is met, we can create s which solely depend on their associated , are optimal to minimize our cost function [[eq:conjugate\_cost]](#eq:conjugate_cost), and do not impact the optimality of other inputs as we previously saw in [[sub:dynamic\_arb\_basis]](#sub:dynamic_arb_basis).

### Extracting Conjugate Basis Functions

The issue we face now is we do not know , so we must derive a method to extract . 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 and their associated output where and is the number of conjugate basis functions to generate. We can apply our operator to this data set to produce and the same sequence for our outputs. We will define the batch notation

so that is all the changes in our episodes inputs, and the change in outputs. We can apply this delta-batch operation to Eq. [[eq:y\_Pu\_d]](#eq:y_Pu_d) to show:

As with all earlier examples, the operator removes the existence of a term.

We now want to find conjugate functions spanned by our inputs in our conjugate space . That is:

We create matrix **W** from this delta-batch data

substituting in Eq. [[eq:del\_y\_batch\_P\_del\_u\_batch]](#eq:del_y_batch_P_del_u_batch) and Eq. [[eq:inputs\_to\_conj\_map]](#eq:inputs_to_conj_map)

The final step can be made due to our conjunctionality condition of from Eq. [[eq:conjuct\_cond\_I]](#eq:conjuct_cond_I).

To extract then, we preform the Cholesky decomposition on . It can be shown that is then an upper-right triangular matrix. From , we can reverse Eq. [[eq:inputs\_to\_conj\_map]](#eq:inputs_to_conj_map) to extract our conjugate basis :

One will note that we use and not . We must ensure that exists by generating a that is full rank by our choice of inputs.

As well as generating the optimal basis function, LQL also finds the to associate with said function to minimize our costs. Recall Eq. [[eq:optimal\_beta\_lql]](#eq:optimal_beta_lql), which gives us our optimal in terms of basis space, system, costs, and error. We do not know , yet we can extract by starting with Eq. [[eq:del\_y\_batch\_P\_del\_u\_batch]](#eq:del_y_batch_P_del_u_batch) and Eq. [[eq:inputs\_to\_conj\_map]](#eq:inputs_to_conj_map) so that

which can be re-written as

Define .

Given this, we can write

which can be transposed and plugged into Eq. [[eq:optimal\_beta\_lql]](#eq:optimal_beta_lql) so that

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

##### Batch Example

We begin the setup of this example exactly as we did the example shown in [[par:fipo]](#par:fipo). So we set , create a of the first 10 chebyshev polynomials, and build , where

$$\beta^\ast = {\begin{bmatrix}1 & 0.2 & -0.3 & 4 & 0 & 0 & 0 & -1 & 0 & 0\end{bmatrix}}^T
\tag{\ref{eq:beta\_star\_in\_basis}}$$

This then defines our . Once again we can be assured that the inputs we will sweep do capture , and this is always a safe assumption. Recall we can always try more inputs, and there will be a definitive number 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 pairs of / data, so we will also include the open-loop response. So

The next step is to apply the operator. Eq. [[eq:delta\_operator]](#eq:delta_operator) defined the operator to be the difference between the 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 results without compromising the underlying logic. As such, we create a slightly modified operator, . This is just to add some notation to the deadbeat assumption we have been employing. We define this as:

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

and

Notice we have removed any dependency on and we can conveniently represent our as just the . Eq. [[eq:del\_y\_batch\_P\_del\_u\_batch]](#eq:del_y_batch_P_del_u_batch) can be seen to still apply. So we easily generate our batched output as

Next we must define our cost matrices and so we may compute **W**. Here we set [[24]](#footnote-233) and . Thus

which we can take the Cholesky decomposition of to get

In this example, our upper-triangular, matrix comes out to be

Our extracted as in Eq. [[eq:conj\_from\_rho]](#eq:conj_from_rho) is then and is partially shown below.

We verify this satisfies the condition of Eq. [[eq:conjuct\_cond\_I]](#eq:conjuct_cond_I) and indeed is such that

The associated can be calculated as

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 , we can perfectly produce the true signal without any additional information. If we apply , we see we capture our goal output perfectly (Figure [2.71](#fig:batch_learned))

|  |  |
| --- | --- |
|  |  |

Output Generated from Batch Generated Basis Space with weights overlaid on the Goal Output. As the was generated within a space spanned by all of the 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 given pre-existing basis functions.

To generate a new we need to introduce a new input to , ensuring full rank is maintained. This new input can be added to our batch so that

logically produces a which can similarly be added to the output batch.

By applying our operator[[25]](#footnote-238) and returning to Eq. [[eq:w\_for\_batch]](#eq:w_for_batch), we can write

For our th trial, we want to get , or the last column of . We can then write Eq. [[eq:full\_b1\_W]](#eq:full_b1_W) in terms of only its last columns

We can expand this as

Recall that that is upper-right triangular. Given this, we can extract the first element of as

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

and the final element can be more easily computed by first defining to capture all the previous components

so we can write

We can then derive our new conjugate basis function

and the associated from

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 step process

1. Choose your and
2. Define your input sequences such that they span the whole input space
3. Perform two initial experiments and to generate and [[26]](#footnote-239)
4. Extract from Eq. [[eq:w\_for\_batch]](#eq:w_for_batch) and [[eq:conj\_from\_rho]](#eq:conj_from_rho)
   * Compute for (Eq. [[eq:batch\_Hb]](#eq:batch_Hb) and Eq. [[eq:beta\_batch]](#eq:beta_batch)) and apply it to the system
5. Perform a third experiment with to generate . Use Eqs. [[eq:first\_rho\_iterative]](#eq:first_rho_iterative) and [[eq:last\_rho\_iterative]](#eq:last_rho_iterative) to extract , then plug into Eq. [[eq:new\_phi\_iterative]](#eq:new_phi_iterative) to compute
   * Compute for the given (Eq. [[eq:new\_h\_iterative]](#eq:new_h_iterative) and Eq. [[eq:new\_beta\_iterative]](#eq:new_beta_iterative)) and apply it to the system
6. Repeat with to extract as needed, using Eqs. [[eq:first\_rho\_iterative]](#eq:first_rho_iterative)– [[eq:new\_phi\_iterative]](#eq:new_phi_iterative)
   * Compute for the given (Eq. [[eq:new\_h\_iterative]](#eq:new_h_iterative) and Eq. [[eq:new\_beta\_iterative]](#eq:new_beta_iterative)) 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 and from the Batch Example [[par:batch\_ex]](#par:batch_ex).

To begin the iterative approach, we must first run two episode to such that we may compute our first [[27]](#footnote-240). As before, we let our 0th episode contain the open loop data, and apply as to create .

Since we are using the operator, as in the batch example, , and . This applies across all episodes, and allows for neater notation.

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

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

Applying then just to our system we see from Figure [2.72](#fig:iterative_1_basis), it clearly is not sufficient to produce the output we desire. So we proceed with our second basis function.

We excite the system with to generate . We must solve for in multiple steps now, but we find

once again matching that exactly of our batched approach.

Just for completeness, we will show one more example where we excite with . Following the steps outlined, we get

With explicit examples for all the three major steps, we now carry out the rest of the trials. [[28]](#footnote-241) After 10 trials (plus the one open-loop response), we find

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 . Figure [2.72](#fig:iterative_1_basis) shows that with just one basis function, we do not capture much. By Figure [2.73](#fig:iterative_3_basis) 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 , we can perfectly capture . Except now through the iterative approach, we can stop once we have enough, as in Figure [2.74](#fig:iterative_8_basis) where we stop after basis functions are generated.

|  |  |
| --- | --- |
|  |  |

Output Generated from Iteratively Generated Basis Space with weights when overlaid on the Goal Output

|  |  |
| --- | --- |
|  |  |

Output Generated from Iteratively Generated Basis Space with weights when overlaid on the Goal Output

|  |  |
| --- | --- |
|  |  |

Output Generated from Iteratively Generated Basis Space with weights when 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.

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

#### Properties

We see that both the batch and the iterative approach yield the same conjugate basis space and optimal weights . 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.

# Results

## 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 the 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.

### 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 . This translates exponentially to the number of trials needed to update a controller in the RL framework, and given that 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 and were the only requirements in order to capture . 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 trials. This ideal scenario may seem pointless at first, as we can only do so if . However, in the RL framework we introduce exploration noise, so it would be possible to have a with the same learning results.

To can determine this approximate however, we need to build out a starting . 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 (Eq. [[eq:conjugate\_subspace]](#eq:conjugate_subspace)), we hope to find that each additional does not impact the previous learned controllers. If is the controller found for the entire basis space , then is the controller learned for the from episode , such that

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

### The Conjugate LQR Controller

We know the conditions to ensure a found controller captures 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 in our , but we still find a controller in this space. It should be that if we find and separately, they would have the exact same parameters as controller [[29]](#footnote-256).

#### 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 of a circle. We set , and use Eq. [[eq:y1\_y2\_star]](#eq:y1_y2_star) 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

we generate 20 conjugate basis functions

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

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

We begin with . Recall that for these test to be logical, we must maintain the same on the outputs, so our output will still have 20 states – . For the ILC problem in a basis space, , so for our LQR controller’s and must be modified. Using Eq. [[eq:ilc\_basis\_H]](#eq:ilc_basis_H), we can write

It is important to remember to grab the right component of now. We have equal weightings for all our inputs, so this is easy to mess up and not realize. For , we grab [[30]](#footnote-257).

Solving for the under these parameters, we find

We repeat the same process for , defining with respect to and updating our for completeness. Computing the associated we find

These two results by themselves are not that important. It is when we learn a new controller with that we see the conjugate property emerge. For the found controller is

We will note that the controllers are ‘almost’ exactly identical. ’s first term differs from that of ’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 rounding[[31]](#footnote-258).

For completeness, we will run an ILC Trial as we did when exploring the requirements on basis functions. For our , system trying to draw a circle, we generate 20 conjugate basis functions. Once again employing our controller, we see that perfect control is possible just as it was before (Figures [3.1](#fig:conj_ilc_error) and [3.2](#fig:conj_ilc_shape)). So we are guaranteed to fully learn the necessary output / controller in at most basis attempts.

|  |  |
| --- | --- |
|  |  |

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

|  |  |
| --- | --- |
|  |  |

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

### 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 [[sub:dynamic\_arb\_basis]](#sub:dynamic_arb_basis).

For our dynamic basis space, there are a few different scenarios to consider. Recall our starting notation in section [[sec:basis\_functions]](#sec:basis_functions) where our input basis space had notation and our output had notation . We will re-introduce this now to consider our options.

#### Fixed Resolution:

In the first scenario, we define a fixed resolution and a , where is the most recently learned conjugate basis function. We use a fixed, full-resolution such that the controller we learn in this input space can be stacked with other controllers for spaces (where ) to form a larger controller, as we did with input decoupling. Here, we would run two initial trials to generate our first , the begin learning the controller in that space. While learning, we would produce s that could be used to generate a and have that ready to go when we are done learning with . This process could be repeated, and would at worst be done in a finite number of s. However, one would need to properly choose to ensure that the used to capture obeyed our condition in Eq. [[eq:basis\_ilc\_num\_basis\_condition]](#eq:basis_ilc_num_basis_condition) of . At worst, this means [[32]](#footnote-265).

##### Example – Full Resolution Identity on Output

We are back to our goal shown in Figure [3.2](#fig:conj_ilc_shape). To handle the worst case scenario, set . Our conjugate basis parameters (generated iteratively, but will still work out to match Eq. [[eq:circle\_conj\_basis]](#eq:circle_conj_basis)) are

Our RL parameters will be similar. Recall the earlier computational issues with and the need for proper exploration noise, shown in our derivation of in Eq. [[eq:rl\_ilc\_lqr\_small\_R\_controller]](#eq:rl_ilc_lqr_small_R_controller).

We generate our first iteratively, and learn the associated controller in that basis space. Denoting that as , we find

We repeat this process of generating conjugate basis functions (using Chebyshev Polynomials as our s for consistency with earlier examples, but we could use any input that was independent of previous s). We then take our and learn the new . 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 s in a stack, we find

We then take and compute our . Similar to in Eq. [[eq:ilc\_AB\_for\_lqr]](#eq:ilc_AB_for_lqr), we compute the Basis Space ILC Matrices as

When we solve for the with the process shown in Eq. [[eq:discounted\_LQR\_solution]](#eq:discounted_LQR_solution) and the parameters from Eq. [[eq:rl\_params\_ilc\_fixed\_res]](#eq:rl_params_ilc_fixed_res), we get

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](#fig:policy_ilc_rolling_error) appears so poor. However after a single trial of the complete controller, we have near zero error, as shown in Figure [3.4](#fig:policy_ilc_rolling_final_shaped).

|  |  |
| --- | --- |
|  |  |

Error Progression through Policy Learning ILC Trials with Rolling on inputs. For a fixed output basis , 20 s are tried to capture the input, and the associated is learned (but not applied).

|  |  |
| --- | --- |
|  |  |

Shaped Output Progression through Application of a Final Controller that was Learned One at a time. The resultant controller is then built out of the individually learned s for each

It can be easy to confuse this approach with input decoupling. While they are very similar, 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

Another option is to use a fixed resolution , where we have 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 () when our 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 in our examples, in which every component is . Compare this to when we used and only one component had a value of . It is easy to see that the basis coefficients that results from these different basis functions (see Eq. [[eq:a\_pinv\_Ty\_y]](#eq:a_pinv_Ty_y)) will differ drastically in magnitude. To rectify this discrepancy in magnitude, we are best off scaling our and such that . A scaling which increases the magnitude of results in smaller s,meaning our 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. [[eq:conjuct\_cond\_I]](#eq:conjuct_cond_I). We can safely normalize individual basis functions and still have our conjunct condition of diagonality, but for simplicity we scale the entire by some . Then our conjunctionality matrix from Eq. [[eq:conjuct\_setup\_matrix]](#eq:conjuct_setup_matrix) just equals .

So where to deal with small s we had to introduce more exploration noise, for large s we must scale up relative to .

##### Example – Full Resolution Conjugate Basis on Output

In this example, we start with the entirety of our basis functions pre-generated (computed through trials). Keeping the same parameters and goal from above, our can be seen in Eq. [[eq:circle\_conj\_basis]](#eq:circle_conj_basis).

We then set our LQR parameters as

As we are working with a relatively large , we can use this smaller exploration term. If we were to reduce as we have in the past, the same steps must be made to ensure rich data by increasing the range from which draws from.

If we were to run this example as is, setting , where is our conjugate basis functions, we see that our LQR Controller constructed from perfect knowledge

does not match that of the one produced from Policy Iteration

Even increasing our s range by a factor of a million does not resolve this discrepancy. If we set 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 and values are arbitrary and only mean something in relation to each other, the magnitudes of and only matter in relativity. In this case

Holding all the other parameters the same, we once again find that our controller can be found by learning one controller at a time and then stacking them together, such that . In this example

We can see this controller works to reduce error in Figure [3.5](#fig:fixed_conjugate_scaled), but predictably takes a extreme amount of trials due to the higher .

|  |  |
| --- | --- |
|  |  |

Error Progression through Policy Learning ILC Trials with Scaled Conjugate Output Basis

To confirm that this descent continues, we check the poles of () and see that that are within the unit circle - though just barely.

|  |
| --- |
|  |

Pole locations of the system under a controller found with a fixed and by rotating through . Of the poles, are exactly while are and . This is due to the required values.

#### Fixed Resolution

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 , though as previously stated it does not matter – we will see that some additional steps must be taken.

##### Example –

The first step is to establish our basis functions. will be our conjugate basis functions , and . 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 by 100 (bringing the norm to 316) was sufficient to prevent issues.

We then set our LQR parameters as[[33]](#footnote-276)

We then iteratively learn s. For each , our stays fixed at since we only have one ‘output’ , but our rolls along the diagonals of . 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 and find after all possible controllers

for the output basis space and input basis space .

We check this against our LQR solution of perfect knowledge to get

which is practically identical[[34]](#footnote-277).

Applying this controller, we see in Figure [3.6](#fig:policy_ilc_rolling_error_y_ast) that it does bring down our error!

|  |  |
| --- | --- |
|  |  |

Error Progression through Policy Learning ILC Trials with Rolling on inputs and . Given the relatively high , 0 error is not achieved quickly but it can be seen to be being reduced.

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 and , we end up with the results in Figure [3.7](#fig:policy_ilc_y_star_crushed). 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.

|  |  |
| --- | --- |
|  |  |

Complete Error Progression through Policy Learning ILC Trials with Rolling on inputs and . 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.

##### Growing

The final approach to try is relaxing our assumption of holding fixed. Suppose we learn one conjugate function , and use that for our basis functions. So and (scaled for reasons previously shown). The associated LQR controller would be , and thus very unlikely to capture . So we generated and add another conjugate basis function . Here, we learned the LQR controller where and . Now our LQR controller is , 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 in both input and output space, as we are applying RL to learn the associated controller. So after only trials, we could update our first controller. Then , and so on, where each added controller needs trials to update itself once through RL techniques.

##### Example – Growing

We establish the parameters our parameters from which we will define our conjugate basis functions , as we will be learning iteratively. Sticking with the earlier examples

Next, we establish the learning parameters with which we will be working. There is a frustrating balancing act between our , , and now the amount by which we scale our vs - . To demonstrate the functionality of this approach without extreme numbers, we choose

The process now is very similar to our earlier examples with a fixed .

To start, we generate our first . However, with each generated , we now must update our and . On this initial pass:

We now take these basis spaces and learn the ILC controller in their space. Taking care to grab and as our costs, we learn that our first controller is

Keep in mind, we want this to be a scalar. Both and are in this initial pass.

We now move to our second trial. We generate iteratively and updated our basis spaces as

Notice that now , and as such our found controller is the

To combine this with our previous controller, which found the optimal input in the space for error in that same space, we stack the two and pad it with zeros such that

We repeat the process until we have explored all necessary s to ensure we fully capture . 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

Note that this is not our in the space of and . However, the bottom row of both should perfectly match as they are both defined in that space.

Due to our relatively large , we wouldn’t expect the application of this controller to converge quickly. It would seem that from Figure [3.8](#fig:policy_ilc_rolling_phi_error_y) this approach appears to work.

|  |  |
| --- | --- |
|  |  |

Error Progression through Policy Learning ILC Trials with Rolling on inputs and outputs

Fearing similar results from our earlier example of a fixed , we check the poles of our system. Recall that anything indicates the system is unstable under the listed controller. For this example, we have 5 such poles that indicate instability. They exceed by magnitudes of 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 . We have seen that smaller s help reduce the magnitude of these poles. More on this in Section [[sec:future\_work]](#sec:future_work) on Future Work.

# 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 step process will take our number of states () and inputs () to learn and scale them to and , 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 was capture, but we had to be able to properly generate our . 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

Under these conditions, we were practically no better off than we started. The only way to guarantee 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 from two directions at once. The conjugate approach offers a way to learn the in a discrete number of steps, while the basis functions generated in the process can be used to learn a controller. This two-pronged approach means that even if we end up needing basis functions to capture our , we can find that in a definite number of trials. Then both while finding this and after it has been found, we can learn the controller to keep it there should new disturbances or errors be introduced.

# 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 matrix would have to be redefined but the logic is consistent when are some function of . 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 , , , , and values. For many examples, it was necessary to drastically increase to show that the RL framework continued to function, but obviously increased the learning time necessary for a given problem. Exploring ways to reduce 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 , , and 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 [[sec:rl\_on\_conjugate\_basis\_ilc]](#sec:rl_on_conjugate_basis_ilc). 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 trials when a new basis function comes along. If this formulation worked (with a sufficiently small ), then it would be possible to find a controller for a complete system in trials - trials for the RL approach and to generate the associated conjugate basis function. This is essentially what we shown when we learned with , except our 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 and 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 is when the does not capture the . It has been shown to not be the projection of . This is likely due to the fact that the is transformed by the matrix, and then looped through conversions before going back through a controller to get s (or s) all before going back to being a . I would suspect that the projection of is then onto some space that is a transform of , with matrices and involved.

Finally, we have investigated initial steps to reduce dimensions then grow, but no techniques to then work backwards. As shown in Eq. [[eq:conjugate\_beta\_star]](#eq:conjugate_beta_star), it is possible to learn the weighting of some inputs even when they were not in the original . 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

that previously satisfied our conjunctionality condition such that

If we were to combine and into one term, we could reduce the dimension of our basis space to created . Now

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.

# Derivations

## LQL Derivation

Recall we are starting with the cost function :

where is some cost matrix, and a one.

We need a that minimizes the cost function, so we must express all relevant items in term of .

Plugging those into Eq. [[eq:ap\_conjugate\_cost]](#eq:ap_conjugate_cost)

We first re-arrange items under their transpose operators. We utilize the fact that to expand our cost function to

It can be helpful here to remind ourselves of the dimensions of each component. Sticking to our convention of number of states , number of inputs , number of steps in the ILC process , and number of basis functions [[35]](#footnote-294), our components follow as

| Variable | Dimensions |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

As expected, each component in Eq. [[eq:ap\_expanded\_cost]](#eq:ap_expanded_cost) is a scalar. It is then important to remember that the transpose of a scalar is merely itself, and our and matrices are symmetric such that and . This leads to use being able to recognize certain components in Eq. [[eq:ap\_expanded\_cost]](#eq:ap_expanded_cost) are equal to each other as

So that we may now write our cost function as

Before we can take our derivative with respect to , we must make one final substitution. Recognize that

and expand our cost function to

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 with respect to , and to do so it is helpful to remember the following identifies of matrix calculus

Then,

Recognizing the common pattern of and , we introduce the shorthand

where can see that and

We re-write the above Eq. [[eq:cost\_deriv\_del\_beta]](#eq:cost_deriv_del_beta) as

Setting the derivative and re-arranging, we find

Now we define the matrix we see in Eq. [[eq:conjuct\_setup\_matrix]](#eq:conjuct_setup_matrix)

and divide both sides of our equation by so

Remember the operator defined as in Eq. [[eq:delta\_operator]](#eq:delta_operator). Reversing that, we see

Subtracting from both sides,

can be shown to be , so multiplying it by its inverse produces the identity matrix. We left multiply by to get

# Matlab Simulations and Examples

## Github

All code and images can be found at the repository <https://github.com/NoahDunleavy/thesis_rl_on_ilc.git>

## Background

%A General Intro to State Space and Modern Control Theory  
%Noah Dunleavy  
%Honors Thesis under direction of Professor Minh Q. Phan  
%Thayer School of Engineering  
clc; clear;  
addpath('Saved Data', 'Functions');  
setDefaultFigProp();  
plot\_all = false; %most run time goes into generating figs  
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  
if update\_file\_path ~= -1  
 keyboard %ensure we have to be very concious about ever updating all the images  
end  
  
%% Description  
%{  
This intro code provides the basics of 'known' status in the field. Tot hose familiar with the Thayer School of Engineering's curriculum, it is a recap 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 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.  
  
One will note that the deadbeat controller, while time optimal, requires significant control effort that may not be realsitic 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.   
  
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).   
  
All of the previous examples rely on perfect knowledge not typically possible in the real world (although we can approximate and identiy 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.   
  
