Solving the Iterative Learning Control Problem with Renforcement Learning in the Basis Space

Noah Dunleavy

March 11, 2025

Contents

Chapter 1

Background

1.1 Introduction

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 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 defined system inputs and states.

Next we introduce the Iterative Learnign Control ("ILC") problem and show that it can learn to generate any output (so long as permitted by the physical characteristics of the system).

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

1.2 Introduction to Continuous State Space

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

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

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

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

1.2.1 Example — State Space Formulation

Most, if not all, of the worlds physical systems can be modelled as a spring-mass-dampener 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

One can see that we have two masses, connected in series with springs and dampers, and bounded on one end with a wall. Constructing the equations of motion for the system simply follows Newton's second law ($F = ma$),

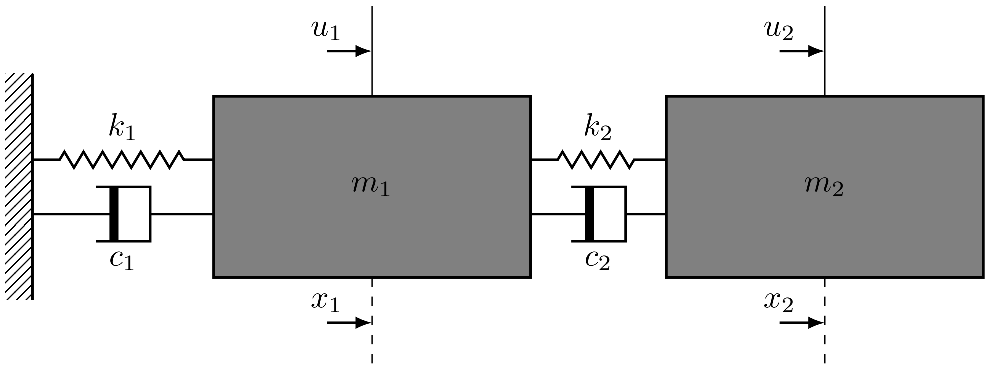

Figure 1.1: Dual Spring-Mass-Damper System

employing Hooke's ($F = -kx$) and Damping Laws ($F = -cv$). Recognizing that the derivative of position is velocity, and the derivative of that is acceleration, it can be shown that the equations of motion (EoM) for each mass are shown in Eqs. 1.3 and 1.4.

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

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

The next step is to 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 in the springs - we have four elements and thus four states. It is most common and logical in spring-mass problems to select the position and velocity of the masses. That is

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} \tag{1.5}$$

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

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1.6}$$

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

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} \tag{1.7}$$

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

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

Recognizing this simple system is already messy, further substitutions can be made by formatting the masses, spring constants, and damping coefficients into a Mass (1.9), Stiffness (1.10), and Damping (1.11) Matrices. These are known as physical parameter matrices.

$$M = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \tag{1.9}$$

$$K = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} \tag{1.10}$$

$$C = \begin{bmatrix} c_1 + c_2 & -c_2 \\ -c_2 & c_2 \end{bmatrix} \tag{1.11}$$

This would allow us to re-express our equations of motion in a single line as in Eq. 1.12

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

and our state-space model can now be written as

$$\dot{x} = \begin{bmatrix} 0_{2\times2} & I_{2\times2} \\ -M^{-1}K & -M^{-1}C \end{bmatrix} x + \begin{bmatrix} 0_{2\times2} \\ -M^{-1} \end{bmatrix} u \tag{1.13}$$

This is exactly the format for the state-space model we described earlier in Eq. 1.1. Now to put some numbers to our example so we can simulate behavior. We will define our system as follows

$$m_1 = 1\,\mathrm{kg} \quad m_2 = 0.5\,\mathrm{kg} \tag{1.14}$$

$$k_1 = \frac{100\,\mathrm{N}}{\mathrm{m}} \quad k_2 = \frac{200\,\mathrm{N}}{m} \tag{1.15}$$

$$c_1 = \frac{1\,\mathrm{Ns}}{m} \quad c_2 = \frac{0.5\,\mathrm{Ns}}{m} \tag{1.16}$$

such that our physical parameter matrices are

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{1.17}$$

$$K = \begin{bmatrix} 300 & -200 \\ -200 & 200 \end{bmatrix} \tag{1.18}$$

$$C = \begin{bmatrix} 1.5 & -0.5 \\ -0.5 & 0.5 \end{bmatrix} \tag{1.19}$$

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

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

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

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

We now only need one more piece of information to complete describe this systems behavior, and that is its initial conditions. We will choose to display the first mass one meter to the right and have the second mass moving to the right at two meters per second

$$x_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} \tag{1.22}$$

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

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

where we will once again explictly note

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

The resulting outputs from simulation out this system can be seen in Fig. 1.2 and Fig. 1.3