Finally, the idea of basis functions is introduced briefly to lay the  
groundwork for subsequent explorations.  
%}  
  
%% System Creation  
  
%Scalar Representation  
m1 = 1; %Mass of the block, [kg]  
m2 = 0.5;  
num\_masses = 2;  
k1 = 100; %Spring constant, [N/m]  
k2 = 200;  
c1 = 1; %Dampning Coeffeceint, [Ns/m]  
c2 = 0.5;  
  
%Matrix Formulations  
M = diag([m1, m2]); %mass matrix  
K = [k1 + k2, -k2; -k2, k2]; %stiffness matrix  
C\_damp = [c1 + c2, -c2; -c2, c2] ; %Dampning matrix  
  
%Continuous Statespace Formulations  
Ac = [zeros(num\_masses), eye(num\_masses); -M^-1 \* K, -M^-1 \* C\_damp];  
Bc = [zeros(num\_masses); M^-1];  
C = [1, 0, 0, 0; 0, 1, 0, 0]; %Monitor the block positions as outputs  
D = [0, 0; 0, 0]; %  
  
%Dimensional Variables  
num\_states = height(Ac); %Ac is n x n  
num\_inputs = width(Bc); %Bc is n x r  
num\_outputs = height(C); %C is m x n  
  
%Ensure against Nyquist  
eigens = eig(Ac); %Ac captures the modes of a given system  
natural\_freqs = imag(eigens)/(2\*pi);%Get all the natural frequencies in Hz (convert from rad/sec)  
max\_freq = max(natural\_freqs); %The highest frequency is the one we must cater to  
nyquist\_freq = 2 \* max\_freq; %Minimum sampling frequency to avoid nyquist sampling  
nyquist\_rate = 1 / nyquist\_freq; %Sample frequency to period  
  
dt = 0.01; %Set the sampling rate we will be using  
if (dt > nyquist\_rate) %Ensure that the true sampling period is below the nyquist specifications  
 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  
 sprintf('Selected sample rate was insufficient, setting dt = %.2e seconds', dt)  
end  
  
%Check Controllability  
if ~is\_controllable(Ac, Bc) %always check it is even possible to control  
 printf('The Continuous System is not Controllable!')  
end  
  
%Convert to Discrete  
[A, B] = c2d(Ac, Bc, dt); %Performs the necessary discretization operations - verify this for yourself  
  
%Set initial conditions  
initial\_displacement = [1; 0]; %displace the cooresponding blocks initially by this amount, [meters]  
initial\_velocity = [0; 2]; %Start cooresponding blocks with this velocity [m/s]  
x0 = [initial\_displacement; initial\_velocity]; %The top half sets position ICs, and the bottom half sets velocity  
  
%Simulation Duartion  
max\_samples = 500; %set the maximum number of steps out to simulate (note: max k will be -1, since we start at k=0)  
max\_time = max\_samples \* dt; %Important distinction: sample STEPS is not the same as time  
  
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  
%% Open-Loop Simulation  
cont\_vs\_disc\_resolution = 1; %Additional resolution to give continuous sinulations for smoother plots, and so the evident differences can be seen between continious and discrete  
%Setting to 1 speeds up rendering process, and can just use matlabs  
%'plot' vs stairs to highlight  
  
open\_time\_continuous = linspace(0, max\_time, max\_samples \* cont\_vs\_disc\_resolution)';%Generate the time scale at the higher resolution  
continuous\_system = ss(Ac, Bc, C, D); %Turn the continuous matricies into a system for simulation  
continuous\_open\_outputs = lsim(continuous\_system, zeros(height(open\_time\_continuous), num\_inputs), open\_time\_continuous, x0);  
[discrete\_open\_outputs, discrete\_open\_states] = dlsim(A, B, C, D, zeros(max\_samples, num\_inputs), x0); %Generate x(0) -> x(max\_samples - 1), and cooresponding outputs (y)  
  
%% Render Open Loop Responses  
if plot\_all  
 %Block 1 Position  
 fig = figure('Name', 'Open-loop Mass 1 Position');  
 plot(open\_time\_continuous, continuous\_open\_outputs(:, 1));  
  
 title('Mass 1 Position');  
 subtitle('Open-Loop', 'FontSize', getappdata(groot, 'DefaultSubtitleFontSize'));  
 xlabel('Time (sec)') %only point where we will be using time as the x axis  
 ylabel('Position (m)')  
 save\_figure(update\_file\_path, fig, 'Continuous Open-Loop - Mass 1'); %save just the continuous Model  
 %Add in discrete element  
 hold on;  
 stairs(dt \* (0:(max\_samples-1)), discrete\_open\_outputs(:, 1)); %note the dt scaling such that axises are the same  
 hold off;  
 legend('Continuous', 'Discrete')  
 save\_figure(update\_file\_path, fig);  
   
 %Mass 2 position  
 fig = figure('Name', 'Open-loop Mass 2 Position');  
 plot(open\_time\_continuous, continuous\_open\_outputs(:, 2));  
 title('Mass 2 Position');  
 subtitle('Open-Loop', 'FontSize', getappdata(groot, 'DefaultSubtitleFontSize'));  
 xlabel('Time (sec)')  
 ylabel('Position (m)')  
 save\_figure(update\_file\_path, fig, 'Continuous Open-Loop - Mass 2'); %save just the continuous Model  
 %Add in discrete element  
 hold on;  
 stairs(dt \* (0:(max\_samples-1)), discrete\_open\_outputs(:, 2));  
 hold off;  
 legend('Continuous', 'Discrete')  
 save\_figure(update\_file\_path, fig);  
  
 %Zoomed in to show the models match  
 num\_zoomed\_in = 11;  
 fig = figure('Name', 'Zoomed-in Open-loop Mass 1 Position');  
 plot(open\_time\_continuous(1:num\_zoomed\_in), continuous\_open\_outputs(1:num\_zoomed\_in, 1));  
 hold on;  
 stairs(dt \* (0:(num\_zoomed\_in-1)), discrete\_open\_outputs(1:num\_zoomed\_in, 1)); %note the dt scaling such that axises are the same  
 scatter(dt \* (0:(num\_zoomed\_in-1)), discrete\_open\_outputs(1:num\_zoomed\_in, 1), 'filled', '\*', 'MarkerFaceColor', 'red', 'MarkerEdgeColor', 'red', 'SizeData', 90);  
 hold off;  
 legend('Continuous', 'Discrete')  
 title('Mass 1 Position');  
 subtitle('Open-Loop', 'FontSize', getappdata(groot, 'DefaultSubtitleFontSize'));  
 xlabel('Time (sec)') %only point where we will be using time as the x axis  
 ylabel('Position (m)')  
 xlim([0, dt\*(num\_zoomed\_in-1)])  
 save\_figure(update\_file\_path, fig);  
  
 fig = figure('Name', 'Zoomed-in Open-loop Mass 2 Position');  
 plot(open\_time\_continuous(1:num\_zoomed\_in), continuous\_open\_outputs(1:num\_zoomed\_in, 2));  
 hold on;  
 stairs(dt \* (0:(num\_zoomed\_in-1)), discrete\_open\_outputs(1:num\_zoomed\_in, 2)); %note the dt scaling such that axises are the same  
 scatter(dt \* (0:(num\_zoomed\_in-1)), discrete\_open\_outputs(1:num\_zoomed\_in, 2), 'filled', '\*', 'MarkerFaceColor', 'red', 'MarkerEdgeColor', 'red', 'SizeData', 90);  
 hold off;  
 legend('Continuous', 'Discrete')  
 title('Mass 2 Position');  
 subtitle('Open-Loop', 'FontSize', getappdata(groot, 'DefaultSubtitleFontSize'));  
 xlabel('Time (sec)') %only point where we will be using time as the x axis  
 ylabel('Position (m)')  
 xlim([0, dt\*(num\_zoomed\_in-1)])  
 save\_figure(update\_file\_path, fig);  
end  
  
%% Simple Pole Placement Controller  
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  
placed\_controller = -place(A, B, pole\_locations);  
verify\_poles = eig(A + B \* placed\_controller);  
if any(abs(verify\_poles) >= 1)  
 printf('A Pole was placed outside of the unit circle, unstable controller!')  
end  
  
%Visualize the poles  
if plot\_all  
 plot\_pole\_placement('Simple Pole Placement', 'Simple Pole Placement', verify\_poles, pole\_locations, update\_file\_path);  
end  
closed\_pole\_samples = 60;  
pole\_states = zeros(num\_states, closed\_pole\_samples);  
pole\_inputs = zeros(num\_inputs, closed\_pole\_samples);  
pole\_outputs = zeros(num\_outputs, closed\_pole\_samples);  
pole\_states(:, 1) = x0; %set the initial conditions  
  
for k = 1:closed\_pole\_samples  
 pole\_inputs(:, k) = placed\_controller \* pole\_states(:, k); %u(k) = F \* x(k) - linear control law  
 pole\_states(:, k + 1) = A \* pole\_states(:, k) + B \* pole\_inputs(:, k); %x(k+1) = A\*x(k) + B\*u(k)  
 pole\_outputs(:, k) = C \* pole\_states(:, k) + D \* pole\_inputs(:, k); %y(k) = C\*x(k) + D\*u(k)  
end  
  
if plot\_all  
 plot\_two\_mass('Pole Placement', 'Closed-Loop System with Pole Placement', pole\_outputs, pole\_inputs, update\_file\_path); %function to plot positions and forces  
end  
  
%% Deadbeat Controller  
deadbeat\_poles = [-0.00001, 0.00001, 0.00000i, -0.00000i]; %plce will not allow for multiple poles at same location, so make all really close to 0  
deadbeat\_controller = -place(A, B, deadbeat\_poles);  
verify\_poles = eig(A + B \* deadbeat\_controller);  
if any(abs(verify\_poles) > 1)  
 printf('A Pole was placed outside of the unit circle, unstable controller!')  
end  
  
%Visualize the poles  
if plot\_all  
 plot\_pole\_placement('Deadbeat Pole Placement', 'Deadbeat Pole Placement', verify\_poles, deadbeat\_poles, update\_file\_path);  
end  
deadbeat\_samples = 20;  
deadbeat\_states = zeros(num\_states, deadbeat\_samples);  
deadbeat\_inputs = zeros(num\_inputs, deadbeat\_samples);  
deadbeat\_outputs = zeros(num\_outputs, deadbeat\_samples);  
deadbeat\_states(:, 1) = x0; %set the initial conditions  
  
for k = 1:deadbeat\_samples  
 deadbeat\_inputs(:, k) = deadbeat\_controller \* deadbeat\_states(:, k); %u(k) = F \* x(k) - linear control law  
 deadbeat\_states(:, k + 1) = A \* deadbeat\_states(:, k) + B \* deadbeat\_inputs(:, k); %x(k+1) = A\*x(k) + B\*u(k)  
 deadbeat\_outputs(:, k) = C \* deadbeat\_states(:, k) + D \* deadbeat\_inputs(:, k); %y(k) = C\*x(k) + D\*u(k)  
end  
  
if plot\_all  
 plot\_two\_mass('Deadbeat Controller', 'Deadbeat Controller', deadbeat\_outputs, deadbeat\_inputs, update\_file\_path);  
end  
  
%% LQR Controller  
Q = 100 \* eye(num\_states); %cost of each states being away from 0  
R = 1 \* eye(num\_inputs); %cost od each input away from 0  
gamma = 0.8; %learning factor  
  
%Consturct an LQR controller  
%The LQR controller minimizes the cost function J = u' \* R \* u + x' \* x  
%Where u = F\*x  
F\_lqr = discounted\_LQR(A, B, gamma, Q, R);  
  
lqr\_samples = 200;  
lqr\_states = zeros(num\_states, lqr\_samples);  
lqr\_inputs = zeros(num\_inputs, lqr\_samples);  
lqr\_outputs = zeros(num\_outputs, lqr\_samples);  
lqr\_states(:, 1) = x0; %set the initial conditions  
  
for k = 1:lqr\_samples  
 lqr\_inputs(:, k) = F\_lqr \* lqr\_states(:, k); %u(k) = F \* x(k) - linear control law  
 lqr\_states(:, k + 1) = A \* lqr\_states(:, k) + B \* lqr\_inputs(:, k); %x(k+1) = A\*x(k) + B\*u(k)  
 lqr\_outputs(:, k) = C \* lqr\_states(:, k) + D \* lqr\_inputs(:, k); %y(k) = C\*x(k) + D\*u(k)  
end  
%Find where it placed the poles  
lqr\_poles = eig(A + B \* F\_lqr);  
if plot\_all  
 plot\_pole\_placement('Big Q LQR Pole Locations', sprintf('Q/R = %.d Pole Placement', Q(1, 1)/R(1, 1)), lqr\_poles, -1, update\_file\_path);  
end  
%Plot full system out  
if plot\_all  
 plot\_two\_mass('Big Q LQR Controller', sprintf('With LQR Controller (Q/R=%.d)', Q(1, 1)/R(1, 1)), lqr\_outputs, lqr\_inputs, update\_file\_path);  
end  
  
%% Demo Higher R LQR  
R\_big = 10 \* eye(num\_inputs); %make relaive weightings more skewed to input  
gamma = 0.8;  
  
  
big\_R\_F\_lqr = discounted\_LQR(A, B, gamma, Q, R\_big);  
  
big\_R\_lqr\_samples = 1200;  
big\_R\_lqr\_states = zeros(num\_states, big\_R\_lqr\_samples);  
big\_R\_lqr\_inputs = zeros(num\_inputs, big\_R\_lqr\_samples);  
big\_R\_lqr\_outputs = zeros(num\_outputs, big\_R\_lqr\_samples);  
big\_R\_lqr\_states(:, 1) = x0; %set the initial conditions  
  
for k = 1:big\_R\_lqr\_samples  
 big\_R\_lqr\_inputs(:, k) = big\_R\_F\_lqr \* big\_R\_lqr\_states(:, k); %u(k) = F \* x(k) - linear control law  
 big\_R\_lqr\_states(:, k + 1) = A \* big\_R\_lqr\_states(:, k) + B \* big\_R\_lqr\_inputs(:, k); %x(k+1) = A\*x(k) + B\*u(k)  
 big\_R\_lqr\_outputs(:, k) = C \* big\_R\_lqr\_states(:, k) + D \* big\_R\_lqr\_inputs(:, k); %y(k) = C\*x(k) + D\*u(k)  
end  
%Find where it placed the poles  
big\_R\_lqr\_poles = eig(A + B \* big\_R\_F\_lqr);  
if plot\_all  
 plot\_pole\_placement('Big R Pole Placement', sprintf('Q/R = %.d Pole Placement', Q(1, 1)/R\_big(1, 1)), big\_R\_lqr\_poles, -1, update\_file\_path);  
end  
%Plot full system out  
if plot\_all  
 plot\_two\_mass('Big R LQR Controller', sprintf('With LQR Controller (Q/R = %.d)', Q(1, 1)/R\_big(1, 1)), big\_R\_lqr\_outputs, big\_R\_lqr\_inputs, update\_file\_path);  
end  
  
%% ILC Introduction  
%Setup ILC Parameters  
p = 100; %number of steps to a process  
num\_ilc\_states = num\_outputs \* p; %effective states for ILC are the errors, generated from outputs  
num\_ilc\_inputs = num\_inputs \* p; %r inputs for every time step  
  
x0\_ilc = x0;  
  
%Descriptive Matrix  
[P\_full, d\_full] = P\_from\_ABCD(A, B, C, D, p, x0\_ilc); %generate a descriptive matrix which captures the relation of inputs to outputs  
  
%ILC State-space:  
% e(j+1) = e(j) - P\*del\_u(j)  
% y(j) = e(j)  
A\_ilc = eye(num\_ilc\_states);  
B\_ilc = -P\_full;  
C\_ilc = eye(num\_ilc\_states);  
D\_ilc = 0;  
  
if ~is\_controllable(A\_ilc, B\_ilc) %always check it is even possible to control  
 printf('The ILC System is not Controllable!')  
end  
  
F\_ilc = 0.8 \* pinv(P\_full); %Note: any controller that palces poles inside the unit circle works.  
  
%Set the Goal Outputs (only have goals for p steps, regardless of lead)  
%Use function draw\_to\_XY(p) if you would like to set your own  
%drawing/special output. This is not written to be super robust, choose  
%matchig p values  
%[drawn\_x, drawn\_y] = draw\_to\_XY(p);  
%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  
  
y\_star\_x = cos(2 \* pi \* (0:(p-1)) / p)';  
y\_star\_y = sin(2 \* pi \* (0:(p-1)) / p)';  
goal\_matrix = [y\_star\_x, y\_star\_y]; %stack inputs next to eachother  
y\_star = reshape(goal\_matrix', [], 1); %combine the seperate goals of each output into one vertical vector, alternating as necessary  
  
%ILC Structure  
num\_trials = 10; %number of trials to learn input  
ILC\_Trial(num\_trials).output = [];  
ILC\_Trial(num\_trials).input = [];  
ILC\_Trial(num\_trials).del\_u = []; %relevant control parameter  
ILC\_Trial(num\_trials).output\_error = []; %relevant control parameter  
  
%First Trial  
trial\_num = 1;  
ILC\_Trial(trial\_num).del\_u = zeros(num\_ilc\_inputs, 1); %first trial we set whatever del\_u we want  
ILC\_Trial(trial\_num).input = zeros(num\_ilc\_inputs, 1); %Similarly, first input is arbitary  
  
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  
relevant\_output = ILC\_Trial(trial\_num).output((num\_outputs + 1):end); %only compare to the ones we can / want to control  
ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
  
%Subsequent Iterations  
for trial\_num = 2:num\_trials  
 %Inputs  
 ILC\_Trial(trial\_num).del\_u = F\_ilc \* ILC\_Trial(trial\_num - 1).output\_error;  
 ILC\_Trial(trial\_num).input = ILC\_Trial(trial\_num).del\_u + ILC\_Trial(trial\_num - 1).input; %d\_u(j) = u(j) - u(j-1)  
 %Output  
 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];  
 relevant\_output = ILC\_Trial(trial\_num).output((1 + num\_outputs):end);  
 %Error  
 ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
end  
  
if plot\_all  
 to\_plot\_ilc = [1, 2, 5, num\_trials];  
 plot\_dual\_ilc('Placed Controller ILC', 'ILC with a Placed Controller', ILC\_Trial, y\_star, -1, to\_plot\_ilc, update\_file\_path); %update\_file\_path  
end  
  
%% Arbitrary ILC  
%Setup ILC Parameters  
if false  
 clear 'ILC\_Trial' %make sure we do not carry over any ILC from previous  
 p = 500; %number of steps to a process  
 num\_ilc\_states = num\_outputs \* p; %effective states for ILC are the errors, generated from outputs  
 num\_ilc\_inputs = num\_inputs \* p; %r inputs for every time step  
   
 x0\_ilc = x0;  
   
 %Descriptive Matrix  
 [P\_full, d\_full] = P\_from\_ABCD(A, B, C, D, p, x0\_ilc); %generate a descriptive matrix which captures the relation of inputs to outputs  
   
 A\_ilc = eye(num\_ilc\_states);  
 B\_ilc = -P\_full;  
 C\_ilc = eye(num\_ilc\_states);  
 D\_ilc = 0;  
   
 if ~is\_controllable(A\_ilc, B\_ilc) %always check it is even possible to control  
 printf('The ILC System is not Controllable!')  
 end  
   
 F\_ilc = 0.8 \* pinv(P\_full); %Note: any controller that palces poles inside the unit circle works.  
   
 loadedShape = load('Saved Data\dartmouth\_p500.mat'); %read in the file  
 y\_star\_x = loadedShape.drawn\_x';  
 y\_star\_y = loadedShape.drawn\_y';  
 scale = 10;  
 goal\_matrix = [y\_star\_x, y\_star\_y]; %stack inputs next to eachother  
 y\_star = scale \* reshape(goal\_matrix', [], 1); %combine the seperate goals of each output into one vertical vector, alternating as necessary  
   
 %ILC Structure  
 num\_trials = 10; %number of trials to learn input  
 ILC\_Trial(num\_trials).output = [];  
 ILC\_Trial(num\_trials).input = [];  
 ILC\_Trial(num\_trials).del\_u = []; %relevant control parameter  
 ILC\_Trial(num\_trials).output\_error = []; %relevant control parameter  
   
 %First Trial  
 trial\_num = 1;  
 ILC\_Trial(trial\_num).del\_u = zeros(num\_ilc\_inputs, 1); %first trial we set whatever del\_u we want  
 ILC\_Trial(trial\_num).input = zeros(num\_ilc\_inputs, 1); %Similarly, first input is arbitary  
   
 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  
 relevant\_output = ILC\_Trial(trial\_num).output((num\_outputs + 1):end); %only compare to the ones we can / want to control  
 ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
   
 %Subsequent Iterations  
 for trial\_num = 2:num\_trials  
 %Inputs  
 ILC\_Trial(trial\_num).del\_u = F\_ilc \* ILC\_Trial(trial\_num - 1).output\_error;  
 ILC\_Trial(trial\_num).input = ILC\_Trial(trial\_num).del\_u + ILC\_Trial(trial\_num - 1).input; %d\_u(j) = u(j) - u(j-1)  
 %Output  
 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];  
 relevant\_output = ILC\_Trial(trial\_num).output((1 + num\_outputs):end);  
 %Error  
 ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
 end  
   
 if plot\_all  
 to\_plot\_ilc = [1, 2, 5, num\_trials];  
   
 resave\_dartmouth = update\_file\_path  
 %Super expliclity re-save the dartmouth plot. When doing so, go into  
 %save\_figure and adjust legend location for ONLY this one. Just  
 %because where it draws things  
   
 plot\_dual\_ilc('Dartmouth ILC', 'Arbitrary ILC with a Placed Controller', ILC\_Trial, y\_star, -1, to\_plot\_ilc, resave\_dartmouth); %update\_file\_path  
 end  
else  
 sprintf('Not doing the big ILC to save computation time')  
end  
  
%% Reinforcement Learning - Policy Iteration  
converged\_k = 20; %number of steps to run out for with 'converged' controller  
num\_controllers = 5; %number of iterations of each controller before updating  
Pj\_dim = (num\_states + num\_inputs); %minimum dimensions of the P matrix such that we can solve (noise free)  
num\_collections = Pj\_dim^2; %number of data collections to make before solving the inv (noise free)  
  
total\_tries = num\_controllers \* num\_collections + converged\_k;  
  
F\_policy = zeros(num\_inputs, num\_states, 1); %start with no controller  
  
policy\_states = zeros(num\_states, total\_tries);  
policy\_states(:, 1) = x0;  
policy\_inputs = zeros(num\_inputs, total\_tries);  
policy\_outputs = zeros(num\_outputs, total\_tries);  
k = 1;  
  
  
for iteration = 1:num\_controllers  
 %Reset Stacks each iteration  
 Uk\_stack = zeros(num\_collections, 1);  
 Xk\_stack = zeros(num\_collections, (Pj\_dim)^2);  
  
 %Compute stack infos  
 for collection = 1:num\_collections  
 %Simulate nature  
 %policy\_inputs(:, k) = rand(num\_inputs, 1); %can replicate the 'random state' approach by just randomizing inputs  
 policy\_inputs(:, k) = F\_policy(:, :, end) \* policy\_states(:, k) + rand\_range(num\_inputs, 1, -1, 1); %compute input and exploration term  
  
 %Next Step regardless of nature vs random  
 policy\_states(:, k + 1) = A \* policy\_states(:, k) + B \* policy\_inputs(:, k);  
 policy\_outputs(:, k) = C \* policy\_states(:, k) + D \* policy\_inputs(:, k);  
  
  
 %Construct stacks  
 xu\_stack = [policy\_states(:, k); policy\_inputs(:, k)];  
 xu\_next\_stack = [policy\_states(:, k + 1); F\_policy(:, :, end) \* policy\_states(:, k + 1)];  
 Xk\_stack(collection, :) = kron(xu\_stack', xu\_stack') - gamma \* kron(xu\_next\_stack', xu\_next\_stack');  
 Uk\_stack(collection, :) = policy\_states(:, k)' \* Q \* policy\_states(:, k) + policy\_inputs(:, k)' \* R \* policy\_inputs(:, k); %where Q and R come into play  
  
 %Move to next  
 k = k + 1;  
 end  
 % [P, S, V] = svd(Xk\_stack);  
 % rank(S), iteration %we do not need the stack to be full rank, but it  
 % must not drop ranks thorugh iterations  
 %Calculate P and new controller  
 PjS = pinv(Xk\_stack) \* Uk\_stack;  
 Pj = reshape(PjS, Pj\_dim, Pj\_dim); %undo the stack  
 Pj = 0.5 \* (Pj + Pj'); %to impose symmetry (significantly reduces error)  
 Pjuu = Pj((num\_states+1):end, (num\_states+1):end); %grab the bottom right quadrant, Puu  
 PjxuT = Pj((num\_states+1):end, 1:num\_states); %Grab bottom left, Pxu transpose  
 new\_F = -pinv(Pjuu) \* PjxuT; %definition of controller  
 F\_policy(:, :, end + 1) = new\_F; %add to end of list  
end  
  
%Plot / simulate past convergance (without exploration/noise) so actual  
%control possible  
for ndx = 1:converged\_k  
 %Simulate nature  
 policy\_inputs(:, k) = F\_policy(:, :, end) \* policy\_states(:, k);  
 policy\_states(:, k + 1) = A \* policy\_states(:, k) + B \* policy\_inputs(:, k);  
 policy\_outputs(:, k) = C \* policy\_states(:, k) + D \* policy\_inputs(:, k);  
 k = k + 1;  
end  
  
if plot\_all  
 plot\_two\_mass('Policy Iteration IO', 'Under Policy Iteration', policy\_outputs, policy\_inputs, update\_file\_path);  
 plot\_controller\_history('Policy Iteration Controller', 'Under Policy Iteration', F\_policy, -1, -1, update\_file\_path); %-1s denote to plot every state and input weight  
end  
  
  
%% Reinforcement Learning - Input Decoupling  
converged\_k = 20;  
num\_controllers = 5;  
Pj\_dim = (num\_states + 1); %effective input count is 1 now  
num\_collections = Pj\_dim^2;  
  
total\_tries = num\_controllers \* num\_collections + converged\_k;  
  
F\_decoupled = zeros(num\_inputs, num\_states, 1); %start with no controller  
  
decoupled\_states = zeros(num\_states, total\_tries);  
decoupled\_states(:, 1) = x0;  
decoupled\_inputs = zeros(num\_inputs, total\_tries);  
decoupled\_outputs = zeros(num\_outputs, total\_tries);  
k = 1;  
  
for iteration = 1:num\_controllers  
 for input\_num = 1:num\_inputs  
 %Reset Stacks each iteration  
 Uk\_stack = zeros(num\_collections, 1);  
 Xk\_stack = zeros(num\_collections, (Pj\_dim)^2);  
 F\_i = F\_decoupled(input\_num, :, end); %input on the current input of inspection  
 %Compute stack infos  
 for collection = 1:num\_collections  
 %Simulate nature  
 %decoupled\_inputs(:, k) = rand(num\_inputs, 1); %can replicate the 'random state' approach by just randomizing inputs  
 decoupled\_inputs(:, k) = F\_decoupled(:, :, end) \* decoupled\_states(:, k); %compute input  
 decoupled\_inputs(input\_num, k) = rand\_range(1, 1, -1, 1); %+decoupled\_inputs(input\_num, k); %  
  
 %Next Step regardless of nature vs random  
 decoupled\_states(:, k + 1) = A \* decoupled\_states(:, k) + B \* decoupled\_inputs(:, k);  
 decoupled\_outputs(:, k) = C \* decoupled\_states(:, k) + D \* decoupled\_inputs(:, k);  
  
  
 %Construct stacks  
 xu\_stack = [decoupled\_states(:, k); decoupled\_inputs(input\_num, k)];  
 xu\_next\_stack = [decoupled\_states(:, k + 1); F\_i \* decoupled\_states(:, k + 1)];  
 Xk\_stack(collection, :) = kron(xu\_stack', xu\_stack') - gamma \* kron(xu\_next\_stack', xu\_next\_stack');  
 Uk\_stack(collection, :) = decoupled\_states(:, k)' \* Q \* decoupled\_states(:, k) + decoupled\_inputs(:, k)' \* R \* decoupled\_inputs(:, k); %where Q and R come into play  
  
 %Move to next  
 k = k + 1;  
 end  
  
 %Calculate P and new controller  
 PjS = pinv(Xk\_stack) \* Uk\_stack;  
 Pj = reshape(PjS, Pj\_dim, Pj\_dim); %undo the stack  
 Pj = 0.5 \* (Pj + Pj'); %to impose symmetry (significantly reduces error)  
 Pjuu = Pj((num\_states+1):end, (num\_states+1):end); %grab the bottom right quadrant, Puu  
 PjxuT = Pj((num\_states+1):end, 1:num\_states); %Grab bottom left, Pxu transpose  
 new\_F\_i = -pinv(Pjuu) \* PjxuT; %definition of controller  
 F\_decoupled(:, :, end + 1) = F\_decoupled(:, :, end); %copy old controller  
 F\_decoupled(input\_num, :, end) = new\_F\_i; %update current input  
 end  
end  
  
%Plot / simulate past convergance (without exploration/noise) so actual  
%control possible  
for ndx = 1:converged\_k  
 %Simulate nature  
 decoupled\_inputs(:, k) = F\_decoupled(:, :, end) \* decoupled\_states(:, k);  
 decoupled\_states(:, k + 1) = A \* decoupled\_states(:, k) + B \* decoupled\_inputs(:, k);  
 decoupled\_outputs(:, k) = C \* decoupled\_states(:, k) + D \* decoupled\_inputs(:, k);  
 k = k + 1;  
end  
  
if plot\_all  
 plot\_two\_mass('Input Decoupling IO', 'Under Input Decoupling', decoupled\_outputs, decoupled\_inputs, update\_file\_path);  
 plot\_controller\_history('Input Decoupling Controller', 'Under Input Decoupling', F\_decoupled, -1, -1, update\_file\_path, true); %note only updates every other controller #  
end  
  
  
%% Compute Costs of Different Controllers  
cost\_max\_k = 500;  
%LQR  
cost\_lqr\_states = zeros(num\_states, cost\_max\_k);  
cost\_lqr\_inputs = zeros(num\_inputs, cost\_max\_k);  
%Policy  
cost\_policy\_states = zeros(num\_states, cost\_max\_k);  
cost\_policy\_inputs = zeros(num\_inputs, cost\_max\_k);  
%Decoupled  
cost\_decoupled\_states = zeros(num\_states, cost\_max\_k);  
cost\_decoupled\_inputs = zeros(num\_inputs, cost\_max\_k);  
  
%ICs  
cost\_decoupled\_states(:, 1) = x0;  
cost\_policy\_states(:, 1) = x0;  
cost\_lqr\_states(:, 1) = x0;  
  
%Cost values  
lqr\_cost = 0;  
policy\_cost = 0;  
decoupled\_cost = 0;  
  
%Simulate  
for k = 1:cost\_max\_k  
 %LQR  
 cost\_lqr\_inputs(:, k) = F\_lqr \* cost\_lqr\_states(:, k);  
 cost\_lqr\_states(:, k + 1) = A \* cost\_lqr\_states(:, k) + B \* cost\_lqr\_inputs(:, k);  
 lqr\_cost = lqr\_cost + (cost\_lqr\_inputs(:, k)' \* R \* cost\_lqr\_inputs(:, k) + cost\_lqr\_states(:, k)' \* Q \* cost\_lqr\_states(:, k));  
 %Policy  
 cost\_policy\_inputs(:, k) = F\_policy(:, :, end) \* cost\_policy\_states(:, k);  
 cost\_policy\_states(:, k + 1) = A \* cost\_policy\_states(:, k) + B \* cost\_policy\_inputs(:, k);  
 policy\_cost = policy\_cost + (cost\_policy\_inputs(:, k)' \* R \* cost\_policy\_inputs(:, k) + cost\_policy\_states(:, k)' \* Q \* cost\_policy\_states(:, k));  
 %Decoupled  
 cost\_decoupled\_inputs(:, k) = F\_decoupled(:, :, end) \* cost\_decoupled\_states(:, k);  
 cost\_decoupled\_states(:, k + 1) = A \* cost\_decoupled\_states(:, k) + B \* cost\_decoupled\_inputs(:, k);  
 decoupled\_cost = decoupled\_cost + (cost\_decoupled\_inputs(:, k)' \* R \* cost\_decoupled\_inputs(:, k) + cost\_decoupled\_states(:, k)' \* Q \* cost\_decoupled\_states(:, k));  
end  
  
%% Compare LQR, Policy, and Input Decoupled Controller  
lqr\_cost, policy\_cost, decoupled\_cost %all controllers should have same cost  
F\_lqr  
F\_policy(:, :, end)  
F\_decoupled(:, :, end)  
norm(F\_lqr - F\_policy(:, :, end))/numel(F\_lqr)  
norm(F\_lqr - F\_decoupled(:, :, end))/numel(F\_lqr)

## Basis Functions

%Basis Function Requirements and Applications  
%Noah Dunleavy  
%Honors Thesis under direction of Professor Minh Q. Phan  
%Thayer School of Engineering  
clc; clear;  
addpath('Saved Data', 'Functions');  
setDefaultFigProp();  
plot\_all = false;  
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  
if update\_file\_path ~= -1  
 keyboard %ensure we have to be very concious about ever updating all the images  
end  
  
%% Basics of Basis  
max\_cheby = 20; %for demo purposes, maximum number of cheby functions to generate  
  
p = 100; %resolution of the cheby functions  
cheby\_x = linspace(-1, 1, p)'; %define the cheby 'x'  
cheby\_functions = ones(p, max\_cheby);  
cheby\_functions(:, 2) = cheby\_x;  
for ndx = 3:max\_cheby %this is the recursive cheby generation. Could be from a file, but more helpful to see  
 cheby\_functions(:, ndx) = 2 \* cheby\_x .\* cheby\_functions(:, ndx - 1) - cheby\_functions(:, ndx - 2); %build cheby out  
end  
  
%Plot some of the core chebys  
core\_ndx = [1, 2, 8, 20]; %use 1 as the DC offset, then jst random  
if plot\_all  
 core\_cheby\_fig = figure('Name', 'Example Cheby Functions');  
 stairs(1:p, cheby\_functions(:, core\_ndx))  
 xlabel('Chebyschev Step')  
 ylabel('Amplitude')  
 title('Example Chebyshev Functions')  
 legend({'$T\_0$', '$T\_1$', '$T\_7$', '$T\_{19}$'}, 'Interpreter', 'latex'); %-1 off of matlab ndxs  
 save\_figure(update\_file\_path, core\_cheby\_fig);  
end  
  
%Plot the output signal  
rng('default'); rng(10);  
cheby\_weights = rand\_range(max\_cheby, 1, -3, 3);  
cheby\_signal = cheby\_functions \* cheby\_weights;  
if plot\_all  
 fig = figure('Name', 'Example Cheby Signal');  
 stairs(1:p, cheby\_signal)  
 xlabel('Chebyshev Step')  
 ylabel('Amplitude')  
 title('Example Chebyshev Composite Signal')  
 save\_figure(update\_file\_path, fig);  
end  
%Plot the Weights  
if plot\_all  
 fig = figure('Name', 'Cheby Weights');  
 temp\_plot = plot(1:(max\_cheby), cheby\_weights, 'Marker', 'o', 'MarkerFaceColor', 'auto');  
 xlabel('Coeffecient Number')  
 ylabel('Weight')  
 title('Chebyshev Coeffecient Weights')  
 xlim([1, max\_cheby])  
 xticks(1:(max\_cheby))  
 save\_figure(update\_file\_path, fig);  
end  
  
%% System Creation  
thesis\_system = load('Saved Data\thesis\_system.mat');  
A = thesis\_system.A;  
B = thesis\_system.B;  
C = thesis\_system.C;  
D = thesis\_system.D;  
num\_inputs = thesis\_system.num\_inputs;  
num\_outputs = thesis\_system.num\_outputs;  
num\_states = thesis\_system.num\_states;  
x0 = thesis\_system.x0;  
  
%% ILC Parameters  
num\_ilc\_states = p \* num\_outputs;  
num\_ilc\_inputs = p \* num\_inputs;  
  
[P, d] = P\_from\_ABCD(A, B, C, D, p, x0);  
  
%% Chebyshev Polynomial Creation  
max\_cheby = 10;  
cheby\_functions = generate\_chebyshev(num\_ilc\_inputs, max\_cheby);  
%In theory, there could be different resolution for inputs and outputs,  
%but since num\_inputs = num\_ouputs, we can use the same for both  
  
%% General Parameters  
%We have said p = 100  
%Controller in basis\_ilc\_sim defaults to 0.5H+  
%Built chebys out of 10 functions  
  
%% Setup: u\* in Basis Space, y\* is not  
T\_u = cheby\_functions; %cheby functions is only out of 10, so grab them all  
beta\_star = [1, 0.2, -0.3, 4, 0, 0, 0, -1, 0, 0]'; %manually set betas   
u\_star = T\_u \* beta\_star; %from basis to total space  
y\_star = P \* u\_star + d; %from input to output  
T\_y = T\_u; %equal - we do not want T\_y to capture y\*  
alpha\_star = pinv(T\_y) \* y\_star; %compute what our goal alpha is  
y\_Ty\_alpha = T\_y \* alpha\_star;  
u\_star\_Tu\_u\_star\_error = norm(y\_star - y\_Ty\_alpha); %if fully capured, should be 0  
u\_star\_Tu\_b\_star\_error = norm(u\_star - T\_u\*beta\_star); %should be zero since forced to be within  
  
%% Render: u\* in Basis Space, y\* is not  
if plot\_all  
 %u1\* Plot  
 fig = figure('Name', 'Goal Input 1');  
 stairs(0:(p-1), u\_star(1:2:end))  
 title('Goal Input 1')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Full input Partial output - Goal Input 1');  
 %U2\* plot  
 fig = figure('Name', 'Goal Input 2');  
 stairs(0:(p-1), u\_star(2:2:end))  
 title('Goal Input 2')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Full input Partial output - Goal Input 2');  
  
 %y1\* Plot  
 fig = figure('Name', 'Goal Output 1');  
 stairs(0:(p-1), y\_Ty\_alpha(1:2:end))  
 title('Goal Output 1 ')  
 hold on;  
 stairs(0:(p-1), y\_star(1:2:end))  
 hold off;  
 legend({'$\Phi\_y \alpha^\ast\_1$', '$\underline{y}\_1^\ast$'}, 'Interpreter', 'latex');  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Full input Partial output - Goal Output 1');  
  
 %y2\* Plot  
 fig = figure('Name', 'Goal Output 2');  
 stairs(0:(p-1), y\_Ty\_alpha(2:2:end))  
 title('Goal Output 2')  
 hold on;  
 stairs(0:(p-1), y\_star(2:2:end))  
 hold off;  
 legend({'$\Phi\_y \alpha^\ast\_2$', '$\underline{y}\_2^\ast$'}, 'Interpreter', 'latex');  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Full input Partial output - Goal Output 2');  
end  
  
%% Application: u\* in Basis Space, y\* is not  
num\_trials = 20;  
FIPO\_ILC = basis\_ilc\_sim(P, d, C\*x0, T\_u, T\_y, y\_star, num\_trials);  
norm(FIPO\_ILC(end).output\_error) %overall error  
proj\_u\_star = T\_u \* pinv(T\_u' \* T\_u) \* T\_u' \* u\_star; %project u\* onto Tu  
norm(proj\_u\_star - FIPO\_ILC(end).input)  
if plot\_all  
 to\_plot = [1, 2, 5, 20];  
 plot\_ilc\_coeffecients('Full Input Partial Output', 'Full Input Partial Output', FIPO\_ILC, to\_plot, beta\_star, find(beta\_star)', 5, update\_file\_path);  
 plot\_dual\_ilc('Full Input Partial Output', 'Full Input Partial Output', FIPO\_ILC, y\_star, u\_star, to\_plot, update\_file\_path);  
end   
  
%% Setup: y\* in Basis Space, u\* is not  
%Keep the same Tu and Ty  
alpha\_star = beta\_star; %new alpha\_star is the old beta\_star  
y\_star = T\_y \* alpha\_star;  
u\_star = pinv(P) \* (y\_star - d); %calcuate the input that gets us here  
beta\_star = pinv(T\_u) \* u\_star; %compute what our goal alpha is  
u\_Tu\_beta = T\_u \* beta\_star;  
u\_star\_Tu\_u\_star\_error = norm(u\_star - u\_Tu\_beta) %if fully capured, should be 0  
  
%% Render: y\* in Basis Space, u\* is not  
if plot\_all  
 %u1\* Plot  
 fig = figure('Name', 'Goal Input 1');  
 stairs(0:(p-1), u\_Tu\_beta(1:2:end))  
 hold on;  
 stairs(0:(p-1), u\_star(1:2:end))  
 hold off;  
 legend({'$\Phi\_u \beta^\ast\_1$', '$\underline{u}\_1^\ast$'}, 'Interpreter', 'latex');   
 title('Goal Input 1')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Partial input Full output - Goal Input 1');  
   
 %U2\* plot  
 fig = figure('Name', 'Goal Input 2');  
 stairs(0:(p-1), u\_Tu\_beta(2:2:end))  
 hold on;  
 stairs(0:(p-1), u\_star(2:2:end))  
 hold off;  
 legend({'$\Phi\_u \beta^\ast\_1$', '$\underline{u}\_1^\ast$'}, 'Interpreter', 'latex');   
 title('Goal Input 2')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Partial input Full output - Goal Input 2');  
   
 %y1\* Plot  
 fig = figure('Name', 'Goal Output 1');  
   
 stairs(0:(p-1), y\_star(1:2:end))  
 title('Goal Output 1')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Partial input Full output - Goal Output 1');   
   
 %y2\* Plot  
 fig = figure('Name', 'Goal Output 2');  
   
 stairs(0:(p-1), y\_star(2:2:end))  
 title('Goal Output 2')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Partial input Full output - Goal Output 2');   
end  
  
%% Application: y\* in Basis Space, u\* is not  
%Same number of trials  
PIFO\_ILC = basis\_ilc\_sim(P, d, C\*x0, T\_u, T\_y, y\_star, num\_trials);  
norm(PIFO\_ILC(end).output\_error)  
proj\_u\_star = T\_u \* pinv(T\_u' \* T\_u) \* T\_u' \* u\_star; %project u\* onto Tu  
norm(proj\_u\_star - PIFO\_ILC(end).input)  
if plot\_all  
 to\_plot = [1, 2, 5, 20]; %same to\_plot  
 plot\_ilc\_coeffecients('Partial Input Full Output', 'Partial Input Full Output', PIFO\_ILC, to\_plot, -1, 5, find(alpha\_star)', update\_file\_path);  
 plot\_dual\_ilc('Partial Input Full Output', 'Partial Input Full Output', PIFO\_ILC, y\_star, u\_star, to\_plot, update\_file\_path);  
end  
  
%% Setup: ny < nu  
%Keep the same Tu and Ty  
beta\_star = alpha\_star; %swap back the coeffecients  
u\_star = T\_u \* beta\_star;  
y\_star = P\*u\_star + d;  
T\_y = y\_star; %set our output basis to be pmx1  
  
%% Render: ny < nu  
if plot\_all  
 %Inputs already rendered  
  
 %y1\* Plot  
 fig = figure('Name', 'Goal Output 1');  
   
 stairs(0:(p-1), y\_star(1:2:end))  
 title('Goal Output 1 ')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Full input Single output - Goal Output 1');  
  
 %y2\* Plot  
 fig = figure('Name', 'Goal Output 2');  
   
 stairs(0:(p-1), y\_star(2:2:end))  
 title('Goal Output 2')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Full input Single output - Goal Output 2');  
end  
  
%% Application: ny < nu  
%Same number of trials  
FISO\_ILC = basis\_ilc\_sim(P, d, C\*x0, T\_u, T\_y, y\_star, num\_trials);  
norm(FISO\_ILC(end).output\_error)  
proj\_u\_star = T\_u \* pinv(T\_u' \* T\_u) \* T\_u' \* u\_star; %project u\* onto Tu  
norm(proj\_u\_star - FISO\_ILC(end).input)  
if plot\_all  
 to\_plot = [1, 2, 5, 20]; %same to\_plot  
 plot\_ilc\_coeffecients('Full Input Single Output', 'Full Input Single Output', FISO\_ILC, to\_plot, beta\_star, find(beta\_star)', 1, update\_file\_path);  
 plot\_dual\_ilc('Full Input Single Output', 'Full Input Single Output', FISO\_ILC, y\_star, u\_star, to\_plot, update\_file\_path);  
end  
  
%% Setup: ny > nu  
T\_y = T\_u; %set the basis spaces back to both be chebys  
alpha\_star = beta\_star; %swap back the coeffecients  
y\_star = T\_y \* alpha\_star;  
u\_star = pinv(P) \* (y\_star - d);  
T\_u = u\_star; %set our output basis to be pmx1  
  
%% Render: ny > nu  
if plot\_all  
 %u1\* Plot  
 fig = figure('Name', 'Goal Input 1');  
 stairs(0:(p-1), u\_star(1:2:end))  
 title('Goal Input 1')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Single input Full output - Goal Input 1');  
   
 %U2\* plot  
 fig = figure('Name', 'Goal Input 2');  
 stairs(0:(p-1), u\_star(2:2:end))  
 title('Goal Input 2')  
 %subtitle('Defined in $\Phi\_u$')  
 xlabel('Step Number (k)')  
 ylabel('Amplitude')  
 save\_figure(update\_file\_path, fig, 'Single input Full output - Goal Input 2');  
  
 %Outputs already rendered  
end  
  
%% Application: ny > nu  
%Same number of trials  
SIFO = basis\_ilc\_sim(P, d, C\*x0, T\_u, T\_y, y\_star, num\_trials);  
norm(SIFO(end).output\_error)  
proj\_u\_star = T\_u \* pinv(T\_u' \* T\_u) \* T\_u' \* u\_star; %project u\* onto Tu  
norm(proj\_u\_star - SIFO(end).input)  
if plot\_all  
 to\_plot = [1, 2, 5, 20]; %same to\_plot  
 plot\_ilc\_coeffecients('Single Input Full Output', 'Single Input Full Output', SIFO, to\_plot, 1, 1, find(alpha\_star)', update\_file\_path);  
 plot\_dual\_ilc('Single Input Full Output', 'Single Input Full Output', SIFO, y\_star, u\_star, to\_plot, update\_file\_path);  
end  
  
%% Setup: Rolling Tu  
num\_rolling\_basis\_input = 3; %num\_basis - 1  
num\_basis\_output = num\_rolling\_basis\_input + 1; %why do anything worse than optimal  
T\_u\_ndx = 2:(num\_rolling\_basis\_input+1); %initilaize off a littl  
T\_u\_full = cheby\_functions; %grab max functions we will try  
T\_y\_full = T\_u\_full;  
T\_u = T\_u\_full(:, [1, T\_u\_ndx]); %grab the first couple of basis  
T\_y = T\_u;  
%Alpha star assigned rolling  
beta\_star = [1, 0.2, -0.3, 4, 0, 0, 0, -1, 0, 0]'; %manually set betas   
u\_star = T\_u\_full \* beta\_star; %from basis to total space  
y\_star = P \* u\_star + d; %from input to output  
  
num\_loops = 5;%how many loops through we'll do  
num\_rolls\_per\_loop = max\_cheby / num\_rolling\_basis\_input;  
num\_tries = ceil(num\_loops \* num\_rolls\_per\_loop);  
  
%% Render: Rolling Tu  
if plot\_all  
 %Inputs already rendered  
  
 %Outputs already rendered  
end  
  
%% Application: Rolling Tu  
%Same number of trials  
rolling\_ILC = [];  
learned\_inputs = zeros(num\_ilc\_inputs, num\_tries + 1);  
  
for ndx = 1:num\_tries  
 %Run the ilc trial with basis spaces  
 temporary\_ILC = basis\_ilc\_sim(P, d, C\*x0, T\_u, T\_y, y\_star, num\_trials); %sim out a trial  
 rolling\_ILC = [rolling\_ILC, temporary\_ILC]; %tack it on to previous results  
   
 %What input did we learn  
 learned\_u = temporary\_ILC(end).input;  
 learned\_inputs(:, ndx+1) = learned\_u;  
 %Take learned input, make it a bsis function  
 T\_u\_ndx = mod(T\_u\_ndx + num\_rolling\_basis\_input - 1, max\_cheby) + 1; %update which new functions we'll add  
 T\_u = [learned\_u, T\_u\_full(:, T\_u\_ndx)];  
 %T\_y doesnt matter  
 T\_y = T\_u;  
end  
  
output\_error = norm(rolling\_ILC(end).output\_error)  
input\_error = norm(T\_u(:, 1) - u\_star)  
  
if plot\_all  
 to\_plot = 4;   
 plot\_ilc\_coeffecients('Rolling Input Basis', 'Rolling Input Basis', rolling\_ILC, to\_plot, -1, num\_rolling\_basis\_input + 1, num\_rolling\_basis\_input + 1, update\_file\_path);  
 plot\_dual\_ilc('Rolling Input Basis', 'Rolling Input Basis', rolling\_ILC, y\_star, u\_star, to\_plot, update\_file\_path);  
end

## Derivation and Demonstration of Conjugate Basis Functions

%Demonstration and Determination of Conjugate Basis Functions Noah Dunleavy  
%Honors Thesis under direction of Professor Minh Q. Phan Thayer School of  
%Engineering  
clc; clear;  
addpath('Saved Data', 'Functions');  
setDefaultFigProp();  
plot\_all = true;  
update\_file\_path = 'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis Images\Derive and Demo Conjugate Basis'; %set this to the save location if want to update figures, or -1 if not  
if update\_file\_path ~= -1  
 keyboard %ensure we have to be very concious about ever updating all the images  
end  
  
%% System Creation  
thesis\_system = load('Saved Data\thesis\_system.mat');  
A = thesis\_system.A;  
B = thesis\_system.B;  
C = thesis\_system.C;  
D = thesis\_system.D;  
num\_inputs = thesis\_system.num\_inputs;  
num\_outputs = thesis\_system.num\_outputs;  
num\_states = thesis\_system.num\_states;  
x0 = thesis\_system.x0;  
  
%% ILC Parameters  
p = 100;  
  
num\_ilc\_states = p \* num\_outputs;  
num\_ilc\_inputs = p \* num\_inputs;  
  
[P, d] = P\_from\_ABCD(A, B, C, D, p, x0);  
  
%% Chebyshev Generation  
max\_cheby = 10;  
cheby\_functions = generate\_chebyshev(num\_ilc\_inputs, max\_cheby);  
  