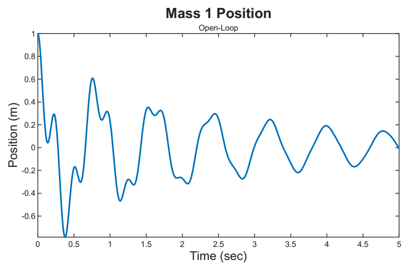

Figure 1.2: Continuous Model Open-loop Simulation Mass 1 Position

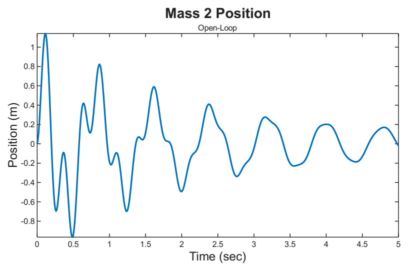

Figure 1.3: Continuous Model Open-loop Simulation Mass 2 Position

1.3 Discretization of a Continuous Model

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

$$A = e^{A_c \Delta t} \tag{1.25}$$

$$B = \int_0^{\Delta t} e^{A_c \alpha}\, d\alpha B_c \tag{1.26}$$

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

$$x(k+1) = Ax(k) + Bu(k) \tag{1.27}$$

$$y(k) = Cx(k) + Du(k) \tag{1.28}$$

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

$$t = k\Delta t \tag{1.29}$$

As with all sampling and discretization, one must be wary of Nyquist sampling. A given system with have inherent 'modes' - frequencies which it is easily excited and operates at. If your sampling rate Δt is not sufficiently small, the exactness of the model will be illogical and fail to capture true 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, even in the presence of computational limits.

1.3.1 Example — Discretization

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

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

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

Now our system is captured in discrete form, as outlined in Eq. 1.27. Running that out and overlaying it with the results of the continuous system we arrive upon outputs depicted in Figures 1.4 and 1.5.

From this zoomed out view, it can be difficult to believe that the exact relationship does exist. Figures 1.6 and 1.7 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.

One thing to note – moving forward the discrete system models x-axis will be expressed in terms of sample numbers k. For example, Figure 1.6 axis will be marked with k intervals 0 through 10, as opposed to time 0 through 0.1 seconds.

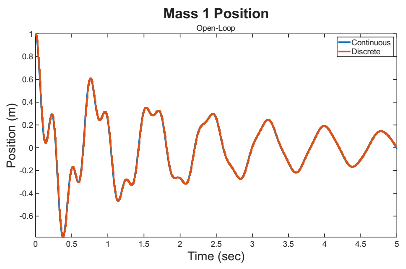

Figure 1.4: Discrete Model Open-loop Simulation Mass 1 Position

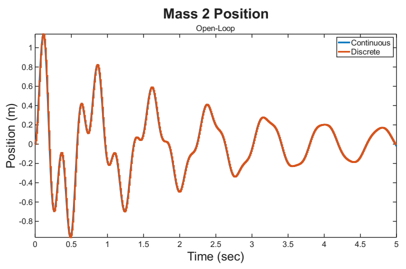

Figure 1.5: Discrete Model Open-loop Simulation Mass 2 Position

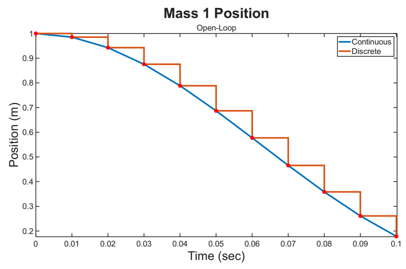

Figure 1.6: Zoomed in Discrete over Continuous Model Open-loop Simulation of Mass 1 Position

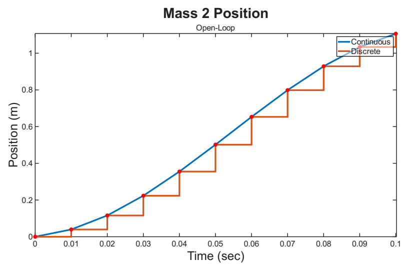

Figure 1.7: Zoomed in Discrete over Continuous Model Open-loop Simulation of Mass 2 Position

1.4 Defining Control

With our model now modified to still be exact for a computer, we can now proceed to do something with it. From a system model, it is then 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 behavior as 'closed loop'. A typical controller demarked by F will be $r \times n$ and is used to calculate inputs from collected data. As such, each input obeys the following control law when seeking stabilization (all states go to zero):

$$u(k) = Fx(k) \tag{1.31}$$

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

$$\begin{aligned} x(k+1) &= Ax(k) + BFx(k) && (1.32)\\ &= (A+BF)x(k) && (1.33) \end{aligned}$$

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

$$x(k+1) = \alpha x(k) \tag{1.34}$$

Where $-1 < \alpha < 1$, then $\lim_{k\to\infty} x(k) = 0$. Taking this to a matrix-space preserves this intuition, only now instead of placing a scalar between -1 and 1, we now seek to place the eigen values, or poles, of the system within the unit circle of the complex plane. The regardless of any dynamics, a system will once again converge to zero. Poles placed at the boundary of the unit circle would denote an asymptotically stable system – like a ball at the top of a hill, it will be stable until some force comes along and pushes it. For any system defined by their A and B matrices, it is relevant to check if it is controllable. This is thankfully quite simple thanks to the Controllability Matrix. Defined as

$$\mathcal{C} = \left[A^{(n-1)}B, \ldots, AB, B\right] \tag{1.35}$$

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

If the Controllability Matrix $\mathcal{C}$ is full rank in rows, then the system is controllable (recall n is the number of states). What full row rank means is that no row of the matrix can be made by any linear combination of any of the other rows – in other words, they are all independent of one another.

1.4.1 Example — Basic Control with Pole Placement

There are a practically infinite number of controllers F that could then send our system to stability. The closer the poles 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 1.8. Placement is done using MATLAB's *place* function and then verified. Poles must always come in conjugate pairs, as visible in the reflection over the imaginary axis The poles, placed at $0.5 + -0.5i$ and $-0.7 + -0.1i$ result in controller F_{simple}

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

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

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

The same process can be repeated for u_2 and for any sample number k. The way to verbalize a controller like this is as a state-input-weight relation, where the states are the columns, the inputs the rows, and the weight the corresponding value. For example, for every unit away from control x_1 is (recall 'control' is a zero state), u_1 will generate -40,364 units of input. The net input will be the summed effects of each state. When the controller in Eq. 1.36 is applied to our dual-mass system, our system produces outputs seen in Figs. 1.9 and 1.10, under the inputs shown in Figs. 1.11 and 1.12