%% Create u\* and y\*  
beta\_star = [1, 0.2, -0.3, 4, 0, 0, 0, -1, 0, 0]'; %manually set betas   
u\_star = cheby\_functions \* beta\_star; %from basis to total space  
y\_star = P \* u\_star + d; %from input to output  
  
%Learning Weights  
Q = 100\*eye(num\_ilc\_states);  
R = 0\*eye(num\_ilc\_inputs);  
  
out1\_ndx = (1:2:(num\_ilc\_states)); %convert stacked output to individual positions  
out2\_ndx = (2:2:(num\_ilc\_states));  
  
%% Conjugate Basis Creation (Batch)  
num\_conjugate\_basis = max\_cheby; %how many of the cheby functions we will stimulate the system with / how many conjugate functions to make  
batch\_input = cheby\_functions; %for batch input (and code simplicity)  
%Batch input must be full rank to ensure wronskian exists  
  
%Generate Batch output  
batch\_outputs\_delta = P \* batch\_input; %do not include the d term, because we want the difference in outputs, which excludes d  
  
W = batch\_outputs\_delta' \* Q \* batch\_outputs\_delta; %W matrix to get wronskian from  
  
rho\_batch = chol(W); %cholesky decomposition of W to get the optimal coeffecients for the batch  
T\_b\_batch = batch\_input / (rho\_batch); %/ is same as \* inv()  
H\_b\_batch = batch\_outputs\_delta / (rho\_batch); %  
beta\_batch = H\_b\_batch' \* Q \* (y\_star - d); %determined optimal weights for given basis functions (off of e\_0)  
  
%Generate final output  
batch\_learned\_u = T\_b\_batch \* beta\_batch;  
batch\_learned\_y = P \* (batch\_learned\_u) + d;  
  
%Verify conjunct condition  
batch\_conj\_cond = T\_b\_batch' \* (R + P' \* Q \* P) \* T\_b\_batch; %should be idenitity matrix  
  
%% Conjugate Basis Creation (Iterative)  
%We already defined Q and R above We have set our input sequences through  
%cheby\_functions  
  
%Initial experiments  
u0 = zeros(num\_ilc\_inputs, 1);  
y0 = d;  
u1 = cheby\_functions(:, 1);  
y1 = P\*u1 + d;  
  
%Compute del u1 and del y1 explictly (P\*del\_u also works)  
Episode(1).del\_u = u1 - u0;  
Episode(1).del\_y = y1 - y0;  
  
%Compute W  
W = Episode(1).del\_u' \* R \* Episode(1).del\_u + Episode(1).del\_y' \* Q \* Episode(1).del\_y;  
  
%rho, phi\_1, h\_1, and beta\_1  
Episode(1).rho = chol(W);  
phi\_1 = Episode(1).del\_u \* Episode(1).rho^-1;   
h\_1 = Episode(1).del\_y \* Episode(1).rho^-1;  
beta\_1 = h\_1' \* Q \* (y\_star - d); %use e0 for all  
  
Episode(1).Phi = phi\_1;  
Episode(1).Hb = h\_1;  
Episode(1).Betas = beta\_1;  
  
%Third experiment for phi\_2  
u2 = cheby\_functions(:, 2);  
y2 = P\*u2 + d;  
  
%Define all our deltas wrt u0 and y0  
Episode(2).del\_u = u2 - u0;  
Episode(2).del\_y = y2 - y0;  
  
%Episode(2).rho, phi\_2, h\_2, and beta\_2  
Episode(2).rho(1) = 1/Episode(1).rho(1) \* (Episode(1).del\_u' \* R \* Episode(2).del\_u + Episode(1).del\_y' \* Q \* Episode(2).del\_y);  
Episode(2).gamma = Episode(2).rho(1)^2;  
Episode(2).rho(2) = sqrt(Episode(2).del\_u' \* R \* Episode(2).del\_u + Episode(2).del\_y' \* Q \*Episode(2).del\_y - Episode(2).gamma);  
phi\_2 = (1/Episode(2).rho(2)) \* (Episode(2).del\_u - phi\_1 \* Episode(2).rho(1));  
h\_2 = 1/Episode(2).rho(2) \* (Episode(2).del\_y - (h\_1 \* Episode(2).rho(1)));  
beta\_2 = h\_2' \* Q \* (y\_star - d);  
  
Episode(2).Phi = [Episode(1).Phi, phi\_2];  
Episode(2).Hb = [Episode(1).Hb, h\_2];  
Episode(2).Betas = [Episode(1).Betas; beta\_2];  
  
%Fourth experiment for phi\_3  
u3 = cheby\_functions(:, 3);  
y3 = P\*u3 + d;  
  
%Define all our deltas wrt u0 and y0  
Episode(3).del\_u = u3 - u0;  
Episode(3).del\_y = y3 - y0;  
  
%Episode(3).rho, phi\_3, h\_3, and beta\_3  
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);  
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)));  
Episode(3).gamma = Episode(3).rho(1)^2 + Episode(3).rho(2)^2;  
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);  
phi\_3 = (1/Episode(3).rho(3)) \* (Episode(3).del\_u - (phi\_1 \* Episode(3).rho(1) + phi\_2 \* Episode(3).rho(2)) );  
h\_3 = 1/Episode(3).rho(3) \* (Episode(3).del\_y - ((h\_1 \* Episode(3).rho(1) + h\_2 \* Episode(3).rho(2))));  
beta\_3 = h\_3' \* Q \* (y\_star - d);  
  
Episode(3).Phi = [Episode(2).Phi, phi\_3];  
Episode(3).Hb = [Episode(2).Hb, h\_3];  
Episode(3).Betas = [Episode(2).Betas; beta\_3];  
  
for b = 3:(max\_cheby-1) %for the rest of the trials   
 Episode = generate\_iterative\_conjugate(P, d, y\_star, Episode, Q, R);  
end  
  
%% Visualize Approaches  
if plot\_all  
 %Plot control under batch (output, we already have input%  
 fig = figure('Name', 'Batched Learned Input');  
 plot(batch\_learned\_y(out1\_ndx), batch\_learned\_y(out2\_ndx));  
 hold on;  
 plot(y\_star(out1\_ndx), y\_star(out2\_ndx));  
 hold off;  
 legend('Learned', 'Goal')  
 xlabel('Mass 1 Position (m)')  
 ylabel('Mass 2 Position (m)')  
 title(sprintf('Batch LQL Output - 10 Conjugate Basis Functions'))  
 axis equal  
 save\_figure(update\_file\_path, fig);  
  
 %Iterative Plots Plot after 1  
 learned\_u = Episode(1).Phi \* Episode(1).Betas;  
 learned\_y = P \* learned\_u + d;  
 fig = figure('Name', 'Iterative Learned Input - 1 Basis');  
 plot(learned\_y(out1\_ndx), learned\_y(out2\_ndx));  
 hold on;  
 plot(y\_star(out1\_ndx), y\_star(out2\_ndx));  
 hold off;  
 legend('Learned', 'Goal')  
 xlabel('Mass 1 Position (m)')  
 ylabel('Mass 2 Position (m)')  
 title(('Output for LQL with 1 Conjugate Basis Function'))  
 axis equal  
 save\_figure(update\_file\_path, fig);  
  
 %Plot after 3  
 learned\_u = Episode(3).Phi \* Episode(3).Betas;  
 learned\_y = P \* learned\_u + d;  
 fig = figure('Name', 'Iterative Learned Input - 3 Basis');  
 plot(learned\_y(out1\_ndx), learned\_y(out2\_ndx));  
 hold on;  
 plot(y\_star(out1\_ndx), y\_star(out2\_ndx));  
 hold off;  
 legend('Learned', 'Goal')  
 xlabel('Mass 1 Position (m)')  
 ylabel('Mass 2 Position (m)')  
 title(('Output for LQL with 3 Conjugate Basis Functions'))  
 axis equal  
 save\_figure(update\_file\_path, fig);  
  
 %Plot after 8  
 learned\_u = Episode(8).Phi \* Episode(8).Betas;  
 learned\_y = P \* learned\_u + d;  
 fig = figure('Name', 'Iterative Learned Input - 8 Basis');  
 plot(learned\_y(out1\_ndx), learned\_y(out2\_ndx));  
 hold on;  
 plot(y\_star(out1\_ndx), y\_star(out2\_ndx));  
 hold off;  
 legend('Learned', 'Goal')  
 xlabel('Mass 1 Position (m)')  
 ylabel('Mass 2 Position (m)')  
 title(('Output for LQL with 8 Conjugate Basis Functions'))  
 axis equal  
 save\_figure(update\_file\_path, fig);  
end  
  
%% Arbitrary Shape  
p = 200;  
loadedShape = load('Saved Data\heart\_p200.mat'); %read in the file  
[P\_arb, d\_arb] = P\_from\_ABCD(A, B, C, D, p, x0);  
num\_ilc\_states = height(P\_arb);  
num\_ilc\_inputs = width(P\_arb);  
Q = 100 \* eye(num\_ilc\_states);  
R = 0 \* eye(num\_ilc\_inputs);  
  
scale = 10;  
y\_star\_x = scale \* loadedShape.drawn\_x';  
y\_star\_y = scale \* loadedShape.drawn\_y';  
goal\_matrix = [y\_star\_x, y\_star\_y]; %stack inputs next to eachother  
y\_star = reshape(goal\_matrix', [], 1); %combine the seperate goals of each output into one vertical vector, alternating as necessary  
  
num\_to\_try = 100;  
[arb\_conj, arb\_betas] = generate\_conjugate(num\_ilc\_inputs, num\_to\_try, P\_arb, Q, R, d\_arb, y\_star);  
  
%% Plot Arbitrary Progression  
num\_basis\_to\_include = [1, 5, 10, 20, 50, 100, num\_to\_try];  
  
if plot\_all  
 for num\_basis = num\_basis\_to\_include  
 learned\_u = arb\_conj(:, 1:num\_basis) \* arb\_betas(1:num\_basis);  
 learned\_y = P\_arb \* learned\_u + d\_arb;  
 fig = figure('Name', sprintf('Iterative Learned Input for Specified Shape - %.d Basis', num\_basis));  
 plot(learned\_y(1:2:end), learned\_y(2:2:end));  
 hold on;  
 plot(y\_star\_x, y\_star\_y);  
 hold off;  
 legend('Learned', 'Goal')  
 xlabel('Mass 1 Position (m)')  
 ylabel('Mass 2 Position (m)')  
 title(sprintf('Shaped Output for LQL with %.d Conjugate Basis Functions', num\_basis))  
 axis equal  
 %save\_figure(update\_file\_path, fig);  
 end  
end

## RL on Conjugate Basis

%Appllication of RL to the ILC Problem with Conjugate  
%Basis Functions  
%Noah Dunleavy  
%Honors Thesis under direction of Professor Minh Q. Phan  
%Thayer School of Engineering  
clc; clear;  
addpath('Saved Data', 'Functions');  
setDefaultFigProp();  
plot\_all = false;  
update\_file\_path = -1;%'C:\Users\noahd\OneDrive\Desktop\Thesis\Thesis Images\RL on Conjugate'; %set this to the save location if want to update figures, or -1 if not  
if update\_file\_path ~= -1  
 keyboard %ensure we have to be very concious about ever updating all the images  
end  
  
%% System Creation  
thesis\_system = load('Saved Data\thesis\_system.mat');  
A = thesis\_system.A;  
B = thesis\_system.B;  
C = thesis\_system.C;  
D = thesis\_system.D;  
num\_inputs = thesis\_system.num\_inputs;  
num\_outputs = thesis\_system.num\_outputs;  
num\_states = thesis\_system.num\_states;  
x0 = thesis\_system.x0;  
  
%% ILC Parameters  
p = 10;  
  
num\_ilc\_states = p \* num\_outputs;  
num\_ilc\_inputs = p \* num\_inputs;  
  
[P, d] = P\_from\_ABCD(A, B, C, D, p, x0);  
  
%% Goal Definition  
y\_star\_x = cos(2 \* pi \* (0:(p-1)) / p)';  
y\_star\_y = sin(2 \* pi \* (0:(p-1)) / p)';  
goal\_matrix = [y\_star\_x, y\_star\_y]; %stack inputs next to eachother  
y\_star = reshape(goal\_matrix', [], 1); %combine the seperate goals of each output into one vertical vector, alternating as necessary  
  
%% Conjugate Basis Creation (Batch)  
%Learning Weights for the Batch Learning  
Q\_batch = 100\*eye(num\_ilc\_states);  
R\_batch = 0 \* eye(num\_ilc\_inputs);  
max\_conj = num\_ilc\_inputs;  
[conjugate\_basis\_functions, conjugate\_betas] = generate\_conjugate(num\_ilc\_inputs, max\_conj, P, Q\_batch, R\_batch, d, y\_star);  
  
%% Conjugate Basis Functions Definition  
num\_basis\_output = max\_conj; %full definition / resolution  
num\_basis\_input = num\_basis\_output;  
  
output\_basis\_functions = conjugate\_basis\_functions; %full resolution output %conjugate\_basis\_functions;  
input\_basis\_functions = conjugate\_basis\_functions;  
  
output\_basis\_functions\_pinv = pinv(output\_basis\_functions); %since done a lot, just compute once and use as constant  
  
%% Render Goal  
if (plot\_all)  
 fig = figure('Name', 'Goal Output for Arbitrary Basis ILC');  
 plot(y\_star(1:2:end), y\_star(2:2:(end+1)))  
 axis equal  
 %Should have already saved in RL on ILC  
 %save\_figure(update\_file\_path, fig);  
end  
  
%% F\_lqr with Varying Basis Functions  
%RL Parameters  
Q = 100 \* eye(num\_ilc\_states);  
R = 10 \* eye(num\_ilc\_inputs);  
gamma = 0.8;  
  
  
output\_basis = output\_basis\_functions;  
A\_ilc\_basis = eye(num\_ilc\_states); %fixed - capture the whole output  
for ndx = 1:3  
 current\_basis = ndx;  
 if ndx == 3  
 current\_basis = [1, 2];  
 end  
 input\_basis = conjugate\_basis\_functions(:, current\_basis);  
 H = -pinv(output\_basis) \* P \* input\_basis;  
 F\_lqr = discounted\_LQR(A\_ilc\_basis, -H, gamma, Q, R(current\_basis, current\_basis));  
end  
  
%% Full Resolution Demo  
FIFO\_Trials = basis\_ilc\_sim(P, d, C\*x0, conjugate\_basis\_functions, conjugate\_basis\_functions, y\_star, 20);  
if plot\_all  
 plot\_dual\_ilc('FIFO Conjugate Basis', 'Full Resolution Conjugate Basis', FIFO\_Trials, y\_star, -1, 4, update\_file\_path);  
end  
  
%% Fixed Resolution T\_y = I  
%Fix the output basis  
num\_output\_basis = num\_ilc\_inputs; %assume worst case  
output\_basis\_functions = eye(num\_output\_basis);  
%Conjugate episodes holder  
Episode = [];   
%Setup learning parameters  
Q = 100 \* eye(num\_output\_basis);  
big\_R = 1 \* eye(num\_ilc\_inputs);  
small\_R = 1e-6 \* eye(num\_ilc\_inputs);  
gamma = 0.8;  
exploration\_mag = 10;  
num\_controllers = 5;  
num\_converged = 10;  
fixed\_I\_Trials\_big\_R = []; %hold all the trials  
fixed\_I\_Trials\_small\_R = [];   
F\_big\_R = zeros(num\_ilc\_inputs, num\_output\_basis);  
F\_small\_R = zeros(num\_ilc\_inputs, num\_output\_basis);  
for phi\_num = 1:num\_ilc\_inputs %try every combo possible  
 %Update basis space  
 Episode = generate\_iterative\_conjugate(P, d, y\_star, Episode, Q, 0);  
 phi = Episode(phi\_num).Phi(:, phi\_num); %get the most recent phi  
 %Learn the controller with big R  
 [temp\_Trial, F\_phi, ~, ~] = policy\_ilc(P, d, C, D, x0, y\_star, gamma, Q, big\_R(phi\_num, phi\_num), num\_controllers, exploration\_mag, phi, output\_basis\_functions, num\_converged);  
 F\_big\_R(phi\_num, :) = F\_phi(:, :, end); %save the controller in the stack  
 fixed\_I\_Trials\_big\_R = [fixed\_I\_Trials\_big\_R, temp\_Trial]; %save teh trial info  
   
 %Controller with small R  
 [temp\_Trial, F\_phi, ~, ~] = policy\_ilc(P, d, C, D, x0, y\_star, gamma, Q, small\_R(phi\_num, phi\_num), num\_controllers, exploration\_mag, phi, output\_basis\_functions, num\_converged);  
 F\_small\_R(phi\_num, :) = F\_phi(:, :, end); %save the controller in the stack  
 fixed\_I\_Trials\_small\_R = [fixed\_I\_Trials\_small\_R, temp\_Trial]; %save teh trial info  
end  
F\_lqr\_big\_R = discounted\_LQR(eye(num\_output\_basis), -pinv(output\_basis\_functions) \* P \* Episode(end).Phi, gamma, Q, big\_R);  
F\_lqr\_small\_R = discounted\_LQR(eye(num\_output\_basis), -pinv(output\_basis\_functions) \* P \* Episode(end).Phi, gamma, Q, small\_R);  
  
  
%Show ow they compare  
sprintf('F\_lqr vs Policy when Fixed Output Basis of I with Big R')  
lqr = F\_lqr\_big\_R([1, 2, 19, 20], [1, 2, 19, 20])  
policy = F\_big\_R([1, 2, 19, 20], [1, 2, 19, 20])  
  
sprintf('F\_lqr vs Policy when Fixed Output Basis of I with Small R')  
lqr = F\_lqr\_small\_R([1, 2, 19, 20], [1, 2, 19, 20])  
policy = F\_small\_R([1, 2, 19, 20], [1, 2, 19, 20])  
  
%Plot the path through learning and straight application  
if plot\_all  
 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);  
  
 one\_trial = basis\_ilc\_sim(P, d, C\*x0, Episode(end).Phi(:, 1), output\_basis\_functions, y\_star, 10, F\_big\_R(1, :));   
 plot\_dual\_ilc('Controller Application of One Conjugate Input Basis with Fixed I Output Basis', 'One-Controller Fixed I Output Basis', one\_trial, y\_star, -1, 4, update\_file\_path);  
  
  
 half\_trial = basis\_ilc\_sim(P, d, C\*x0, Episode(end).Phi(:, 1:10), output\_basis\_functions, y\_star, 10, F\_big\_R(1:10, :));   
 plot\_dual\_ilc('Half Controller Application of Conjugate Input Basis with Fixed I Output Basis', 'Half-Controller Fixed I Output Basis', half\_trial, y\_star, -1, 4, update\_file\_path);  
  
 seven\_five\_trial = basis\_ilc\_sim(P, d, C\*x0, Episode(end).Phi(:, 1:15), output\_basis\_functions, y\_star, 10, F\_big\_R(1:15, :));   
 plot\_dual\_ilc('Three-Quarter Controller Application of Conjugate Input Basis with Fixed I Output Basis', 'Three-Quarter Fixed I Output Basis', seven\_five\_trial, y\_star, -1, 4, update\_file\_path);  
  
 final\_Trial = basis\_ilc\_sim(P, d, C\*x0, Episode(end).Phi, output\_basis\_functions, y\_star, 10, F\_big\_R);   
 plot\_dual\_ilc('Controller Application of Conjugate Input Basis with Fixed I Output Basis', 'Fixed I Output Basis', final\_Trial, y\_star, -1, 4, update\_file\_path);  
end  
  
%% Fixed Resolution T\_y = y\*  
%Fix the output basis  
num\_output\_basis = 1;   
output\_basis\_functions = 100 \* y\_star;  
%Conjugate episodes holder  
clear 'Episode';  
Episode = [];   
%Setup learning parameters  
Q = 1000 \* eye(num\_output\_basis);  
R\_y\_star = 10 \* eye(num\_ilc\_inputs);  
gamma = 0.8;  
exploration\_mag = 1;  
num\_controllers = 2;  
num\_converged = 10;  
y\_star\_fixed\_I\_Trials = []; %hold all the trials  
F = zeros(num\_ilc\_inputs, num\_output\_basis);  
for phi\_num = 1:num\_ilc\_inputs %try every combo possible  
 %Update basis space  
 Episode = generate\_iterative\_conjugate(P, d, y\_star, Episode, Q, 0);  
 phi = Episode(phi\_num).Phi(:, phi\_num); %get the most recent phi  
 %Learn the controller  
 [temp\_Trial, F\_phi, ~, ~] = policy\_ilc(P, d, C, D, x0, y\_star, gamma, Q, R\_y\_star(phi\_num, phi\_num), num\_controllers, exploration\_mag, phi, output\_basis\_functions, num\_converged);  
 F(phi\_num, :) = F\_phi(:, :, end); %save the controller in the stack  
 y\_star\_fixed\_I\_Trials = [y\_star\_fixed\_I\_Trials, temp\_Trial]; %save teh trial info  
end  
F\_lqr\_y\_star = discounted\_LQR(eye(num\_output\_basis), -pinv(output\_basis\_functions) \* P \* Episode(end).Phi, gamma, Q, R\_y\_star);  
  
  
%Show ow they compare  
sprintf('F\_lqr vs Policy when Fixed Output Basis of y\*')  
lqr = F\_lqr\_y\_star([1, 2, 19, 20])  
policy = F([1, 2, 19, 20])  
  
num\_sim = 100;  
y\_star\_Trial = basis\_ilc\_sim(P, d, C\*x0, Episode(end).Phi, output\_basis\_functions, y\_star, num\_sim, F);   
%y\_star\_lqr\_Trial = basis\_ilc\_sim(P, d, C\*x0, Episode(end).Phi, output\_basis\_functions, y\_star, num\_sim, F\_lqr\_y\_star);   
if plot\_all  
 %plot\_dual\_ilc('Controller Application of Conjugate Input Basis with y star Output Basis', 'Fixed y\* Output Basis RL', y\_star\_Trial, y\_star, -1, 4, update\_file\_path);  
 plot\_dual\_ilc('LQR Controller Application of Conjugate Input Basis with y star Output Basis', 'Fixed y\* Output Basis LQR', y\_star\_Trial, y\_star, -1, 4, update\_file\_path);  
end  
  
%% Fixed Resolution T\_y = Phi^b  
  
fixed\_phi\_R = eye(num\_ilc\_inputs);  
exploration\_mag = 1; %little extra noise to be safe  
T\_y\_scale = 10;  
  
num\_converged = 0;  
num\_controllers = 5;  
%Generate our conjugate basis  
conjugate\_basis\_functions = Episode(end).Phi; %use our previous iteratively calculated instead of doing a batch - computationally faster and more consistent  
%Fix the output basis  
num\_output\_basis = num\_ilc\_inputs; %assume worst case  
output\_basis\_functions = conjugate\_basis\_functions(:, 1:num\_output\_basis) \* T\_y\_scale;  
%Setup learning parameters  
%keep same as before  
fixed\_Phi\_b\_Trials = []; %hold all the trials  
F = zeros(num\_ilc\_inputs, num\_output\_basis);  
  
for phi\_num = 1:num\_ilc\_inputs %try every combo possible  
 %Update basis space  
 phi = conjugate\_basis\_functions(:, phi\_num);  
 %Learn the controller  
 [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);  
 F(phi\_num, :) = F\_phi(:, :, end); %save the controller in the stack  
 fixed\_Phi\_b\_Trials = [fixed\_Phi\_b\_Trials, temp\_Trial]; %save teh trial info  
end  
F\_lqr = discounted\_LQR(eye(num\_basis\_output), -pinv(output\_basis\_functions) \* P \* conjugate\_basis\_functions, gamma, Q, fixed\_phi\_R);  
%Show ow they compare  
sprintf('F\_lqr vs Policy when Fixed Conjugate Output Basis')  
lqr = F\_lqr([1, 2, 19, 20], [1, 2, 19, 20])  
policy = F([1, 2, 19, 20], [1, 2, 19, 20])  
  
%Plot the path  
if plot\_all  
 plot\_dual\_ilc('Scaled Conjugate Basis', 'Scaled Conjugate Basis', fixed\_Phi\_b\_Trials, y\_star, -1, 4, update\_file\_path);  
  
 final\_conj\_Trial = basis\_ilc\_sim(P, d, C\*x0, conjugate\_basis\_functions, output\_basis\_functions, y\_star, 100, F);   
 plot\_dual\_ilc('Scaled Conjugate Basis Application', 'Scaled Conjugate Basis Application', final\_conj\_Trial, y\_star, -1, 4, update\_file\_path);  
end  
  