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

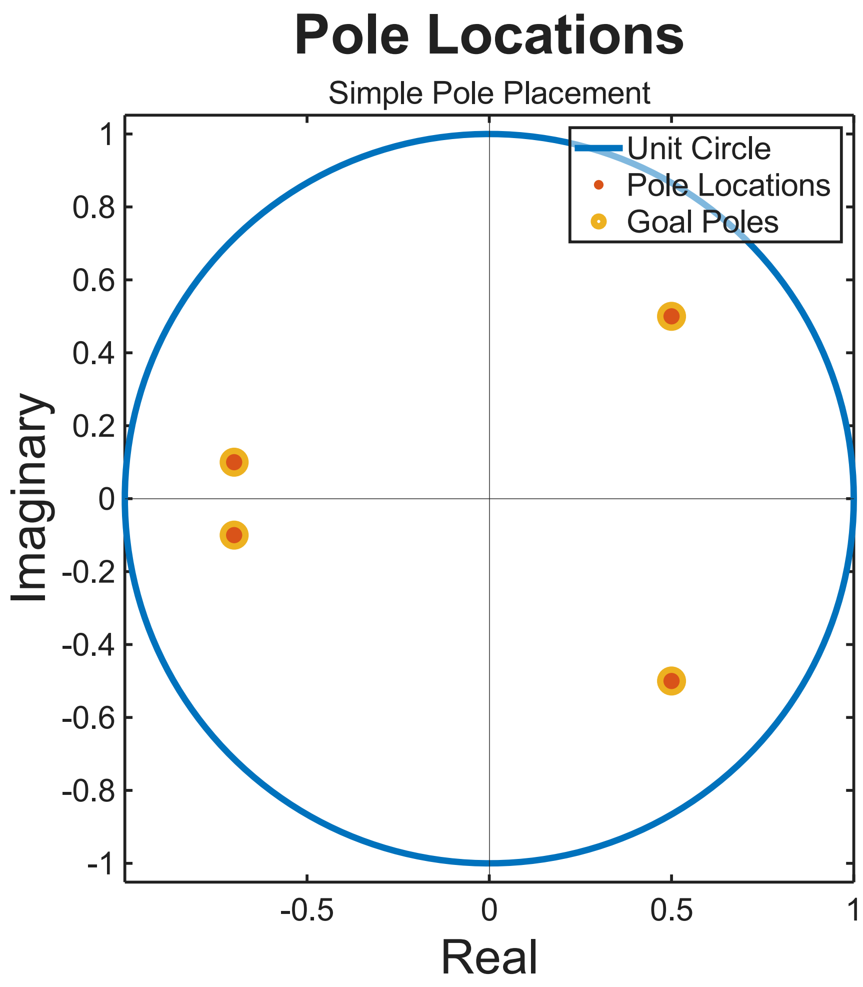

Figure 1.8: Simple Pole Placement Locations

can be taken even further with a special type of controller known as a 'dead-beat' controller. This is a controller which places the poles of a closed-loop system at the origin[2] and produces the time-optimal solution. Once again, the system poles are shown in Figure 1.13 and produces controller $F_{deadbeat}$

[2]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 ($\leq 1 \times 10^{-6}$)