A\_fixed = eye(num\_ilc\_states);  
B\_fixed = -pinv(output\_basis\_functions) \* P \* conjugate\_basis\_functions;  
poles = eig(A\_fixed + B\_fixed \* F); %A+BF pole formulation  
mags = abs(poles);  
if any(mags > 1)  
 sprintf('Unstable')  
 mags(mags > 1)  
end  
return  
%% Growing Resolution T\_y = Phi^b  
growing\_phi\_R = 1 \* eye(num\_ilc\_inputs);  
Q = 100 \* eye(num\_ilc\_states); %potential for full range  
exploration\_mag = 1;   
T\_y\_scale = 10;  
  
num\_converged = 0;  
num\_controllers = 5;  
%Conjugate episodes holder  
clear 'Episode';  
Episode = [];   
%Setup learning parameters  
%keep same as before  
growing\_Trials = []; %hold all the trials  
F = zeros(num\_ilc\_inputs, num\_output\_basis);  
  
for phi\_num = 1:num\_ilc\_inputs %try every combo possible  
 %Update basis space  
 Episode = generate\_iterative\_conjugate(P, d, y\_star, Episode, Q);  
 phi = Episode(phi\_num).Phi; %get the most recent phis  
 %Learn the controller  
 output\_basis = phi \* T\_y\_scale;  
 input\_basis = phi(:, phi\_num);  
 [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);  
 F(phi\_num, :) = [F\_phi(:, :, end), zeros(1, num\_ilc\_inputs - phi\_num)]; %save the controller in the stack  
 growing\_Trials = [growing\_Trials, temp\_Trial]; %save teh trial info  
end  
Phi = Episode(end).Phi;  
output\_basis = Phi \* T\_y\_scale;  
input\_basis = Phi;  
F\_lqr = discounted\_LQR(eye(num\_basis\_output), -pinv(output\_basis) \* P \* input\_basis, gamma, Q, growing\_phi\_R);  
  
%Show ow they compare (bottom rows should match)  
sprintf('F\_lqr vs Policy when Growing Conjugate Output Basis')  
lqr = F\_lqr([1, 2, 19, 20], [1, 2, 19, 20]);  
policy = F([1, 2, 19, 20], [1, 2, 19, 20]);  
  
%Check the poles  
A\_growing = eye(num\_ilc\_states);  
B\_growing = -pinv(output\_basis) \* P \* input\_basis;  
poles = eig(A\_growing + B\_growing \* F); %A+BF pole formulation  
mags = abs(poles);  
if any(mags > 1)  
 sprintf('Unstable')  
 mags(mags > 1)  
end  
  
%Plot the path  
if plot\_all  
 plot\_dual\_ilc('Growing Basis on IO', 'Growing Basis on IO', growing\_Trials, y\_star, -1, 4, update\_file\_path);  
  
 growing\_Trial = basis\_ilc\_sim(P, d, C\*x0, input\_basis, output\_basis, y\_star, 100, F);   
 plot\_dual\_ilc('Growing Basis on IO', 'Growing Basis on IO', growing\_Trial, y\_star, -1, 4, update\_file\_path);  
end  
  
%% Rolling Resolution T\_y  
rolling\_phi\_R = 1 \* eye(num\_ilc\_inputs);  
Q = 100 \* eye(num\_ilc\_states); %potential for full range  
exploration\_mag = 1;   
T\_y\_scale = 1;  
  
num\_converged = 0;  
num\_controllers = 5;  
%Conjugate episodes holder  
clear 'Episode';  
Episode = [];   
%Setup learning parameters  
%keep same as before  
rolling\_Trials = []; %hold all the trials  
F = zeros(num\_ilc\_inputs, num\_output\_basis);  
  
for phi\_num = 1:num\_ilc\_inputs %try every combo possible  
 %Update basis space  
 Episode = generate\_iterative\_conjugate(P, d, y\_star, Episode, Q);  
 phi = Episode(phi\_num).Phi; %get the most recent phis  
 %Learn the controller  
 input\_basis = phi(:, phi\_num);  
 output\_basis = input\_basis \* T\_y\_scale;  
 [temp\_Trial, F\_phi, ~, ~] = policy\_ilc(P, d, C, D, x0, y\_star, gamma, Q(phi\_num, phi\_num), rolling\_phi\_R(phi\_num, phi\_num), num\_controllers, exploration\_mag, input\_basis, output\_basis, num\_converged);  
 F(phi\_num, phi\_num) = F\_phi(:, :, end); %save the controller in the stack  
 rolling\_Trials = [rolling\_Trials, temp\_Trial]; %save teh trial info  
end  
Phi = Episode(end).Phi;  
output\_basis = Phi \* T\_y\_scale;  
input\_basis = Phi;  
F\_lqr = discounted\_LQR(eye(num\_basis\_output), -pinv(output\_basis) \* P \* input\_basis, gamma, Q, rolling\_phi\_R);  
  
%Show ow they compare (bottom rows should match)  
sprintf('F\_lqr vs Policy when Rolling Conjugate Output Basis')  
lqr = F\_lqr([1, 2, 19, 20], [1, 2, 19, 20]);  
policy = F([1, 2, 19, 20], [1, 2, 19, 20]);  
  
%Check the poles  
A\_growing = eye(num\_ilc\_states);  
B\_growing = -pinv(output\_basis) \* P \* input\_basis;  
poles = eig(A\_growing + B\_growing \* F); %A+BF pole formulation  
mags = abs(poles);  
if any(mags > 1)  
 sprintf('Unstable')  
 mags(mags > 1)  
end  
  
%Plot the path  
if plot\_all  
 plot\_dual\_ilc('Growing Basis on IO', 'Growing Basis on IO', rolling\_Trials, y\_star, -1, 4, update\_file\_path);  
  
 growing\_Trial = basis\_ilc\_sim(P, d, C\*x0, input\_basis, output\_basis, y\_star, 100, F);   
 plot\_dual\_ilc('Growing Basis on IO', 'Growing Basis on IO', growing\_Trial, y\_star, -1, 4, update\_file\_path);  
end

## RL on ILC

%Appllication of Reinforcement Learning to the ILC Problem  
%Noah Dunleavy  
%Honors Thesis under direction of Professor Minh Q. Phan  
%Thayer School of Engineering  
clc; clear;  
addpath('Saved Data', 'Functions');  
setDefaultFigProp();  
plot\_policy = false; %majority of run time goes to figure gen, so this speeds up when false  
plot\_decoupled = false;  
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  
if update\_file\_path ~= -1  
 keyboard %ensure we have to be very concious about ever updating all the images  
end  
  
%% System Creation  
thesis\_system = load('Saved Data\thesis\_system.mat');  
A = thesis\_system.A;  
B = thesis\_system.B;  
C = thesis\_system.C;  
D = thesis\_system.D;  
num\_inputs = thesis\_system.num\_inputs;  
num\_outputs = thesis\_system.num\_outputs;  
num\_states = thesis\_system.num\_states;  
x0 = thesis\_system.x0;  
  
%% Goal Definition  
p = 10;  
  
% loadedShape = load('Saved Data\heart\_p20.mat'); %read in the file  
% y\_star\_x = loadedShape.drawn\_x';  
% y\_star\_y = loadedShape.drawn\_y';  
  
y\_star\_x = cos(2 \* pi \* (0:(p-1)) / p)';  
y\_star\_y = sin(2 \* pi \* (0:(p-1)) / p)';  
goal\_matrix = [y\_star\_x, y\_star\_y]; %stack inputs next to eachother  
y\_star = reshape(goal\_matrix', [], 1); %combine the seperate goals of each output into one vertical vector, alternating as necessary  
  
%% ILC System  
[P, d] = P\_from\_ABCD(A, B, C, D, p, x0); %y(1:p) = P \* u(0:(p-1)) + d  
num\_ilc\_states = p \* num\_outputs; %error bar - one for each output at each time step  
num\_ilc\_inputs = p \* num\_inputs; %u bar - each time step gets an input  
  
%ILC State-space so that e(j+1) = A \* e(j) + B \* del\_u\_j1 and y(j) = e(j)  
A\_ilc = eye(num\_ilc\_states);  
B\_ilc = -P;  
C\_ilc = eye(num\_ilc\_states);  
D\_ilc = 0;  
if (~is\_controllable(A\_ilc, B\_ilc))  
 fprintf('The ILC System is not Controllable!')  
 return  
end  
  
%% Policy Iteration RL Parameters, R = 1  
%Learning Weights  
Q = 100 \* eye(num\_ilc\_states); %cost of each error  
R = 1 \* eye(num\_ilc\_inputs); %penalize inputs, or more accurately change in input  
 %Setting this too small causes controller to fail, but logically 0  
 %should be safe  
gamma = 0.8;  
policy\_exploration\_mag = 1;  
F\_ilc\_lqr = discounted\_LQR(A\_ilc, B\_ilc, gamma, Q, R); %perfect nowledge controller  
  
  
  
%Iteration Counts  
Pj\_dim = (num\_ilc\_inputs + num\_ilc\_states); %the square diension of the Pj matrix  
num\_controllers = 5; %number of controllers to try to create  
num\_collections\_per\_controller = Pj\_dim^2; %number of datasets needed per controller to update  
num\_converged = 1000; %how many trials to then go through once we are done 'learning' (so we arent noisey)  
  
%To be able to set R smaller, increase exploration magnitude ad the number  
%of collections per controller  
  
total\_trial\_count = num\_controllers \* num\_collections\_per\_controller + num\_converged;  
  
%Preallocate the space and define structure  
ILC\_Trial(total\_trial\_count).output = [];   
ILC\_Trial(total\_trial\_count).input = [];  
ILC\_Trial(total\_trial\_count).output\_error = [];  
ILC\_Trial(total\_trial\_count).del\_u = [];  
  
%% Policy Iteration Learning Process, R = 1  
rng('default'); rng(10);%ensure same randomization each time  
F\_ilc = zeros(num\_ilc\_inputs, num\_ilc\_states, num\_controllers + 1); %start with no controller  
  
%Prepopulate the first trial  
trial\_num = 1;  
ILC\_Trial(trial\_num).input = zeros(num\_ilc\_inputs, 1); %start with open-loop / no input  
ILC\_Trial(trial\_num).output = [C\*x0; d]; %open loop response is IC and then d term  
ILC\_Trial(trial\_num).output\_error = y\_star - d; %relevant error  
  
trial\_num = 2; %start at second trial now  
for iteration = 1:num\_controllers   
 Uk\_stack = zeros(num\_collections\_per\_controller, 1);  
 Xk\_stack = zeros(num\_collections\_per\_controller, (Pj\_dim)^2);  
  
 %Simulate the necessary trials  
 for trial = 1:num\_collections\_per\_controller %number of trials to collect before updatin controller  
 %ILC / Real Controller Process  
 %Inputs  
 exploration\_term = rand\_range(num\_ilc\_inputs, 1, -policy\_exploration\_mag, policy\_exploration\_mag); %jiggle to learn  
 ILC\_Trial(trial\_num).del\_u = F\_ilc(:, :, iteration) \* ILC\_Trial(trial\_num - 1).output\_error + exploration\_term;  
 ILC\_Trial(trial\_num).input = ILC\_Trial(trial\_num - 1).input + ILC\_Trial(trial\_num).del\_u;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial(trial\_num).input + d; %y(1) -> y(p)  
 ILC\_Trial(trial\_num).output = [C\*x0 + D\*ILC\_Trial(trial\_num).input(1:num\_inputs); relevant\_output]; %total output y(0) -> y(p) for completeness  
 %Calculate Error  
 ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
 %error\_next\_law = ILC\_Trial(trial\_num - 1).output\_error - P \* ILC\_Trial(trial\_num).del\_u; %verify that this matches the produced error, that is:  
 %e(j+1) = e(j) - P\*del\_u(j+1) when del\_u(j+1) = L \* e(j) = u(j) - u(j-1)  
  
 %RL Translation  
 state = ILC\_Trial(trial\_num - 1).output\_error; %analogous state, x(k) -> e\_(j-1)  
 input = ILC\_Trial(trial\_num).del\_u;%analogous input, u(k) -> del\_u\_(j) = L \* e\_(j-1)  
 next\_state = ILC\_Trial(trial\_num).output\_error; %x(k+1) = e(j)  
 next\_input = F\_ilc(:, :, iteration) \* next\_state; %no exploration term here  
  
 xu\_stack = [state; input];  
 xu\_next\_stack = [next\_state; next\_input];  
  
 Xk\_stack(trial, :) = kron(xu\_stack', xu\_stack') - gamma \* kron(xu\_next\_stack', xu\_next\_stack');  
 Uk\_stack(trial, :) = input' \* R \* input + state' \* Q \* state;  
  
 trial\_num = trial\_num + 1;  
 end  
  
 %Calculate P and new controller  
 % [U, S, V] = svd(Xk\_stack);  
 % rank(S)  
 PjS = pinv(Xk\_stack) \* Uk\_stack;  
 Pj = reshape(PjS, Pj\_dim, Pj\_dim);  
 Pj = 0.5 \* (Pj + Pj'); %to impose symmetry (significantly reduces error)  
 Pjuu = Pj((num\_ilc\_states+1):end, (num\_ilc\_states+1):end);  
 PjxuT = Pj((num\_ilc\_states+1):end, 1:num\_ilc\_states);  
 new\_F = -pinv(Pjuu) \* PjxuT;  
 F\_ilc(:, :, iteration + 1) = new\_F;  
end  
  
for ndx = 1:num\_converged  
 %Inputs  
 ILC\_Trial(trial\_num).del\_u = F\_ilc(:, :, end) \* ILC\_Trial(trial\_num - 1).output\_error;  
 ILC\_Trial(trial\_num).input = ILC\_Trial(trial\_num - 1).input + ILC\_Trial(trial\_num).del\_u;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial(trial\_num).input + d; %y(1) -> y(p)  
 ILC\_Trial(trial\_num).output = [C\*x0 + D\*ILC\_Trial(trial\_num).input(1:num\_inputs); relevant\_output]; %total output y(0) -> y(p) for completeness  
 %Calculate Error  
 ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
 trial\_num = trial\_num + 1;  
end  
controller\_error = norm(F\_ilc(:, :, end) - F\_ilc\_lqr)/numel(F\_ilc\_lqr)  
  
%% Visualize Learning  
if plot\_policy  
 to\_plot = 4;  
 plot\_dual\_ilc('Policy Iteration on ILC', 'Policy Iteration for ILC', ILC\_Trial, y\_star, -1, to\_plot, update\_file\_path);  
 plot\_controller\_history('Policy Iteration ILC Controller', 'Policy Iteration ILC Controller Weights', F\_ilc, 4, 5, update\_file\_path); %we cannot plot \*all\* the IOs because there are so many, pick a few  
end  
  
%% Input Decoupled Learning  
%Iteration Counts  
decoupled\_exploration\_mag = policy\_exploration\_mag;  
Pj\_dim = (1 + num\_ilc\_states);   
num\_controllers = 5;   
num\_collections\_per\_controller = Pj\_dim^2;   
num\_converged\_decoupled = num\_collections\_per\_controller;   
  
total\_trial\_count\_decoupled = num\_ilc\_inputs \* num\_controllers \* num\_collections\_per\_controller + num\_converged\_decoupled; %now need a \* num inputs  
  
%Preallocate the space and define structure  
ILC\_Trial\_decoupled(total\_trial\_count\_decoupled).output = [];   
ILC\_Trial\_decoupled(total\_trial\_count\_decoupled).input = [];  
ILC\_Trial\_decoupled(total\_trial\_count\_decoupled).output\_error = [];  
ILC\_Trial\_decoupled(total\_trial\_count\_decoupled).del\_u = [];  
  
%% Input Decoupled Learning Process  
rng('default'); rng(10);%ensure same randomization each time  
F\_ilc\_decoupled = zeros(num\_ilc\_inputs, num\_ilc\_states, 1);   
  
%Prepopulate the first trial  
trial\_num = 1;  
ILC\_Trial\_decoupled(trial\_num).input = zeros(num\_ilc\_inputs, 1); %start with open-loop / no input  
ILC\_Trial\_decoupled(trial\_num).output = [C\*x0; d]; %open loop response is IC and then d term  
ILC\_Trial\_decoupled(trial\_num).output\_error = y\_star - d; %relevant error  
  
trial\_num = 2; %start at second trial now  
for iteration = 1:(num\_controllers)   
 for input\_num = 1:num\_ilc\_inputs  
 Uk\_stack = zeros(num\_collections\_per\_controller, 1);  
 Xk\_stack = zeros(num\_collections\_per\_controller, (Pj\_dim)^2);  
 F\_i = F\_ilc\_decoupled(input\_num, :, end);  
 %Simulate the necessary trials  
 for trial = 1:num\_collections\_per\_controller %number of trials to collect before updatin controller  
 %ILC / Real Controller Process  
 %Inputs  
 exploration\_term = rand\_range(1, 1, -decoupled\_exploration\_mag, decoupled\_exploration\_mag); %jiggle to learn  
 ILC\_Trial\_decoupled(trial\_num).del\_u = F\_ilc\_decoupled(:, :, iteration) \* ILC\_Trial\_decoupled(trial\_num - 1).output\_error;  
  
 ILC\_Trial\_decoupled(trial\_num).del\_u(input\_num) = ILC\_Trial\_decoupled(trial\_num).del\_u(input\_num) + exploration\_term; %differnece between PI - only learn on one input  
  
 ILC\_Trial\_decoupled(trial\_num).input = ILC\_Trial\_decoupled(trial\_num - 1).input + ILC\_Trial\_decoupled(trial\_num).del\_u;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial\_decoupled(trial\_num).input + d; %y(1) -> y(p)  
 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  
 %Calculate Error  
 ILC\_Trial\_decoupled(trial\_num).output\_error = y\_star - relevant\_output;  
  
  
 %RL Translation  
 state = ILC\_Trial\_decoupled(trial\_num - 1).output\_error; %analogous state, x(k) -> e\_(j-1)  
 full\_input = ILC\_Trial\_decoupled(trial\_num).del\_u; %Uk vs Xk use different inputs  
 input = full\_input(input\_num);  
 next\_state = ILC\_Trial\_decoupled(trial\_num).output\_error; %x(k+1) = e(j)  
 next\_input = F\_i \* next\_state; %no exploration term here  
  
 xu\_stack = [state; input];  
 xu\_next\_stack = [next\_state; next\_input];  
  
 Xk\_stack(trial, :) = kron(xu\_stack', xu\_stack') - gamma \* kron(xu\_next\_stack', xu\_next\_stack');  
 Uk\_stack(trial, :) = full\_input' \* R \* full\_input + state' \* Q \* state;  
  
 trial\_num = trial\_num + 1;  
 end  
  
 %Calculate P and new controller  
 PjS = pinv(Xk\_stack) \* Uk\_stack;  
 Pj = reshape(PjS, Pj\_dim, Pj\_dim);  
 Pj = 0.5 \* (Pj + Pj'); %to impose symmetry (significantly reduces error)  
 Pjuu = Pj((num\_ilc\_states+1):end, (num\_ilc\_states+1):end);  
 PjxuT = Pj((num\_ilc\_states+1):end, 1:num\_ilc\_states);  
 new\_F\_i = -pinv(Pjuu) \* PjxuT;  
 F\_ilc\_decoupled(:, :, end + 1) = F\_ilc\_decoupled(:, :, end);  
 F\_ilc\_decoupled(input\_num, :, end) = new\_F\_i;  
 end  
end  
  
for ndx = 1:num\_converged\_decoupled  
 %Inputs  
 ILC\_Trial\_decoupled(trial\_num).del\_u = F\_ilc\_decoupled(:, :, end) \* ILC\_Trial\_decoupled(trial\_num - 1).output\_error;  
 ILC\_Trial\_decoupled(trial\_num).input = ILC\_Trial\_decoupled(trial\_num - 1).input + ILC\_Trial\_decoupled(trial\_num).del\_u;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial\_decoupled(trial\_num).input + d; %y(1) -> y(p)  
 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  
 %Calculate Error  
 ILC\_Trial\_decoupled(trial\_num).output\_error = y\_star - relevant\_output;  
 trial\_num = trial\_num + 1;  
end  
controller\_error = norm(F\_ilc\_decoupled(:, :, end) - F\_ilc\_lqr)/numel(F\_ilc\_lqr)  
  
%% Visualize Learning  
if plot\_decoupled  
 to\_plot = 4;  
 plot\_dual\_ilc('Input Decoupling on ILC', 'Input Decoupling for ILC', ILC\_Trial\_decoupled, y\_star, -1, to\_plot, update\_file\_path);  
 plot\_controller\_history('Input Decoupling ILC Controller', 'Input Decoupling ILC Controller Weights', F\_ilc\_decoupled, 4, 5, update\_file\_path, true);  
end  
  
%% Policy Iteration RL Parameters, R = 1e-6  
%Learning Weights  
Q = 100 \* eye(num\_ilc\_states); %cost of each error  
small\_R = 1e-6 \* eye(num\_ilc\_inputs); %penalize inputs, or more accurately change in input  
 %Setting this too small causes controller to fail, but logically 0  
 %should be safe  
gamma = 0.8;  
small\_R\_policy\_exploration\_mag = 1000;  
F\_ilc\_lqr\_small\_R = discounted\_LQR(A\_ilc, B\_ilc, gamma, Q, small\_R); %perfect nowledge controller  
  
%Iteration Counts  
Pj\_dim = (num\_ilc\_inputs + num\_ilc\_states); %the square diension of the Pj matrix  
num\_controllers = 5; %number of controllers to try to create  
num\_collections\_per\_controller = Pj\_dim^2; %number of datasets needed per controller to update  
num\_converged = 100; %how many trials to then go through once we are done 'learning' (so we arent noisey)  
  
%To be able to set R smaller, increase exploration magnitude ad the number  
%of collections per controller  
  
total\_trial\_count = num\_controllers \* num\_collections\_per\_controller + num\_converged;  
  
%Preallocate the space and define structure  
ILC\_Trial\_small\_R(total\_trial\_count).output = [];   
ILC\_Trial\_small\_R(total\_trial\_count).input = [];  
ILC\_Trial\_small\_R(total\_trial\_count).output\_error = [];  
ILC\_Trial\_small\_R(total\_trial\_count).del\_u = [];  
  
%% Policy Iteration Learning Process, R = 1e-6  
rng('default'); rng(10);%ensure same randomization each time  
F\_ilc\_policy\_small\_R = zeros(num\_ilc\_inputs, num\_ilc\_states, num\_controllers + 1); %start with no controller  
  
%Prepopulate the first trial  
trial\_num = 1;  
ILC\_Trial\_small\_R(trial\_num).input = zeros(num\_ilc\_inputs, 1); %start with open-loop / no input  
ILC\_Trial\_small\_R(trial\_num).output = [C\*x0; d]; %open loop response is IC and then d term  
ILC\_Trial\_small\_R(trial\_num).output\_error = y\_star - d; %relevant error  
  
trial\_num = 2; %start at second trial now  
for iteration = 1:num\_controllers   
 Uk\_stack = zeros(num\_collections\_per\_controller, 1);  
 Xk\_stack = zeros(num\_collections\_per\_controller, (Pj\_dim)^2);  
  
 %Simulate the necessary trials  
 for trial = 1:num\_collections\_per\_controller %number of trials to collect before updatin controller  
 %ILC / Real Controller Process  
 %Inputs  
 exploration\_term = rand\_range(num\_ilc\_inputs, 1, -small\_R\_policy\_exploration\_mag, small\_R\_policy\_exploration\_mag); %jiggle to learn  
 ILC\_Trial\_small\_R(trial\_num).del\_u = F\_ilc\_policy\_small\_R(:, :, iteration) \* ILC\_Trial\_small\_R(trial\_num - 1).output\_error + exploration\_term;  
 ILC\_Trial\_small\_R(trial\_num).input = ILC\_Trial\_small\_R(trial\_num - 1).input + ILC\_Trial\_small\_R(trial\_num).del\_u;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial\_small\_R(trial\_num).input + d; %y(1) -> y(p)  
 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  
 %Calculate Error  
 ILC\_Trial\_small\_R(trial\_num).output\_error = y\_star - relevant\_output;  
 %error\_next\_law = ILC\_Trial(trial\_num - 1).output\_error - P \* ILC\_Trial(trial\_num).del\_u; %verify that this matches the produced error, that is:  
 %e(j+1) = e(j) - P\*del\_u(j+1) when del\_u(j+1) = L \* e(j) = u(j) - u(j-1)  
  
 %RL Translation  
 state = ILC\_Trial\_small\_R(trial\_num - 1).output\_error; %analogous state, x(k) -> e\_(j-1)  
 input = ILC\_Trial\_small\_R(trial\_num).del\_u;%analogous input, u(k) -> del\_u\_(j) = L \* e\_(j-1)  
 next\_state = ILC\_Trial\_small\_R(trial\_num).output\_error; %x(k+1) = e(j)  
 next\_input = F\_ilc\_policy\_small\_R(:, :, iteration) \* next\_state; %no exploration term here  
  
 xu\_stack = [state; input];  
 xu\_next\_stack = [next\_state; next\_input];  
  
 Xk\_stack(trial, :) = kron(xu\_stack', xu\_stack') - gamma \* kron(xu\_next\_stack', xu\_next\_stack');  
 Uk\_stack(trial, :) = input' \* small\_R \* input + state' \* Q \* state;  
  
 trial\_num = trial\_num + 1;  
 end  
  
 %Calculate P and new controller  
 % [U, S, V] = svd(Xk\_stack);  
 % rank(S)  
 PjS = pinv(Xk\_stack) \* Uk\_stack;  
 Pj = reshape(PjS, Pj\_dim, Pj\_dim);  
 Pj = 0.5 \* (Pj + Pj'); %to impose symmetry (significantly reduces error)  
 Pjuu = Pj((num\_ilc\_states+1):end, (num\_ilc\_states+1):end);  
 PjxuT = Pj((num\_ilc\_states+1):end, 1:num\_ilc\_states);  
 new\_F = -pinv(Pjuu) \* PjxuT;  
 F\_ilc\_policy\_small\_R(:, :, iteration + 1) = new\_F;  
end  
  
for ndx = 1:num\_converged  
 %Inputs  
 ILC\_Trial\_small\_R(trial\_num).del\_u = F\_ilc\_policy\_small\_R(:, :, end) \* ILC\_Trial\_small\_R(trial\_num - 1).output\_error;  
 ILC\_Trial\_small\_R(trial\_num).input = ILC\_Trial\_small\_R(trial\_num - 1).input + ILC\_Trial\_small\_R(trial\_num).del\_u;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial\_small\_R(trial\_num).input + d; %y(1) -> y(p)  
 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  
 %Calculate Error  
 ILC\_Trial\_small\_R(trial\_num).output\_error = y\_star - relevant\_output;  
 trial\_num = trial\_num + 1;  
end  
controller\_error = norm(F\_ilc\_policy\_small\_R(:, :, end) - F\_ilc\_lqr\_small\_R)/numel(F\_ilc\_lqr\_small\_R)  
  
%% Visualize Learning  
if plot\_policy  
 to\_plot = 4;  
 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);  
 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  
end

# Matlab Functions

## Controllability Check

function [controllable] = is\_controllable(A, B, precision)  
%Check whether or not a matrix specificed by A and B matricies is  
%controllable  
%Inputs:  
 %A: matrix - state dynamics  
 %B: matrix - input dynamics  
 %precision: scalar - set how precise matlab is with its rounding  
%Outputs:  
 %controllable: bool - whether or not the system is controllable  
  
if exist('precision', 'var') %ability to set precision  
 A = vpa(A, precision);   
 B = vpa(B, precision);  
end  
  
controllability\_matrix = ctrb(A, B);  
controllable = (rank(controllability\_matrix) == height(A));  
  
end

## Decoupled Learning for Iterative Learning Control

function [ILC\_Trial, F\_decoupled, controller\_error, F\_lqr] = decoupled\_ilc(P, d, C, D, x0, y\_star, gamma, Q, R, num\_controllers, exploration\_mag, input\_basis\_functions, output\_basis\_functions, num\_converged)  
%Perform the RL input decoupled learning on an ILC system since done so much in  
%thesis  
%Inputs:  
 %P: matrix - inputs u(0->(p-1)) to outputs y(1->p)  
 %d: vector - noise/initial conditions matrix  
 %C: matrix - state to output descriptor  
 %D: matrix - input to output descriptor  
 %x0: vector - initial state  
 %y\_star: vector - goal output  
 %gamma: scalar - discount factor  
 %Q: matrix - cost of states (errors)  
 %R: matrix - cost of inputs (change in inputs)  
 %num\_controllers: scalar - number of controllers to learn  
 %exploration\_mag: scalar - range around 0 to explore (defaults to 1)  
 %input\_basis\_functions: matrix - basis functions on the inputs (defaults to identity)  
 %output\_basis\_functions: matrix - basis functions on the outputs (defaults to identity)   
 %num\_converged: scalar - number of trials to simulate out without exploration (defaults to 0)  
%Outputs:  
 %ILC\_Trial: structure with indexed by trial number, contains  
 %inputs  
 %betas (input basis weights)  
 %del\_betas (change in input betas)  
 %outputs  
 %alphas (output basis weights)  
 %output error (y\* - y)  
 %alpha error  
 %F\_decoupled: matrix - controller learning history  
 %controller\_error: scalar - normalized error from LQR  
 %F\_lqr: matrix - lqr controller to compare to  
  
%Required Parameters to System Info  
num\_inputs = width(D);  
  
%Default paramters  
if ~exist('num\_converged', 'var')  
 num\_converged = 0; %default to no converged trials  
end  
if ~exist('exploration\_mag', 'var')  
 exploration\_mag = 1;  
end  
if ~exist('input\_basis\_functions', 'var')  
 input\_basis\_functions = eye(width(P));  
end  
if ~exist('output\_basis\_functions', 'var')  
 output\_basis\_functions = eye(height(P));  
 output\_basis\_functions\_pinv = output\_basis\_functions;%save on compute time  
else  
 output\_basis\_functions\_pinv = pinv(output\_basis\_functions);  
end  
num\_ilc\_states = width(output\_basis\_functions);  
num\_ilc\_inputs = width(input\_basis\_functions);  
  
alpha\_star = output\_basis\_functions\_pinv \* y\_star;  
  
%Calculate optimal controller  
F\_lqr = discounted\_LQR(eye(num\_ilc\_states), -output\_basis\_functions\_pinv \* P \* input\_basis\_functions, gamma, Q, R);  
  
  
%Iteration Counts  
Pj\_dim = num\_ilc\_states + 1;  
num\_collections\_per\_controller = Pj\_dim^2;   
total\_trial\_count = num\_ilc\_inputs \* num\_controllers \* num\_collections\_per\_controller + num\_converged;  
  
%Preallocate structure  
ILC\_Trial(total\_trial\_count).input = []; %input  
ILC\_Trial(total\_trial\_count).betas = []; %basis representation of input  
ILC\_Trial(total\_trial\_count).del\_beta = [];  
ILC\_Trial(total\_trial\_count).output = []; %output  
ILC\_Trial(total\_trial\_count).alphas = []; %basis representation of output  
ILC\_Trial(total\_trial\_count).output\_error = []; %output error  
ILC\_Trial(total\_trial\_count).alpha\_error = []; %alpha error  
  
F\_decoupled = zeros(num\_ilc\_inputs, num\_ilc\_states, num\_controllers + 1); %start with no controller  
  
%Prepopulate the first trial  
trial\_num = 1;  
ILC\_Trial\_decoupled(trial\_num).betas = zeros(num\_ilc\_inputs, 1); %start with no basis guessed  
ILC\_Trial\_decoupled(trial\_num).input = input\_basis\_functions \* ILC\_Trial\_decoupled(trial\_num).betas;   
  
ILC\_Trial\_decoupled(trial\_num).output = [C\*x0; d]; %open loop response is IC and then d term  
ILC\_Trial\_decoupled(trial\_num).alphas = output\_basis\_functions\_pinv \* d;  
ILC\_Trial\_decoupled(trial\_num).output\_error = y\_star - d; %relevant error  
ILC\_Trial\_decoupled(trial\_num).alpha\_error = alpha\_star - ILC\_Trial\_decoupled(trial\_num).alphas;   
  
trial\_num = 2; %start at second trial now  
for iteration = 1:num\_controllers  
 for input\_num = 1:num\_ilc\_inputs  
 Uk\_stack = zeros(num\_collections\_per\_controller, 1);  
 Xk\_stack = zeros(num\_collections\_per\_controller, (Pj\_dim)^2);  
 F\_i = F\_decoupled(input\_num, :, end);  
 %Simulate the necessary trials  
 for trial = 1:num\_collections\_per\_controller %number of trials to collect before updatin controller  
 %ILC / Basis Controller Process  
 %Beta Coeffecients  
 exploration\_term = rand\_range(1, 1, -exploration\_mag, exploration\_mag); %jiggle to learn  
  
 ILC\_Trial\_decoupled(trial\_num).del\_beta = F\_decoupled(:, :, end) \* ILC\_Trial\_decoupled(trial\_num - 1).alpha\_error;  
 ILC\_Trial\_decoupled(trial\_num).del\_beta(input\_num) = ILC\_Trial\_decoupled(trial\_num).del\_beta(input\_num) + exploration\_term;  
  
 ILC\_Trial\_decoupled(trial\_num).betas = ILC\_Trial\_decoupled(trial\_num - 1).betas + ILC\_Trial\_decoupled(trial\_num).del\_beta;  
 %Beta to Inputs  
 ILC\_Trial\_decoupled(trial\_num).input = input\_basis\_functions \* ILC\_Trial\_decoupled(trial\_num).betas;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial\_decoupled(trial\_num).input + d; %y(1) -> y(p)  
 ILC\_Trial\_decoupled(trial\_num).alphas = output\_basis\_functions\_pinv \* relevant\_output;  
 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  
 %Calculate Error  
 ILC\_Trial\_decoupled(trial\_num).output\_error = y\_star - relevant\_output;  
 ILC\_Trial\_decoupled(trial\_num).alpha\_error = alpha\_star - ILC\_Trial\_decoupled(trial\_num).alphas;  
  
 %RL Translation  
 state = ILC\_Trial\_decoupled(trial\_num - 1).alpha\_error; %analogous state  
 full\_input = ILC\_Trial\_decoupled(trial\_num).del\_beta; %analogous input  
 input = full\_input(input\_num);  
 next\_state = ILC\_Trial\_decoupled(trial\_num).alpha\_error; %x(k+1) = e\_alpha(j)  
 next\_input = F\_i \* next\_state; %no exploration term here  
  
 xu\_stack = [state; input];  
 xu\_next\_stack = [next\_state; next\_input];  
  
 Xk\_stack(trial, :) = kron(xu\_stack', xu\_stack') - gamma \* kron(xu\_next\_stack', xu\_next\_stack');  
 Uk\_stack(trial, :) = full\_input' \* R \* full\_input + state' \* Q \* state;  
  
 trial\_num = trial\_num + 1;  
 end  
  
 %Calculate P and new controller  
 PjS = pinv(Xk\_stack) \* Uk\_stack;  
 Pj = reshape(PjS, Pj\_dim, Pj\_dim);  
 Pj = 0.5 \* (Pj + Pj'); %to impose symmetry (significantly reduces error)  
 Pjuu = Pj((num\_ilc\_states+1):end, (num\_ilc\_states+1):end);  
 PjxuT = Pj((num\_ilc\_states+1):end, 1:num\_ilc\_states);  
 new\_F\_i = -pinv(Pjuu) \* PjxuT;  
 F\_decoupled(:, :, end + 1) = F\_decoupled(:, :, end);  
 F\_decoupled(input\_num, :, end) = new\_F\_i;  
 end  
end  
  
for ndx = 1:num\_converged  
 %ILC / Basis Controller Process  
 %Beta Coeffecients  
 ILC\_Trial\_decoupled(trial\_num).del\_beta = F\_decoupled(:, :, iteration) \* ILC\_Trial\_decoupled(trial\_num - 1).alpha\_error;  
 ILC\_Trial\_decoupled(trial\_num).betas = ILC\_Trial\_decoupled(trial\_num - 1).betas + ILC\_Trial\_decoupled(trial\_num).del\_beta;  
 %Beta to Inputs  
 ILC\_Trial\_decoupled(trial\_num).input = input\_basis\_functions \* ILC\_Trial\_decoupled(trial\_num).betas;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial\_decoupled(trial\_num).input + d; %y(1) -> y(p)  
 ILC\_Trial\_decoupled(trial\_num).alphas = output\_basis\_functions\_pinv \* relevant\_output;  
 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  
 %Calculate Error  
 ILC\_Trial\_decoupled(trial\_num).output\_error = y\_star - relevant\_output;  
 ILC\_Trial\_decoupled(trial\_num).alpha\_error = alpha\_star - ILC\_Trial\_decoupled(trial\_num).alphas;  
  
 trial\_num = trial\_num + 1;  
end  
controller\_error = norm(F\_decoupled(:, :, end) - F\_lqr)/numel(F\_lqr);  
  
end

## Default Figure Properties

function [] = setDefaultFigProp()  
%Text size scaling set in save\_figure  
%Set some global parameters for general figure properties  
  
%Set global properties for plotting for the executed script  
set(groot, 'DefaultFigureColor', 'w'); %Default background color for figures  
  
set(groot, 'DefaultAxesFontName', 'Arial'); %Font  
set(groot, 'DefaultAxesLineWidth', 1.8);   
set(groot, 'DefaultLegendBackgroundAlpha', 0.5);  
  
setappdata(groot, 'DefaultSubtitleFontSize', 10); %just a temp small subtitle font (save\_figure should overwrite)  
  
%Default Line  
set(groot, 'DefaultLineLineWidth', 4);   
set(groot, 'DefaultLineMarkerSize', 16);   
  
%Default stair properties (inherit most from line)  
set(groot, 'DefaultStairLineWidth', 4);   
set(groot, 'DefaultStairLineStyle', '-');   
  
%Default scatter properties  
%set(groot, 'DefaultScatterMarkerSize', 50);  
%set(groot, 'DefaultScatterFilled', '1');  
set(groot, 'DefaultScatterMarker', 'o');   
set(groot, 'DefaultScatterSizeData', 200); % Set default marker size  
set(groot, 'DefaultScatterLineWidth', 4);  
  
  
end

## Discounted LQR Solution

function [F] = discounted\_LQR(A, B, discount\_factor, Q, R, verbose)  
%From system specifications, determine the optimal feedback controller  
%(Linear quadratic regulator). Of form u(k) = F\*x(k) optimally  
%Inputs:  
 %A: matrix - system dynamics  
 %B: matrix - input dynamics  
 %discount\_factor: scalar - discount factor  
 %Q: matrix - state costs  
 %R: matrix - input costs  
 %verbose: bool - system dynamics  
%Outputs:  
 %F: matrix - found controller  
  
%Check if the verbose provided  
if nargin < 6  
 verbose = false; %Default false  
end  
  
%Solve for P - the solution to Algebraic Riccati Equation (Discrete - DARE):  
R\_gamma = R/discount\_factor;  
A\_gamma = sqrt(discount\_factor)\*A;  
[P, ~, ~] = idare(A\_gamma, B, Q, R\_gamma); %solve the riccati equation  
  
%Verify it is a solution (text output for user)  
if verbose  
 test\_P = A\_gamma'\*P\*A\_gamma - A\_gamma'\*P\*B / ((R\_gamma + B'\*P\*B)) \* B'\*P\*A\_gamma + Q; %verify P is a solution  
 sprintf('Solved Ps suitability to solve the Riccati equation is %g', norm(test\_P - P)) %names the answer this way - want this small  
end  
  
%Calculate F  
F = (-1/sqrt(discount\_factor)) \* inv((transpose(B) \* P \* B + R\_gamma)) \* transpose(B)\*P\*A\_gamma; %LQR solution  
end

## Draw a Goal

function [x, y] = draw\_to\_XY(resolution)  
 %Written by ChatGPT with prompt: 'I would like a matlab function that   
 %opens up a figure that I can draw on with my mouse, and then it returns   
 %two arrays of the x positions and y positiions of the thing I drew. I   
 %would like to set the resoltuon (aka number of xy pairs) in the   
 %function call'  
  
 % draw\_to\_XY - Opens a figure to draw with the mouse and returns x, y coordinates.  
 %  
 % Syntax: [x, y] = draw\_with\_mouse(resolution)  
 %  
 % Inputs:  
 % resolution - Number of points to return in the output arrays.  
 %  
 % Outputs:  
 % x - Array of x positions of the drawn curve.  
 % y - Array of y positions of the drawn curve.  
  
 if nargin < 1  
 resolution = 100; % Default resolution if not specified  
 end  
  
 % Create a new figure  
 figure;  
 hold on;  
 grid on  
 axis([0 1 0 1]); % Set axis limits (can be adjusted as needed)  
 title('Draw with your mouse (click and drag). Press Enter when done.');  
 set(gca, 'Position', [0.1, 0.1, 0.8, 0.8]); % Adjust axis position for aesthetics  
 set(gcf, 'WindowButtonMotionFcn', @mouse\_move\_callback); % Enable mouse motion tracking  
 set(gcf, 'WindowButtonDownFcn', @mouse\_down\_callback); % Start drawing on mouse press  
 set(gcf, 'WindowButtonUpFcn', @mouse\_up\_callback); % Stop drawing on mouse release  
 pan off; zoom off; % Disable default interactions  
   
 % Variables to store points  
 drawing = false; % Indicates whether the mouse is pressed  
 points\_x = [];  
 points\_y = [];  
  
 % Callback to track mouse movements while pressed  
 function mouse\_move\_callback(~, ~)  
 if drawing  
 current\_point = get(gca, 'CurrentPoint');  
 points\_x(end + 1) = current\_point(1, 1);  
 points\_y(end + 1) = current\_point(1, 2);  
 plot(points\_x, points\_y, 'b-');  
 drawnow;  
 end  
 end  
  
 % Callback to start drawing  
 function mouse\_down\_callback(~, ~)  
 drawing = true;  
 end  
  
 % Callback to stop drawing  
 function mouse\_up\_callback(~, ~)  
 drawing = false;  
 end  
  
 % Wait for the user to press Enter  
 pause;  
 close(gcf); % Close the figure  
  
 % Interpolate to match the desired resolution  
 t\_raw = linspace(0, 1, length(points\_x));  
 t\_interp = linspace(0, 1, resolution);  
  
 x = interp1(t\_raw, points\_x, t\_interp, 'linear');  
 y = interp1(t\_raw, points\_y, t\_interp, 'linear');  
end

## Figure Saving

function [file\_path] = save\_figure(path, fig\_list, name\_overwrite, shaped)  
%Save figures to specified path following naming convention  
%Inputs:  
 %path: string - where to save the file to,   
 %fig\_list: list - figure handles  
 %name\_overwrite: string - if you wish to overwrite the naming convention  
 %shaped: bool - whether or not it is shaped (should be square)  
%Outputs:  
 %file\_path: string - file save location  
  
%Set save settings  
title\_size = 45;  
subtitle\_size = 30;  
axes\_size = 40;  
tick\_size = 35;  
legend\_size = 30;  
  
%Default condition  
if ~exist('name\_overwrite', 'var')  
 name\_overwrite = -1;  
end  
if ~exist('shaped', 'var')  
 shaped = false;  
end  
  
if path == -1 %condition for if we are not updating the figures  
 return  
end  
  
%Dimension params  
aspect\_ratio = 16/12;  
figure\_width = 16;  
figure\_height = figure\_width / aspect\_ratio;  
if shaped  
 figure\_width = figure\_height;  
end  
  
for fig = fig\_list  
 %Determine what name to give the file  
 if name\_overwrite == -1  
 file\_name = fig.Name;  
 else  
 file\_name = name\_overwrite;  
 end  
  
 %Set figure sizes  
 fig.Units = 'inches';  
 fig.Position = [0, 0, figure\_width, figure\_height];  
  
 % Ensure the axes is square  
 ax = findobj(fig, 'Type', 'axes');  
 for a = ax'  
 %Set font sizes  
 a.FontSize = tick\_size; %tick size - do this one first because otherwise it overwrites others  
 title(a, a.Title.String, 'FontSize', title\_size, 'FontWeight', 'bold');  
 xlabel(a, a.XLabel.String, 'FontSize', axes\_size);  
 ylabel(a, a.YLabel.String, 'FontSize', axes\_size);  
  
 %adjust subtitles  
 if isprop(a, 'Subtitle') && ~isempty(a.Subtitle.String)  
 subtitle(a, a.Subtitle.String, 'FontSize', subtitle\_size, 'FontWeight', 'normal');  
 end  
  
 %adjust legend  
 lgd = findobj(fig, 'Type', 'legend');  
 for l = lgd'  
 l.FontSize = legend\_size;  
 l.Location = 'northeast';  
 end  
  
 set(a, 'PlotBoxAspectRatio', [figure\_width, figure\_height, 1]); %ensure plot matched to our desired scales  
 if shaped  
 set(a, 'DataAspectRatio', [1, 1, 1]); %force square  
 axis(a, 'equal'); %dont distort scaling  
 else  
 set(a, 'DataAspectRatioMode', 'auto');  
 axis(a, 'tight'); % Ensure scaling is not distorted  
 end  
 %Fix offset caused by labels  
 insets = get(a, 'TightInset'); %space needed for labels  
 left\_offset = insets(1); %padding due to ylabel  
 bottom\_offset = insets(2); %padding due to xlabel  
 right\_offset = insets(3); %padding due to xlabel  
 top\_offset = insets(4); %padding due to title  
  
 % Adjust outer position to fit tightly around the plot content  
 % Ensure no unnecessary margins are included  
 a.OuterPosition = [left\_offset/figure\_width, bottom\_offset/figure\_height, (figure\_width - left\_offset - right\_offset)/figure\_width, (figure\_height - top\_offset - bottom\_offset)/figure\_height];  
 end  
  
 %setup space such that image makes room for outer elements (title,  
 %axes, etc)  
 fig.PaperUnits = 'inches';  
 fig.PaperPosition = [0, 0, figure\_width, figure\_height]; %make saved figure match rendered image  
 fig.PaperSize = [figure\_width, figure\_height]; %make paper and fig match  
 drawnow;  
  
 file\_path = sprintf('%s\\%s.pdf', path, file\_name);  
 print(fig, file\_path, '-dpdf', '-r300'); % pdf for LaTeX  
  
end  
  
end

## Generate Chebyshev Polynomials

function [cheby\_functions] = generate\_chebyshev(cheb\_resolution, num\_cheby)  
%Generate a matrix of 'num\_cheby' functions, with depth/resolution of  
%'cheb\_resolution'  
%Input:  
 %cheb\_resolution: scalar - resolution of each functions, 'height' of matrix  
 %num\_cheby: scalar - number of chebyshevs to generate  
%Output:  
 %cheby\_functions: matrix - chebyshev functions, type 1  
  
cheby\_x = linspace(-1, 1, cheb\_resolution)'; %define the cheby 'x'  
  
cheby\_functions = ones(cheb\_resolution, num\_cheby);  
cheby\_functions(:, 2) = cheby\_x;  
  
for ndx = 3:num\_cheby %this is the recursive cheby generation. Could be from a file, but more helpful to see  
 cheby\_functions(:, ndx) = 2 \* cheby\_x .\* cheby\_functions(:, ndx - 1) - cheby\_functions(:, ndx - 2); %build cheby out  
end  
  
end

## Batch Generate Conjugate Basis Functions

function [conjugate\_basis\_functions, conjugate\_betas] = generate\_conjugate(basis\_resolution,num\_basis, P, Q, R, d, y\_star)  
%Create conjugate basis functions for a system, derived from chebyshevs  
%Input:  
 %height: scalar - height/resolution of the conjugate functions to generate  
 %num\_basis: scalar - num\_basis/number of functions to generate  
 %P: matrix - system matrix relating   
 %Q: matrix - cost of states  
 %d: vector - IC vector  
 %y\_star: vector - goal output  
%Output:  
 %conjugate\_basis\_functions: matrix - funcions that satify conjunctionality for the system P  
 %conjuagte\_betas: vector - optimal weighting to minimize error  
  
if nargin < 7  
 y\_star = zeros(height(P), 1); %default goal to zeros  
end  
if nargin < 6  
 d = zeros(height(P), 1); %default noise to zeros  
end  
if nargin < 5  
 R = 0 \* eye(height(P)); %default input cost to 0  
end  
if nargin < 4  
 Q = 100 \* eye(width(P)); %default state costs to 100  
end  
  
%Create chebys  
batch\_input = generate\_chebyshev(basis\_resolution, num\_basis);  
  
%Generate Batch output  
batch\_outputs\_delta = P \* batch\_input; %do not include the d term, because we want the difference in outputs, which excludes d  
  
W = batch\_input' \* R \* batch\_input + batch\_outputs\_delta' \* Q \* batch\_outputs\_delta; %W matrix   
  
while rank(W) < min(basis\_resolution, num\_basis) %if dont get a full rank W  
 batch\_input = rand\_range(basis\_resolution, num\_basis, -10, 10); %go different input approach  
   
 %Generate Batch output  
 batch\_outputs\_delta = P \* batch\_input; %do not include the d term, because we want the difference in outputs, which excludes d  
   