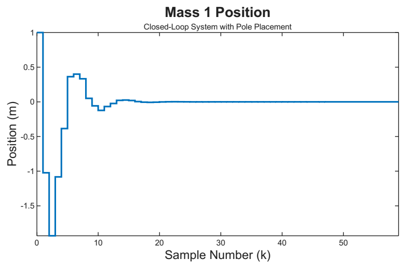

Figure 1.9: Mass 1 Position with a Simple Pole Placement Feedback Controller

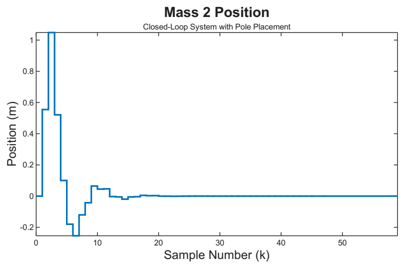

Figure 1.10: Mass 2 Position with a Simple Pole Placement Feedback Controller

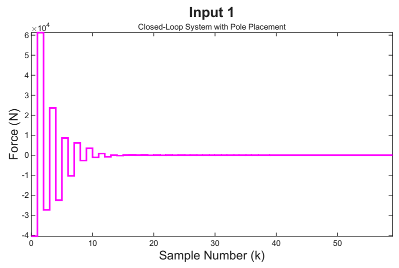

Figure 1.11: Input 1 Magnitude with a Simple Pole Placement Feedback Controller

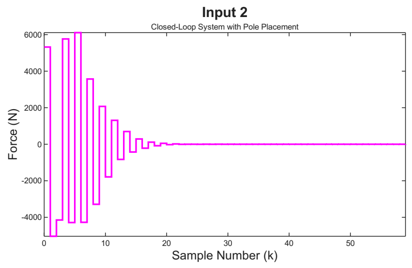

Figure 1.12: Input 2 Magnitude with a Simple Pole Placement Feedback Controller

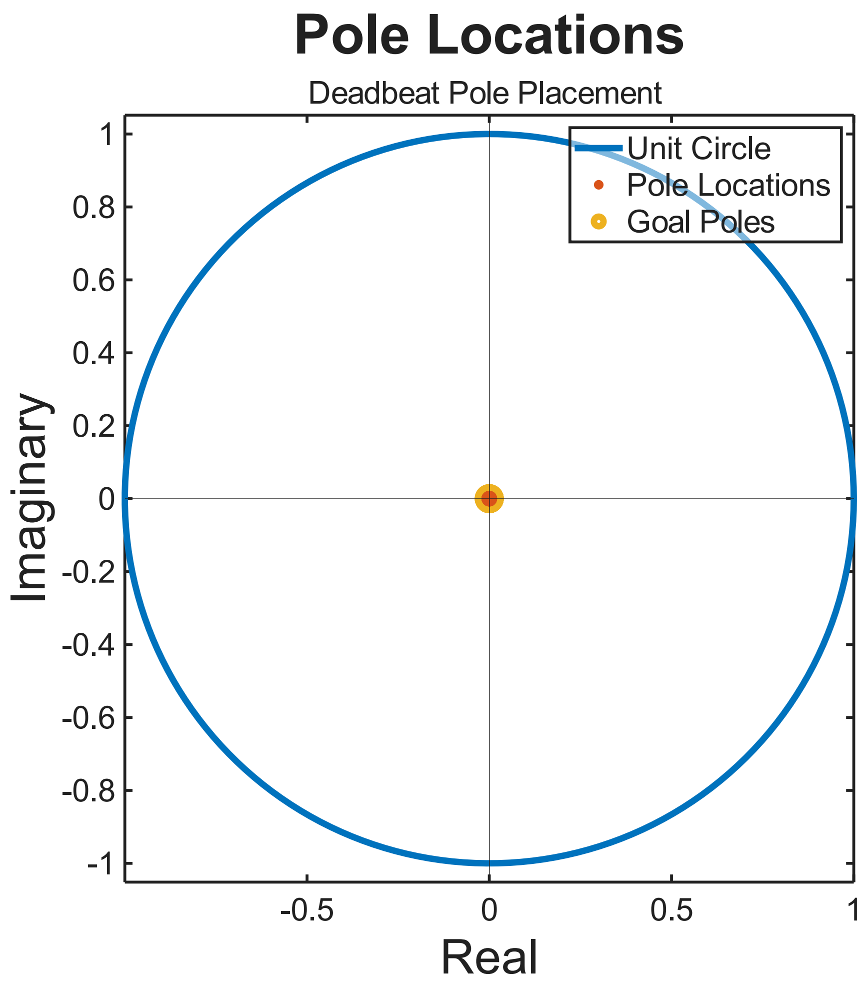

Figure 1.13: Deadbeat Pole Placement Locations

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

Its application results in the outputs and inputs denoted in Figures 1.14 - 1.17

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

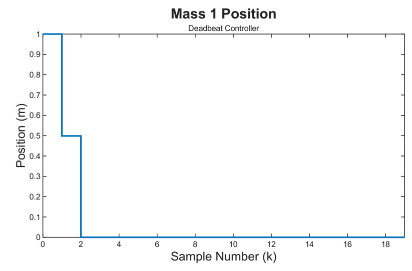

Figure 1.14: Mass 1 Position with a Deadbeat Controller

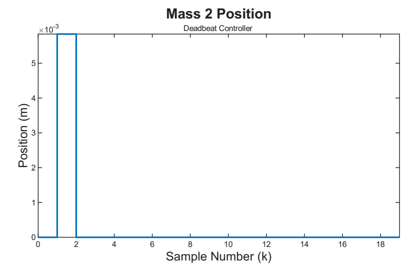

Figure 1.15: Mass 2 Position with a Deadbeat Controller

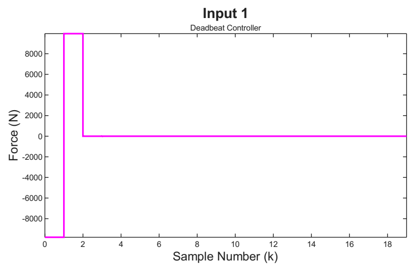

Figure 1.16: Input 1 Magnitude with a Deadbeat Controller

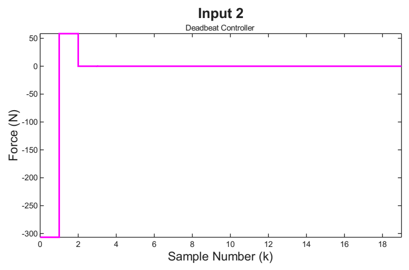

Figure 1.17: Input 2 Magnitude with a Deadbeat Controller

controller which can be tweaked more precisely to adhere to certain system limits.

1.5 Linear Quadratic Regulator Controller

The Linear Quadratic Regulator (LQR) Controller allows for the 'weighting' of system states and inputs. What this means is that a controller can be designed the shies away from extreme inputs, at the cost of uncontrolled states, or conversely will take extreme steps to keep a system under control. This is done by introducing three new variables: Q, R, and γ. Q is an $n \times n$ matrix which applies relative 'costs' (or rewards as desired) to states of the system. Similarly, R is a $r \times r$ matrix which weighs the inputs. γ is a scalar value between 0 and 1 which informs the cost function how much to discount the future versus the now (hence it is sometimes referred to as the 'discount factor'). Q and R are then used to define the cost function we wish to minimize. It is most common to define them as identity matrices with some associated weight, but they truly can be set as whatever so long as they are symmetric ($Q = Q^T$ and $R = R^T$). We will only use the identity approach. What each component of the cost matrices Q and R tells us is how much cost to attribute to a component of the state or input, respectively, being away from zero. This will all make more sense in this section's example. It is also important to note that weights are relative: a controller defined by $Q = 100 \cdot I_{n\times n}$ and $R = 1 \cdot I_{r\times r}$ will be the exact same as one defined by $Q = 200 \cdot I_{n\times n}$ and $R = 2 \cdot I_{r\times r}$ Each sample, k, we want to have a scalar cost as a function of our states and inputs. For a given time step, we generate a utility function $U(k)$