 W = batch\_input' \* R \* batch\_input + batch\_outputs\_delta' \* Q \* batch\_outputs\_delta; %W matrix   
 sprintf('Using random inputs')  
end  
  
rho\_batch = chol(W); %cholesky decomposition of W to get the optimal coeffecients for the batch  
T\_b = batch\_input / (rho\_batch); %/ is same as \* inv()  
  
conjugate\_basis\_functions = T\_b;  
  
H\_b = batch\_outputs\_delta / (rho\_batch);  
conjugate\_betas = H\_b' \* Q \* (y\_star - d); %determined optimal weights for given basis functions (off of e\_0)  
  
  
end

## Iteratively Generate Conjugate Basis Functions

function [Episode] = generate\_iterative\_conjugate(P, d, y\_star, old\_episodes, Q, R)  
%From an exisiting conjugate basis space, apply a new episode to generate a  
%new basis  
%Input:  
 %P: matrix -descriptie matrix to map u to y (y = Pu + d)  
 %d: vector - handle noise in the relation to map u to y  
 %y\_star: vector - goal output  
 %old\_episodes: struct - structure holding old iterative data  
 %Q: matrix - cost of states  
 %R: matrix - cost of inputs  
%Output:  
 %Episode: struct - all the previous trials and info, plus the new one  
  
num\_ilc\_inputs = width(P);  
num\_ilc\_states = height(P);  
  
if nargin < 6%if no R  
 R = zeros(num\_ilc\_inputs);  
end  
if nargin < 5%if no Q  
 Q = 100 \* eye(num\_ilc\_states);  
end  
if ((nargin < 4) || isempty(old\_episodes)) %if we have no episodes to start, do the first trial  
 Episode(1).del\_u = ones(num\_ilc\_inputs, 1); %first chebyshev is all ones  
 Episode(1).del\_y = P \* Episode(1).del\_u;  
  
 %Compute W  
 W = Episode(1).del\_u' \* R \* Episode(1).del\_u + Episode(1).del\_y' \* Q \* Episode(1).del\_y;  
  
 %rho\_1, phi\_1, h\_1, and beta\_1  
 Episode(1).rho = chol(W);  
 phi\_1 = Episode(1).del\_u \* Episode(1).rho^-1;  
 h\_1 = Episode(1).del\_y \* Episode(1).rho^-1;  
 beta\_1 = h\_1' \* Q \* (y\_star - d); %use e0 for all  
  
 Episode(1).Phi = phi\_1;  
 Episode(1).Hb = h\_1;  
 Episode(1).Betas = beta\_1;  
 return  
end  
  
%If this is not our first trial  
Episode = old\_episodes;  
  
b = length(Episode);  
  
%Generate our dels  
tried\_inputs = zeros(num\_ilc\_inputs, b);  
for ndx = 1:b  
 tried\_inputs(:, ndx) = Episode(ndx).del\_u;  
end  
  
gen\_cheby = generate\_chebyshev(num\_ilc\_inputs, b + 1); %there is likely a more effecient way, but to ensure all the generations are exactly the same  
next\_input = gen\_cheby(:, b+1); %next input is our final cheby  
while (rank([tried\_inputs, next\_input]) < (b+1)) %ensure that our added input does not ruin the rank of the system  
 next\_input = rand\_range(num\_ilc\_inputs, 1, -1, 1);  
 return  
end  
  
Episode(b + 1).del\_u = next\_input; %-u0, but open loop  
Episode(b + 1).del\_y = P \* Episode(b + 1).del\_u;  
  
%Compute first component  
Episode(b + 1).rho(1) = 1/Episode(1).rho(1) \* (Episode(1).del\_u' \* R \* Episode(b + 1).del\_u + Episode(1).del\_y' \* Q \* Episode(b + 1).del\_y);  
%Middle Components  
if b >= 2  
 for i = 2:b  
 sum\_term = 0;  
 for j = 1:(i-1)  
 sum\_term = sum\_term + Episode(i).rho(j) \* Episode(b + 1).rho(j); %hard to vector format across structs  
 end  
 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);  
 end  
end  
  
%Last Component  
Episode(b + 1).gamma = Episode(b + 1).rho(1:b) \* Episode(b + 1).rho(1:b)';  
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);  
  
%Phi calculation  
new\_basis = (1/Episode(b + 1).rho(b+1)) \* (Episode(b + 1).del\_u - Episode(b).Phi \* Episode(b + 1).rho(1:b)');  
Episode(b+1).Phi = [Episode(b).Phi, new\_basis];  
%Hb  
new\_h = (1/Episode(b + 1).rho(b+1)) \* (Episode(b + 1).del\_y - Episode(b).Hb \* Episode(b + 1).rho(1:b)');  
Episode(b+1).Hb = [Episode(b).Hb, new\_h];  
%Betas  
new\_beta = new\_h' \* Q \* (y\_star - d);  
Episode(b+1).Betas = [Episode(b).Betas; new\_beta];  
  
end

## ILC Simulation

function [ILC\_Trial] = basis\_ilc\_sim(P, d, y0, T\_u, T\_y, y\_star, num\_trials, error\_controller)  
%Generate the ILC trials and representative data for a given system (P, d)  
%with initial output y0, and basis description (T\_u, T\_y) to move to a goal (y\_star) over specified  
%trial count (num\_trials)  
%Inputs:  
 %P: matrix - inputs u(0->(p-1)) to outputs y(1->p)  
 %d: vector - noise/initial conditions matrix  
 %y0: vector - initial output (since y(0) skipped by P)  
 %T\_u: matrix - Input basis functions (default to identity)  
 %T\_y: matrix - Output basis functions (default to identity)  
 %y\_star: vector - goal output  
 %num\_trials: scalar - how many trials to simulate  
 %error\_controller: matrix - controller of the ILC system (defaults to  
 %0.8)  
%Outputs:  
 %ILC\_Trial: structure with indexed by trial number, contains  
 %inputs  
 %betas (input basis weights)  
 %del\_betas (change in input betas)  
 %outputs  
 %alphas (output basis weights)  
 %output error (y\* - y)  
 %alpha error  
  
if T\_u == -1  
 T\_u = eye(width(P));  
end  
if T\_y == -1  
 T\_y = eye(height(P));  
end  
  
num\_basis\_input = width(T\_u); %basis functions are pr x n\_u  
num\_basis\_output = width(T\_y); %pm x n\_y  
  
T\_y\_pinv = pinv(T\_y); %most common form of T\_y we will use (to go from real to basis land)  
alpha\_star = T\_y\_pinv \* y\_star;  
  
H = T\_y\_pinv \* P \* T\_u; %descriptive IO controller, such that alpha = H \* beta + T\_y+ \* d  
  
if (~is\_controllable(eye(num\_basis\_output), -H))  
 fprintf('The ILC System is not Controllable with %.d basis inputs and %.d basis outputs!\n', num\_basis\_input, num\_basis\_output)  
end  
  
if ~exist('error\_controller', 'var')  
 error\_controller = 0.5 \* pinv(H); %perfect knowledge controller as default  
end  
  
%Preallocate structure  
ILC\_Trial(num\_trials).input = []; %input  
ILC\_Trial(num\_trials).betas = []; %basis representation of input  
ILC\_Trial(num\_trials).del\_betas = [];  
ILC\_Trial(num\_trials).output = []; %output  
ILC\_Trial(num\_trials).alphas = []; %basis representation of output  
ILC\_Trial(num\_trials).output\_error = []; %output error  
ILC\_Trial(num\_trials).alpha\_error = []; %alpha error  
  
%Initial trial  
trial = 1;  
%Inputs  
ILC\_Trial(trial).del\_betas = zeros(num\_basis\_input, 1); %no 'input'  
ILC\_Trial(trial).betas = zeros(num\_basis\_input, 1);  
ILC\_Trial(trial).input = T\_u \* ILC\_Trial(trial).betas;  
  
%Outputs  
relevant\_output = P \* ILC\_Trial(trial).input + d; %y(1) -> y(p)  
ILC\_Trial(trial).output = [y0; relevant\_output];  
ILC\_Trial(trial).alphas = T\_y\_pinv \* relevant\_output; %alpha(j) = T\_y+ \* y(1:p)  
  
%Errors  
ILC\_Trial(trial).output\_error = y\_star - relevant\_output;  
ILC\_Trial(trial).alpha\_error = alpha\_star - ILC\_Trial(trial).alphas;  
  
trial = trial + 1;  
  
%Simulate Rest  
for trial\_num = 2:num\_trials  
 %Inputs  
 ILC\_Trial(trial).del\_betas = error\_controller \* ILC\_Trial(trial - 1).alpha\_error; %del\_B(j) = L \* e\_alpha(j-1)  
 ILC\_Trial(trial).betas = ILC\_Trial(trial - 1).betas + ILC\_Trial(trial).del\_betas; %B(j) = B(j-1) + del\_B(j)  
 ILC\_Trial(trial).input = T\_u \* ILC\_Trial(trial).betas; %u(j) = T\_u \* B(j)  
  
 %Outputs  
 relevant\_output = P \* ILC\_Trial(trial).input + d; %y(1) -> y(p)  
 ILC\_Trial(trial).output = [y0; relevant\_output];  
 ILC\_Trial(trial).alphas = T\_y\_pinv \* relevant\_output; %alpha(j) = T\_y+ \* y(1:p)  
  
 %Errors  
 ILC\_Trial(trial).output\_error = y\_star - relevant\_output;  
 ILC\_Trial(trial).alpha\_error = alpha\_star - ILC\_Trial(trial).alphas;  
  
 trial = trial + 1;  
end  
  
end

## P Matrix from ABCD Model

function [P, d] = P\_from\_ABCD(A, B, C, D, p, x0)  
%Construct the P matrix and d matrix for given matrix values  
%Will satisfy the equations y\_bar = P \* u\_bar + d  
%y is y(1) -> y(p)  
%u is u(0) -> u(p-1)  
%d captures initial conditions and noise  
%Inputs:  
 %A: matrix - state dynamics matrix  
 %B: matrix - input dynamics matrix  
 %C: matrix - state to output  
 %D: matrix - input to output  
 %p: scalar - number of steps to map out  
 %x0: vector - initial conditions  
%Outputs:  
 %P: matrix - ILC system mapping matrix for u(0->(p-1)) -> y(1->p)  
 %d: vector - captures the 'disturbance' caused by initial conditions  
  
num\_states = width(A);  
num\_inputs = width(B);  
num\_outputs = height(C);  
  
%Construct 'True' P  
P = zeros(num\_states, num\_inputs);  
ic\_matrix = zeros(num\_outputs, num\_states); %matrix which governs the impact of the Ics (for sim)  
for row = 1:p  
 row\_start = ((row - 1) \* num\_outputs) + 1;  
 row\_end = row\_start + num\_outputs - 1;  
 for col = 1:row  
 col\_start = ((col - 1) \* num\_inputs) + 1;  
 col\_end = col\_start + num\_inputs - 1;  
 if (row + 1 == col)  
 P(row\_start:row\_end, col\_start:col\_end) = D;  
 else  
 mat\_pow = row - col;  
 P(row\_start:row\_end, col\_start:col\_end) = C \* (A^mat\_pow) \* B;  
 end  
  
 end  
 ic\_matrix(row\_start:row\_end, :) = C \* A^row; %construct the IC matrix  
end  
d = ic\_matrix \* x0;  
  
%Verify P is accurate  
demo\_in = rand\_range(num\_inputs \* p, 1, -2, 2);  
demo\_P\_out = P \* demo\_in + d; %y = Pu + d  
  
demo\_dlsim\_in = reshape(demo\_in, num\_inputs, [])'; %dlsim takes a pxr matrix, whereas P is a pr x 1  
demo\_dlsim\_out\_matrix = dlsim(A, B, C, D, [demo\_dlsim\_in; zeros(1, num\_inputs)], x0);  
demo\_dlsim\_out = reshape(demo\_dlsim\_out\_matrix(2:end, :)', [], 1);  
  
if (norm(demo\_P\_out - demo\_dlsim\_out)/numel(demo\_P\_out) > 1e-6)  
 fprintf('P does not capture output accurately')  
end  
  
  
end

## Plot Basis Coefficients

function [alpha\_error\_fig, beta\_error\_fig, alpha\_prog\_fig, beta\_prog\_fig] = plot\_ilc\_coeffecients(figure\_name, graph\_title, ILC\_Trial, to\_plot, beta\_star, betas\_to\_plot, alphas\_to\_plot, save\_path)  
%Note: beta star error is sometimes illogical  
%Plot out the progression of coeffecients in an ILC problem described in a  
%basis space  
%Inputs:  
 %figure\_name: string - name of the figure that is opened, or precursor to saved image  
 %graph\_title: string - displayed title header on plot  
 %ILC\_Trial: structure - containing iterations of coeffecients  
 %to\_plot: scalar/vector - indicate which trials or how many to plot  
 %beta\_star: vector - sometimes illogical, but when there is a defined beta weights, use to compute the error  
 %betas\_to\_plot: scalar/vector - indicate which beta values (or how many) to plot  
 %alphas\_to\_plot: scalar/vector - see above  
 %save\_path: string - where to save the generated plots to  
%Outputs:  
 %alpha\_error\_fig: figure - handle to alpha errors  
 %beta\_error\_fig: figure - handle to beta erros  
 %alpha\_prog\_fig: figure - handle to showing alpha evolve  
 %beta\_prog\_fig: figure - handle to showing betas evolve  
  
max\_coef = 5; %default maximum # of coeffeceints to plot  
if nargin < 8 %no save path  
 save\_path = -1;  
end  
if nargin < 7 %if no alpha to plot  
 alphas\_to\_plot = max\_coef;  
end  
if nargin < 6  
 betas\_to\_plot = max\_coef;  
end  
if nargin < 5  
 beta\_star = -1; %ensure we have a mtrix for mathing  
end  
beta\_star\_passed = true;  
if isscalar(beta\_star)  
 if(beta\_star == -1)  
 beta\_star\_passed= false;  
 end  
end  
  
num\_trials = length(ILC\_Trial);  
marker\_color\_scale = 0.75; %how to scale the color difference from line to point  
marker\_size = 20;  
subtitle\_size = getappdata(groot, 'DefaultSubtitleFontSize');  
  
if isscalar(to\_plot) %check if we are passing in a count or specific trials to plot  
 trials\_to\_plot = floor(linspace(1, num\_trials, to\_plot));  
 trials\_to\_plot(end) = num\_trials;  
else  
 trials\_to\_plot = to\_plot;  
end  
  
%Process ILC Strcture  
num\_betas = length(ILC\_Trial(1).betas);  
num\_alphas = length(ILC\_Trial(1).alphas);  
num\_plot = length(trials\_to\_plot);  
  
%Determine which coeffecients to plot  
if isscalar(alphas\_to\_plot)  
 if(alphas\_to\_plot == -1) %default plot  
 alphas\_to\_plot = max\_coef;  
 end  
 alphas\_to\_plot = floor(linspace(1, num\_alphas, alphas\_to\_plot)); %space out the plots   
 alphas\_to\_plot(end) = num\_alphas; %ensure the last one is shown  
end  
if isscalar(betas\_to\_plot)  
 if(betas\_to\_plot == -1) %default plot  
 betas\_to\_plot = max\_coef;  
 end  
 betas\_to\_plot = floor(linspace(1, num\_alphas, betas\_to\_plot)); %space out the plots   
 betas\_to\_plot(end) = num\_betas; %ensure the last one is shown  
end  
  
%Progression of errors on coeffecients  
alpha\_error\_norm = zeros(num\_trials, 1);  
beta\_error\_norm = zeros(num\_trials, 1);  
for ndx = 1:num\_trials  
 alpha\_error\_norm(ndx) = norm(ILC\_Trial(ndx).alpha\_error) / num\_alphas;  
 if beta\_star\_passed  
 beta\_error\_norm(ndx) = norm(beta\_star - ILC\_Trial(ndx).betas) / num\_betas;  
 end  
end  
  
alpha\_error\_fig = figure('Name', sprintf('%s - Alpha Error Norm Progression', figure\_name));  
plot(1:num\_trials, alpha\_error\_norm, 'r\*', 'LineStyle',':');  
title('Alpha Error', graph\_title);  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Trial Number')  
ylabel('Normalized Amplitude')  
  
if beta\_star\_passed %if we were given a beta\_star  
 beta\_error\_fig = figure('Name', sprintf('%s - Beta Error Norm Progression', figure\_name));  
 plot(1:num\_trials, beta\_error\_norm, 'r\*', 'LineStyle',':');  
 title('Beta Error', graph\_title)  
 subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
 xlabel('Trial Number')  
 ylabel('Normalized Amplitude')  
end  
  
%Transfer structure to array for plotting  
%For progression smoothing  
full\_betas = zeros(num\_betas, num\_trials);  
full\_alphas = zeros(num\_alphas, num\_trials);  
for ndx = 1:num\_trials  
 full\_betas(:, ndx) = ILC\_Trial(ndx).betas;  
 full\_alphas(:, ndx) = ILC\_Trial(ndx).alphas;  
end  
  
%Progression of Alphas  
alpha\_prog\_fig = figure('Name', sprintf('%s - Alpha Coeffecient Progression', figure\_name));  
legend\_list = "[REMOVE ME]";  
hold on;  
for alpha = alphas\_to\_plot  
 temp\_plot = plot(0:(num\_trials-1), full\_alphas(alpha, :), 'MarkerSize', marker\_size, 'LineStyle', '-', 'Marker', 'o', 'MarkerIndices', trials\_to\_plot);  
 temp\_plot.MarkerFaceColor = temp\_plot.Color \* marker\_color\_scale;%make the marker colors slightly darker than lines  
 temp\_plot.MarkerEdgeColor = temp\_plot.Color \* marker\_color\_scale;  
 legend\_list(end + 1) = sprintf('Alpha %.d', alpha);  
end  
hold off;  
  
title('Alpha Coeffecients')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Trial Number')  
ylabel('Alpha Value')  
xticks(trials\_to\_plot - 1);   
legend(legend\_list(2:end))  
  
%Progression of Beats  
beta\_prog\_fig = figure('Name', sprintf('%s - Beta Coeffecient Progression', figure\_name));  
legend\_list = "[REMOVE ME]";  
hold on;  
for beta = betas\_to\_plot  
 temp\_plot = plot(0:(num\_trials-1), full\_betas(beta, :), 'MarkerSize', marker\_size, 'LineStyle', '-', 'Marker', 'o', 'MarkerIndices', trials\_to\_plot);  
 temp\_plot.MarkerFaceColor = temp\_plot.Color \* marker\_color\_scale; %make the marker colors slightly darker than lines  
 temp\_plot.MarkerEdgeColor = temp\_plot.Color \* marker\_color\_scale;  
 legend\_list(end + 1) = sprintf('Beta %.d', beta);  
end  
hold off;  
  
title('Beta Coeffecients')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Trial Number')  
ylabel('Beta Value')  
xticks(trials\_to\_plot - 1);   
legend(legend\_list(2:end))  
  
%Save images if possible  
if exist('save\_path', 'var') %if a save path was provided  
 save\_figure(save\_path, [alpha\_error\_fig, alpha\_prog\_fig, beta\_prog\_fig]); %save all the figures to the path  
 if beta\_star\_passed  
 save\_figure(save\_path, beta\_error\_fig);  
 end  
end  
  
end

## Plot Controller History

function [figure\_list] = plot\_controller\_history(figure\_name, graph\_title, controllers, inputs\_of\_interest, states\_of\_interest, save\_path, decoupled)  
%Plot the progression of a given controller parameters for indicated inputs  
%Inputs:  
 %figure\_name: string - name of the figure that is opened, or precursor to saved image  
 %graph\_title: string - displayed title header on plot  
 %controllers: matrix - r x n x l matrix, showing the learning of r controllers from n states over l iterations  
 %inputs\_of\_interest: scalar/vector - for when there are lots of inputs, which ones to plot  
 %states\_of\_interest: scalar/vector - see above  
 %save\_path: string - where to save the generated plots to  
 %decoupled: bool - indicates whether or not it was learned via input decoupling (only show trials where input changes)  
%Outputs:  
 %figure\_list: list - figure handles generated  
  
%Plot settings  
marker\_color\_scale = 0.75; %how to scale the color difference from line to point  
marker\_size = 20;  
subtitle\_size = getappdata(groot, 'DefaultSubtitleFontSize'); %default set in fig properties  
  
num\_inputs = height(controllers);  
num\_states = width(controllers);  
  
%Assign defaultss  
if nargin < 7 %if flag of whether or not it was input decoupled  
 decoupled = false;  
end  
if nargin < 6   
 save\_path = -1;  
end  
if nargin < 5 %if no states index passed  
 states\_of\_interest = floor(linspace(1, num\_states, min(num\_states, 4))); %do not overclutter sith state points  
end  
if nargin < 4 %if no passed input indexes  
 inputs\_of\_interest = [1, num\_inputs]; %default to showing 2 of them (sometimes there are a lot)  
end  
  
%Shortcuts  
if (states\_of\_interest == -1) %if -1, plot all components  
 states\_of\_interest = 1:num\_states;  
elseif (isscalar(states\_of\_interest))  
 states\_of\_interest = floor(linspace(1, num\_states, states\_of\_interest));  
 states\_of\_interest(end) = num\_states;  
end  
  
if (inputs\_of\_interest == -1)  
 inputs\_of\_interest = 1:num\_inputs;  
elseif (isscalar(inputs\_of\_interest))  
 inputs\_of\_interest = floor(linspace(1, num\_inputs, inputs\_of\_interest));  
 inputs\_of\_interest(end) = num\_inputs;  
end  
  
figure\_list = [];  
for input\_num = inputs\_of\_interest  
 figure\_list(end + 1) = figure('Name', sprintf('%s - Input %.d Controller Weights', figure\_name, input\_num));  
   
 if decoupled  
 trial\_ndx = [1, (input\_num+1):num\_inputs:size(controllers, 3)]; %if input decoupled, alternate to only show points when updated  
 else  
 trial\_ndx = 1:size(controllers, 3);  
 end  
  
 legend\_list = "[REMOVE ME]";  
 hold on;  
 for state = states\_of\_interest  
 temp\_plot = plot((trial\_ndx-1), squeeze(controllers(input\_num, state, trial\_ndx)), 'MarkerSize', marker\_size, 'Marker', 'o');  
 temp\_plot.MarkerFaceColor = temp\_plot.Color \* marker\_color\_scale;%make the marker colors slightly darker than lines  
 temp\_plot.MarkerEdgeColor = temp\_plot.Color \* marker\_color\_scale;  
 legend\_list(end + 1) = sprintf('Weight on State %.d', state);  
 end  
 hold off;  
  
 legend(legend\_list(2:end), "BackgroundAlpha", 0.5)  
 title(sprintf('Input %.d Controller Weights', input\_num));  
 subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
 xlabel('Controller Number')  
 ylabel('Controller Weight')  
 xticks(trial\_ndx-1);   
  
  
end  
  
%Save images if possible  
if exist('save\_path', 'var') %if a save path was provided  
 for fig\_num = figure\_list  
 save\_figure(save\_path, figure(fig\_num)); %save all the figures to the path  
 end  
end  
  
  
end

## Plot Dual-Mass System

function [mass1\_fig, mass2\_fig, input1\_fig, input2\_fig] = plot\_two\_mass(figure\_name, graph\_title, outputs, inputs, save\_path)  
%For the dual spring-mass system repeatedly used, plot positions and inputs  
%Inputs:  
 %figure\_name: string - name of the figure that is opened, or precursor to saved image  
 %graph\_title: string - displayed title header on plot  
 %outputs: matrix - 2 x k matrix, where row 1 is mass 1 position, row 2 is mass 2 position  
 %inputs: matrix - 2 x k, row 1 is input1 and row 2 is input 2  
 %save\_path: string - where to save the generated plots to  
%Outputs:  
 %mass1\_fig: figure - handle to position of mass 1  
 %mass2\_fig: figure - handle to position of mass 2  
 %input1\_fig: figure - handle to input 1  
 %input2\_fig: figure - handle to input 2  
  
  
if (height(outputs) > width(outputs)) %Ensure wide  
 outputs = outputs';  
end  
  
if (height(inputs) > width(inputs)) %Ensure wide  
 inputs = inputs';  
end  
  
num\_samples = width(outputs);  
subtitle\_size = getappdata(groot, 'DefaultSubtitleFontSize');  
  
%Output Style  
output\_color = [0 0.4471 0.7412];  
output\_size = 1.2;  
output\_style = '-';  
  