$$U(k) = u^T(k)\, R u(k) + x^T(k)\, Q\ x(k) \tag{1.39}$$

In our journey to control, we will work through multiple timesteps, each with their own $U(k)$, so in the whole process we will incur some net cost, J. The cost can be viewed as the summation of all these utilities along the way:

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

Bringing back the aforementioned discount factor γ, we modify the cost function such that we can adjust the time horizon of consequence. This presents us with our discounted cost function

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

Since $0 < \gamma \leq 1$, as k goes to ∞, the impact of the infinite horizon utility reduces to zero. In the LQR process, we are looking for a controller that minimizes the cost function, J. The next important idea is the Principal 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 Point A to Point C What this tells us is that no matter what step we are in 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:

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

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

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

1.5.1 Example — LQR

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

$$Q = 100 \cdot I_{4\times4} \quad R = 1 \cdot I_{2\times2} \quad \gamma = \ 0.8 \tag{1.44}$$

Using the attached function *discount_LQR*[3] will find the correct controller for the cost function (and discount factor) we use. Applying that function to our discrete system, we find

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

[3]MATLAB's dlqr function returns a different result than the one defined by our cost function as it does not have a discount parameter

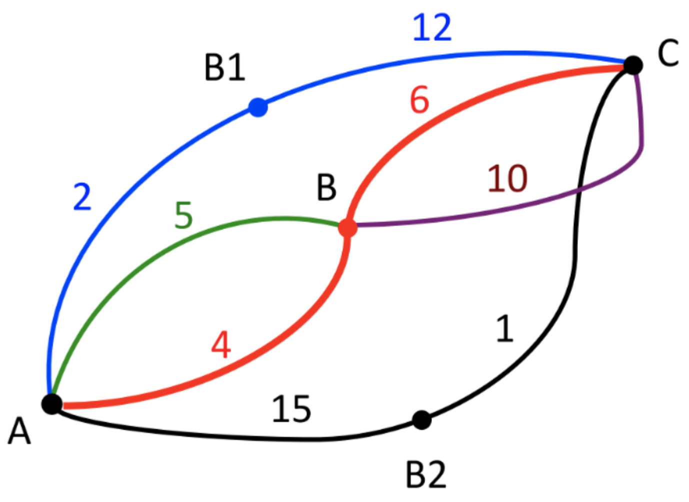

Figure 1.18: A two-step process: The red path from A to C through B is optimal (with minimum cost = 4 + 6 = 10). The principle of optimality states that if one starts from B then the optimal path to C must be the red path B-C. All other paths from B to C must cost more than 6, for example, the purple 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. (Phan **book** Chp8)

Examining where that place the poles (Figure 1.19) and the input-output data (Figures 1.20 - 1.23), we see that control takes almost two seconds, but inputs are magnitudes smaller than that seen by the pole placement controllers.