%Mass 1 Position  
mass1\_fig = figure('Name', sprintf('%s - Mass 1 Position', figure\_name));  
%stairs(0:(num\_samples-1), outputs(1, :), 'Color', output\_color, 'LineStyle', output\_style, 'LineWidth', output\_size);   
stairs(0:(num\_samples-1), outputs(1, :), 'Color', output\_color);  
title('Mass 1 Position');  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Position (m)')  
  
%Mass 2 Position  
mass2\_fig = figure('Name', sprintf('%s - Mass 2 Position', figure\_name));  
stairs(0:(num\_samples-1), outputs(2, :), 'Color', output\_color);   
title('Mass 2 Position')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Position (m)')  
  
%Input Style  
input\_color = [1, 0, 1];  
input\_size = 1.2;  
input\_style = '-';  
  
%Input 1  
input1\_fig = figure('Name', sprintf('%s - Input 1 Magnitude', figure\_name));  
stairs(0:(num\_samples-1), inputs(1, :), 'Color', input\_color);   
title('Input 1')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Force (N)')  
  
%Input 2  
input2\_fig = figure('Name', sprintf('%s - Input 2 Magnitude', figure\_name));  
stairs(0:(num\_samples-1), inputs(2, :), 'Color', input\_color);   
title('Input 2')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Force (N)')  
  
%Save images if possible  
if exist('save\_path', 'var') %if a save path was provided  
 save\_figure(save\_path, [mass1\_fig, mass2\_fig, input1\_fig, input2\_fig]); %save all the figures to the path  
end  
  
end

## Plot Iterative Learning Control Problem

function [error\_fig, out1\_fig, out2\_fig, dual\_out\_fig, input1\_fig, input2\_fig] = plot\_dual\_ilc(figure\_name, graph\_title, ILC\_Trial, goal\_output, goal\_input, to\_plot, save\_path)  
%Plot the ILC Structure information, showing the input and output history  
%for a two-spring-mass system through ILC trials  
%Inputs:  
 %figure\_name: string - name of the figure that is opened, or precursor to saved image  
 %graph\_title: string - displayed title header on plot  
 %ILC\_Trial: structure - containing iterations of coeffecients  
 %goal\_output: vector - y\*, what we want to generate  
 %goal\_input: vector - u\*, what input gets us there  
 %to\_plot: scalar/vector - indicate which trials or how many to plot  
 %save\_path: string - where to save the generated plots to  
%Outputs:  
 %error\_fig: figure - handle to the error progression  
 %out1\_fig: figure - handle to position of mass 1  
 %out2\_fig: figure - handle to position of mass 2  
 %dual\_out\_fig: figure - handle to figure showing a 'shaped' output  
 %input1\_fig: figure - handle to input 1  
 %input2\_fig: figure - handle to input 2  
  
%legend\_alpha = 0.5;  
subtitle\_size = getappdata(groot, 'DefaultSubtitleFontSize');  
  
if nargin < 7 %if no file save path  
 save\_path = -1;  
end  
if nargin < 6  
 to\_plot = 5; %default to plotting 5 trials  
end  
if nargin < 5  
 goal\_input = -1;  
end  
if nargin <4  
 goal\_output = -1;  
end  
  
  
%Process ILC Strcture  
num\_trials = length(ILC\_Trial);  
if isscalar(to\_plot) %check if we are passing in a count or specific trials to plot  
 trials\_to\_plot = floor(linspace(1, num\_trials, to\_plot));  
 trials\_to\_plot(end) = num\_trials;  
else  
 trials\_to\_plot = to\_plot;  
end  
  
%There is potential to clean up the code here, but readability is more  
%important  
num\_inputs = length(ILC\_Trial(1).input);  
num\_outputs = length(ILC\_Trial(1).output);  
  
out1\_ndx = 1:2:num\_outputs;  
out2\_ndx = 2:2:num\_outputs;  
  
in1\_ndx = 1:2:num\_inputs;  
in2\_ndx = 2:2:num\_inputs;  
  
%Error Progression  
error\_progression = zeros(num\_trials, 1);  
for ndx = 1:num\_trials %convert structure to array  
 error\_progression(ndx) = norm(ILC\_Trial(ndx).output\_error)/num\_outputs; %normalize the errors  
end  
error\_fig = figure('Name', sprintf('%s - Error Progression', figure\_name));  
plot(1:num\_trials, error\_progression, 'r\*', 'LineStyle',':');  
title('Error Magnitude');  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Trial Number')  
ylabel('Normalized Amplitude')  
  
%Mass 1 Position  
out1\_fig = figure('Name', sprintf('%s - Mass 1 Position', figure\_name));  
legend\_list = "[REMOVE ME]"; %teach it we're building strings  
hold on;  
for ndx = 1:length(trials\_to\_plot)  
 trial\_num = trials\_to\_plot(ndx); %which trial of the process is being plotted  
 stairs(0:(num\_outputs/2-1), ILC\_Trial(trial\_num).output(out1\_ndx));  
 legend\_list(end + 1) = sprintf('Trial %.d', trial\_num);  
end  
if ~(isscalar(goal\_output) && (goal\_output == -1))  
 stairs(1:((num\_outputs-1)/2), goal\_output(out1\_ndx(1:(end-1))), 'Color', [1, 0, 0], 'LineStyle', '--');  
 legend\_list(end + 1) = 'Goal Output';  
end  
hold off;  
legend(legend\_list(2:end));%, "BackgroundAlpha", legend\_alpha);  
title('Mass 1 Position')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Position (m)')  
  
%Mass 2 Position  
out2\_fig = figure('Name', sprintf('%s - Mass 2 Position', figure\_name));  
legend\_list = "[REMOVE ME]"; %reset legend list  
hold on;  
for ndx = 1:length(trials\_to\_plot)  
 trial\_num = trials\_to\_plot(ndx); %which trial of the process is being plotted  
 stairs(0:(num\_outputs/2-1), ILC\_Trial(trial\_num).output(out2\_ndx));  
 legend\_list(end + 1) = sprintf('Trial %.d', trial\_num);  
end  
  
if ~(isscalar(goal\_output) && (goal\_output == -1))  
 stairs(1:((num\_outputs-1)/2), goal\_output(out2\_ndx(1:(end-1))), 'Color', [1, 0, 0], 'LineStyle', '--');  
 legend\_list(end + 1) = 'Goal Output';  
end  
hold off;  
legend(legend\_list(2:end));%, "BackgroundAlpha", legend\_alpha);  
title('Mass 2 Position')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Position (m)')  
  
%Combined Outputs (for fun 2D shapes)  
dual\_out\_fig = figure('Name', sprintf('%s - Shaped Output', figure\_name));  
legend\_list = "[REMOVE ME]"; %reset legend list  
hold on;  
for ndx = 1:length(trials\_to\_plot)  
 trial\_num = trials\_to\_plot(ndx); %which trial of the process is being plotted  
 plot(ILC\_Trial(trial\_num).output(out1\_ndx), ILC\_Trial(trial\_num).output(out2\_ndx));  
 legend\_list(end + 1) = sprintf('Trial %.d', trial\_num);  
end  
if ~(isscalar(goal\_output) && (goal\_output == -1))  
 plot(goal\_output(out1\_ndx(1:(end-1))), goal\_output(out2\_ndx(1:(end-1))), 'Color', [1, 0, 0], 'LineStyle', '--');  
 legend\_list(end + 1) = 'Goal Output';  
 %Add a start/stop position indicator  
 scatter(goal\_output(1), goal\_output(2), 'filled', '>', 'MarkerFaceColor', [0, 1, 0])  
 scatter(goal\_output(end - 1), goal\_output(end), 'filled', 'hexagram', 'MarkerFaceColor', [1, 0, 0])  
 legend([legend\_list(2:end), 'Start', 'Stop']);%, "BackgroundAlpha", legend\_alpha);  
end  
hold off;  
title('Shaped Outputs')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Mass 1 Position (m)')   
ylabel('Mass 2 Position (m)')  
axis equal %shapes should be equally scaled  
  
  
%Input 1  
input1\_fig = figure('Name', sprintf('%s - Input 1', figure\_name));  
legend\_list = "[REMOVE ME]"; %reset legend list  
hold on;  
for ndx = 1:length(trials\_to\_plot)  
 trial\_num = trials\_to\_plot(ndx); %which trial of the process is being plotted  
 stairs(0:(num\_inputs/2-1), ILC\_Trial(trial\_num).input(in1\_ndx));  
 legend\_list(end + 1) = sprintf('Trial %.d', trial\_num);  
end  
if ~(isscalar(goal\_input) && (goal\_input == -1))  
 stairs(0:((num\_inputs/2)-1), goal\_input(in1\_ndx), 'Color', [1, 0, 0], 'LineStyle', '--');  
 legend\_list(end + 1) = 'Goal Input';  
end  
hold off;  
legend(legend\_list(2:end));%, "BackgroundAlpha", legend\_alpha);  
title('Input 1')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Input (N)')  
  
%Input 2  
input2\_fig = figure('Name', sprintf('%s - Input 2', figure\_name));  
legend\_list = "[REMOVE ME]"; %reset legend list  
hold on;  
for ndx = 1:length(trials\_to\_plot)  
 trial\_num = trials\_to\_plot(ndx); %which trial of the process is being plotted  
 stairs(0:(num\_inputs/2-1), ILC\_Trial(trial\_num).input(in2\_ndx));  
 legend\_list(end + 1) = sprintf('Trial %.d', trial\_num);  
end  
if ~(isscalar(goal\_input) && (goal\_input == -1))  
 stairs(0:((num\_inputs/2)-1), goal\_input(in2\_ndx), 'Color', [1, 0, 0], 'LineStyle', '--');  
 legend\_list(end + 1) = 'Goal Input';  
end  
hold off;  
legend(legend\_list(2:end));%, "BackgroundAlpha", legend\_alpha);  
title('Input 2')  
subtitle(sprintf('%s', graph\_title), 'FontSize', subtitle\_size)  
xlabel('Sample Number (k)')   
ylabel('Input (N)')  
  
%Save images if possible  
if exist('save\_path', 'var') %if a save path was provided  
 save\_figure(save\_path, [error\_fig, out1\_fig, out2\_fig, input1\_fig, input2\_fig]); %save all the figures to the path  
 save\_figure(save\_path, dual\_out\_fig, -1, true); %this one is shapede  
end  
  
end

## Policy Learning for Iterative Learning Control

function [ILC\_Trial, F\_policy, controller\_error, F\_lqr] = policy\_ilc(P, d, C, D, x0, y\_star, gamma, Q, R, num\_controllers, exploration\_mag, input\_basis\_functions, output\_basis\_functions, num\_converged, existing\_controller, existing\_T\_u, existing\_T\_y)  
%Perform the RL policy learning on an ILC system since done so much in  
%thesis  
%Inputs:  
 %P: matrix - inputs u(0->(p-1)) to outputs y(1->p)  
 %d: vector - noise/initial conditions matrix  
 %C: matrix - state to output descriptor  
 %D: matrix - input to output descriptor  
 %x0: vector - initial state  
 %y\_star: vector - goal output  
 %gamma: scalar - discount factor  
 %Q: matrix - cost of states (errors)  
 %R: matrix - cost of inputs (change in inputs)  
 %num\_controllers: scalar - number of controllers to learn  
 %exploration\_mag: scalar - range around 0 to explore (defaults to 1)  
 %input\_basis\_functions: matrix - basis functions on the inputs (defaults to identity)  
 %output\_basis\_functions: matrix - basis functions on the outputs (defaults to identity)   
 %num\_converged: scalar - number of trials to simulate out without exploration (defaults to 0)  
%Outputs:  
 %ILC\_Trial: structure with indexed by trial number, contains  
 %inputs  
 %betas (input basis weights)  
 %del\_betas (change in input betas)  
 %outputs  
 %alphas (output basis weights)  
 %output error (y\* - y)  
 %alpha error  
 %F\_policy: matrix - controller learning history  
 %controller\_error: scalar - normalized error from LQR  
 %F\_lqr: matrix - goal LQR controller  
  
%Required Parameters to System Info  
num\_inputs = width(D);  
  
%Default paramters  
if ~exist('num\_converged', 'var')  
 num\_converged = 0; %default to no converged trials  
end  
if ~exist('exploration\_mag', 'var')  
 exploration\_mag = 1;  
end  
if ~exist('input\_basis\_functions', 'var')  
 input\_basis\_functions = eye(width(P));  
end  
if ~exist('output\_basis\_functions', 'var')  
 output\_basis\_functions = eye(height(P));  
 output\_basis\_functions\_pinv = output\_basis\_functions;%save on compute time  
else  
 output\_basis\_functions\_pinv = pinv(output\_basis\_functions);  
end  
num\_ilc\_states = width(output\_basis\_functions);  
num\_ilc\_inputs = width(input\_basis\_functions);  
  
%Calculate optimal controller  
F\_lqr = discounted\_LQR(eye(num\_ilc\_states), -output\_basis\_functions\_pinv \* P \* input\_basis\_functions, gamma, Q, R);  
  
alpha\_star = output\_basis\_functions\_pinv \* y\_star;  
  
%Iteration Counts  
Pj\_dim = num\_ilc\_states + num\_ilc\_inputs;  
num\_collections\_per\_controller = Pj\_dim^2;   
total\_trial\_count = num\_controllers \* num\_collections\_per\_controller + num\_converged;  
  
%Preallocate structure  
ILC\_Trial(total\_trial\_count).betas = []; %input  
ILC\_Trial(total\_trial\_count).betas = []; %basis representation of input  
ILC\_Trial(total\_trial\_count).del\_beta = [];  
ILC\_Trial(total\_trial\_count).output = []; %output  
ILC\_Trial(total\_trial\_count).alphas = []; %basis representation of output  
ILC\_Trial(total\_trial\_count).output\_error = []; %output error  
ILC\_Trial(total\_trial\_count).alpha\_error = []; %alpha error  
  
F\_policy = zeros(num\_ilc\_inputs, num\_ilc\_states, num\_controllers + 1); %start with no controller  
  
%Prepopulate the first trial  
trial\_num = 1;  
ILC\_Trial(trial\_num).betas = zeros(num\_ilc\_inputs, 1); %start with no basis guessed  
ILC\_Trial(trial\_num).input = input\_basis\_functions \* ILC\_Trial(trial\_num).betas;  
  
ILC\_Trial(trial\_num).output = [C\*x0; d]; %open loop response is IC and then d term  
ILC\_Trial(trial\_num).alphas = output\_basis\_functions\_pinv \* d;  
ILC\_Trial(trial\_num).output\_error = y\_star - d; %relevant error  
ILC\_Trial(trial\_num).alpha\_error = alpha\_star - ILC\_Trial(trial\_num).alphas;  
  
trial\_num = 2; %start at second trial now  
for iteration = 1:num\_controllers  
 Uk\_stack = zeros(num\_collections\_per\_controller, 1);  
 Xk\_stack = zeros(num\_collections\_per\_controller, (Pj\_dim)^2);  
  
 %Simulate the necessary trials  
 for trial = 1:num\_collections\_per\_controller %number of trials to collect before updatin controller  
 %ILC / Basis Controller Process  
 %Beta Coeffecients  
 exploration\_term = rand\_range(num\_ilc\_inputs, 1, -exploration\_mag, exploration\_mag); %jiggle to learn  
 ILC\_Trial(trial\_num).del\_beta = F\_policy(:, :, iteration) \* ILC\_Trial(trial\_num - 1).alpha\_error + exploration\_term;  
 ILC\_Trial(trial\_num).betas = ILC\_Trial(trial\_num - 1).betas + ILC\_Trial(trial\_num).del\_beta;  
 %Beta to Inputs  
 ILC\_Trial(trial\_num).input = input\_basis\_functions \* ILC\_Trial(trial\_num).betas;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial(trial\_num).input + d; %y(1) -> y(p)  
 ILC\_Trial(trial\_num).alphas = output\_basis\_functions\_pinv \* relevant\_output;  
 ILC\_Trial(trial\_num).output = [C\*x0 + D\*ILC\_Trial(trial\_num).input(1:num\_inputs); relevant\_output]; %total output y(0) -> y(p) for completeness  
 %Calculate Error  
 ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
 ILC\_Trial(trial\_num).alpha\_error = alpha\_star - ILC\_Trial(trial\_num).alphas;  
  
 %RL Translation  
 state = ILC\_Trial(trial\_num - 1).alpha\_error; %analogous state  
 input = ILC\_Trial(trial\_num).del\_beta; %analogous input  
 next\_state = ILC\_Trial(trial\_num).alpha\_error; %x(k+1) = e\_alpha(j)  
 next\_input = F\_policy(:, :, iteration) \* next\_state; %no exploration term here  
  
 xu\_stack = [state; input];  
 xu\_next\_stack = [next\_state; next\_input];  
  
 Xk\_stack(trial, :) = kron(xu\_stack', xu\_stack') - gamma \* kron(xu\_next\_stack', xu\_next\_stack');  
 Uk\_stack(trial, :) = input' \* R \* input + state' \* Q \* state;  
  
 trial\_num = trial\_num + 1;  
 end  
  
 %Calculate P and new controller  
 PjS = pinv(Xk\_stack) \* Uk\_stack;  
 Pj = reshape(PjS, Pj\_dim, Pj\_dim);  
 Pj = 0.5 \* (Pj + Pj'); %to impose symmetry (significantly reduces error)  
 Pjuu = Pj((num\_ilc\_states+1):end, (num\_ilc\_states+1):end);  
 PjxuT = Pj((num\_ilc\_states+1):end, 1:num\_ilc\_states);  
 new\_F = -pinv(Pjuu) \* PjxuT;  
 F\_policy(:, :, iteration + 1) = new\_F;  
end  
  
controller\_error = norm(F\_policy(:, :, end) - F\_lqr)/numel(F\_lqr);  
  
  
for ndx = 1:num\_converged  
 %ILC / Basis Controller Process  
 %Beta Coeffecients  
 ILC\_Trial(trial\_num).del\_beta = F\_policy(:, :, iteration) \* ILC\_Trial(trial\_num - 1).alpha\_error;  
 ILC\_Trial(trial\_num).betas = ILC\_Trial(trial\_num - 1).betas + ILC\_Trial(trial\_num).del\_beta;  
 %Beta to Inputs  
 ILC\_Trial(trial\_num).input = input\_basis\_functions \* ILC\_Trial(trial\_num).betas;  
 %Simulate Reality  
 relevant\_output = P \* ILC\_Trial(trial\_num).input + d; %y(1) -> y(p)  
 ILC\_Trial(trial\_num).alphas = output\_basis\_functions\_pinv \* relevant\_output;  
 ILC\_Trial(trial\_num).output = [C\*x0 + D\*ILC\_Trial(trial\_num).input(1:num\_inputs); relevant\_output]; %total output y(0) -> y(p) for completeness  
 %Calculate Error  
 ILC\_Trial(trial\_num).output\_error = y\_star - relevant\_output;  
 ILC\_Trial(trial\_num).alpha\_error = alpha\_star - ILC\_Trial(trial\_num).alphas;  
  
 trial\_num = trial\_num + 1;  
end  
  
end

## Generate Random Numbers in a Range

function [num] = rand\_range(height, width, lower, upper)  
%Return a random number in a range  
%Inputs:  
 %height: scalar - height (num rows) of random vector  
 %width: scalar - width (num columns)  
 %lower: scalar - lower bound of numbers to generate  
 %upper: scalar - upper bound  
%Outputs:  
 %num: matrix - height x width matrix of random numbers in range  
  
num = lower + rand(height, width)\*(upper - lower); %random times the amplitude, shift by lower bound  
  
end

## Plot Poles

function [fig\_pole] = plot\_pole\_placement(figure\_name, graph\_title, pole\_locations, goal\_poles, save\_path)  
%Render the pole placements (or more accurately, plot eigen values)  
%Inputs:  
 %figure\_name: string - name of the figure that is opened, or precursor to saved image  
 %graph\_title: string - displayed title header on plot  
 %pole\_locations: vector - location of poles  
 %goal\_poles: vector - where we hoped/wanted poles (not rendered if excluded)  
 %save\_path: string - location to save the figure  
%Outputs:  
 %fig\_pole: figure - handle to showing poles  
  
if ((nargin < 4) || (all(goal\_poles == -1))) %if there is no goal poles  
 goal\_poles = false;  
end  
  
fig\_pole = figure('Name', figure\_name);  
circle\_resolution = 200; %how many points to make up the circle  
circle\_domain = linspace(0, 2\*pi, circle\_resolution); %trace out a full period  
unit\_x = cos(circle\_domain);  
unit\_y = sin(circle\_domain);  
plot(unit\_x, unit\_y);  
hold on;  
scatter(real(pole\_locations), imag(pole\_locations), 'filled', 'o', 'LineWidth', 3, 'SizeData', 200);  
  
xline(0)  
yline(0)  
if (goal\_poles == false) %yes I know usually you can just do ~goal\_poles, but if this is a matrix that breaks it  
 legend('Unit Circle', 'Placed Poles')  
else  
 scatter(real(goal\_poles), imag(goal\_poles), 'o', 'SizeData', 200);  
 legend('Unit Circle', 'Pole Locations', '', '', 'Goal Poles') %skip the x and y line  
end  
hold off;  
  
xlabel('Real')  
ylabel('Imaginary')  
title('Pole Locations')  
subtitle(graph\_title, 'FontSize', getappdata(groot, 'DefaultSubtitleFontSize'));  
axis equal %ensure the circle looks like a circle  
  
  
%Save images if possible  
if exist('save\_path', 'var') %if a save path was provided  
 save\_figure(save\_path, fig\_pole, -1, true); %save all the figures to the path, marking it as shaped  
end  
  
end

1. Note that the controller depends only on the system matrices and , and not at all on initial conditions [↑](#footnote-ref-39)
2. The same logic applies for full ‘column rank’ [↑](#footnote-ref-40)
3. MATLAB’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 () [↑](#footnote-ref-47)
4. When , we can only make this reduction if stabilizes and goes to [↑](#footnote-ref-56)
5. MATLAB’s dlqr function returns a different result than the one defined by our cost function as it does not have a discount parameter. [↑](#footnote-ref-58)
6. , [↑](#footnote-ref-74)
7. To ILC, every shape is as arbitrary as the last. [↑](#footnote-ref-82)
8. Or is stable. [↑](#footnote-ref-89)
9. Note this is not the same P matrix defined in the ILC problem [↑](#footnote-ref-90)
10. For learning, is best when randomized. If following a control-law process, added exploration terms are needed (see example) [↑](#footnote-ref-91)
11. Note the ’ to transpose the vector into the column format [↑](#footnote-ref-128)
12. Each function must be 200 points in resolution, as our number of ILC inputs is 200 () [↑](#footnote-ref-138)
13. Check this yourself using and [↑](#footnote-ref-142)
14. It also works to convert the output into the output basis space and compute [↑](#footnote-ref-147)
15. See Appendix [[code:ilc\_sim]](#code:ilc_sim) for a more explicit showing of this [↑](#footnote-ref-164)
16. Unless of course [↑](#footnote-ref-175)
17. This is consistent with the findings of [↑](#footnote-ref-176)
18. Computed as [↑](#footnote-ref-181)
19. The 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. [↑](#footnote-ref-205)
20. For our given examples we have a 2-input, 2-output system such that but this is not always the case. In that case, still does not matter, but may need to be defined in different dimensions and thus not be able to equal . [↑](#footnote-ref-211)
21. This is the input which generates . [↑](#footnote-ref-212)
22. [↑](#footnote-ref-213)
23. Recall that Matlab ‘zero’ [↑](#footnote-ref-216)
24. Unlike in our RL examples where small s can lead to numerical ill-conditioning, the construction of our conjunct condition matrix prevents this. Since it relies on and as well, we can simplify our math by removing from our computations. It would be possible to conversely set and have . [↑](#footnote-ref-233)
25. Note we are not using the shown in the Batch example for the derivation, even though we could [↑](#footnote-ref-238)
26. Typically is no input such that [↑](#footnote-ref-239)
27. We are once again using the for convenience [↑](#footnote-ref-240)
28. See function *generate\_iterative\_conjugate* in Code Appendix [[code:iterative\_conj]](#code:iterative_conj) to provide a more structured understanding. [↑](#footnote-ref-241)
29. So long as they all use the same basis space on the output. [↑](#footnote-ref-256)
30. or R(1, 1) in Matlab syntax [↑](#footnote-ref-257)
31. On the topic of computational limits, we once again are limited on how small we can set are 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 [↑](#footnote-ref-258)
32. Remember we only need to fully capture , so if is not fully captured, that is ok. [↑](#footnote-ref-265)
33. Similar to the scaling of , there is potential for future work here. The produced result of this approach of 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. [↑](#footnote-ref-276)
34. The small error on the final term is negligible. [↑](#footnote-ref-277)
35. We are only describing our input in the basis space here [↑](#footnote-ref-294)