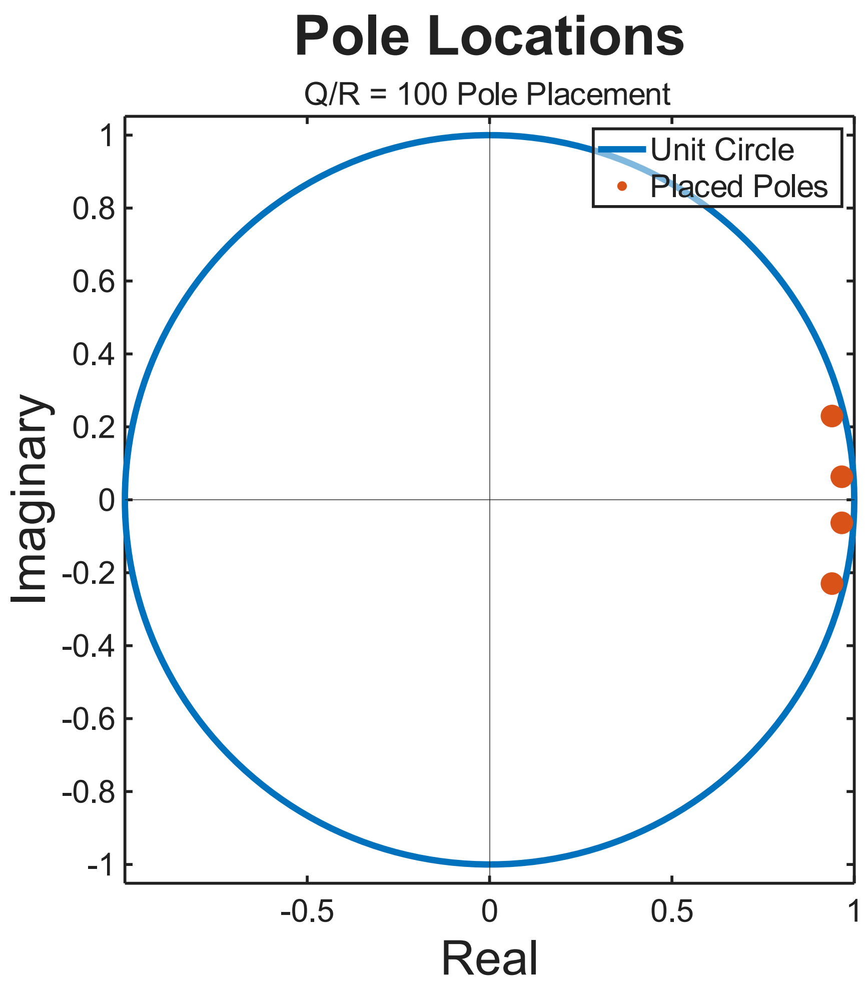

Figure 1.19: Pole Locations of a Q/R = 100 LQR Controller on our Spring-Mass System

If we were inclined to tweak the parameters, perhaps the inputs were beyond the capabilities of our actual system, we could easily do so. If one

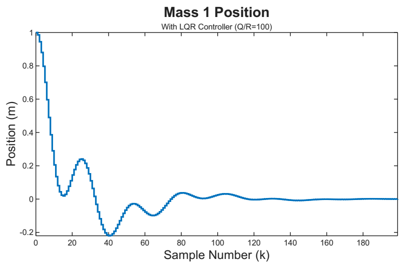

Figure 1.20: Mass 1 Position with a Q/R = 100 LQR Feedback Controller

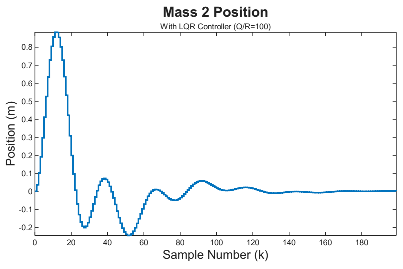

Figure 1.21: Mass 2 Position with a Q/R = 100 LQR Feedback Controller

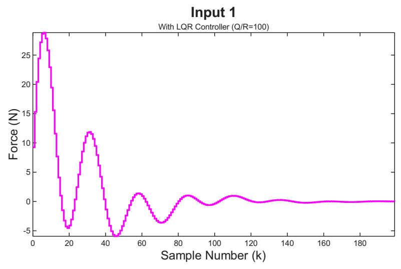

Figure 1.22: Input 1 Magnitude with a Q/R = 100 LQR Feedback Controller

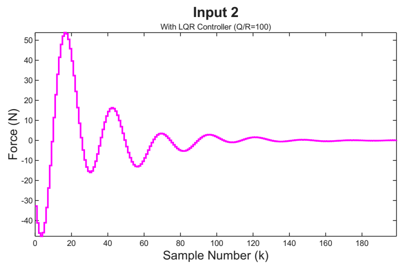

Figure 1.23: Input 2 Magnitude with a Q/R = 100 LQR Feedback Controller

were the scale R by a factor of ten (or Q by a factor of 0.1), then the following controller F_{bigR}, poles (Figure 1.24), and IO date (Figures 1.25 - 1.28) would result:

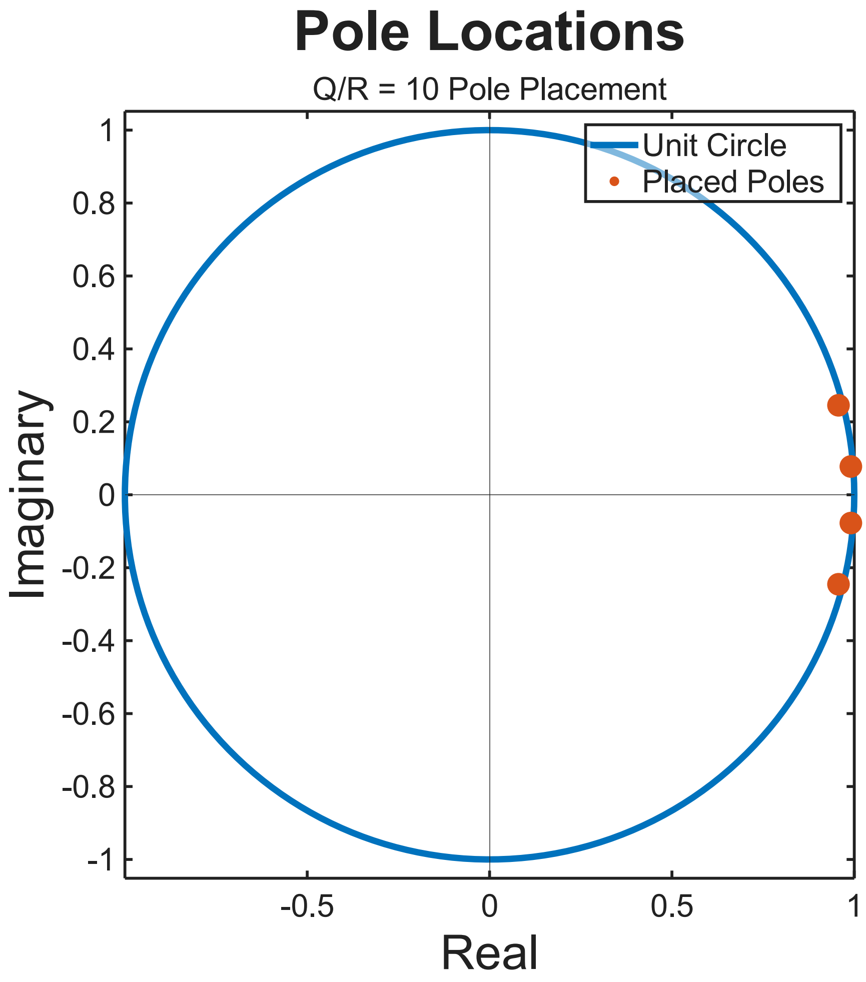

Figure 1.24: Pole Locations of a Q/R = 10 LQR Controller on our Spring-Mass System

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. Setting $R = 0_{2\times2}$ logically does place poles at the unit circle, but it also places two at

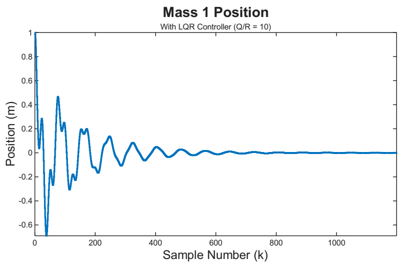

Figure 1.25: Mass 1 Position with a Q/R = 10 LQR Feedback Controller

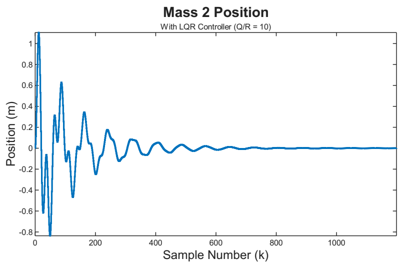

Figure 1.26: Mass 2 Position with a Q/R = 10 LQR Feedback Controller

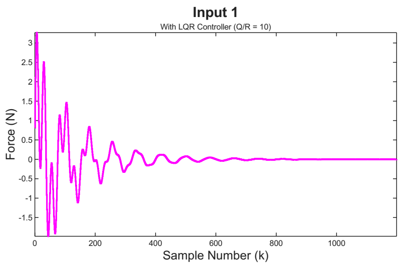

Figure 1.27: Input 1 Magnitude with a Q/R = 10 LQR Feedback Controller

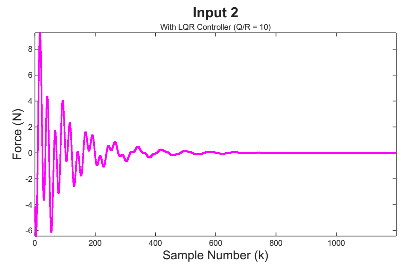

Figure 1.28: Input 2 Magnitude with a Q/R = 10 LQR Feedback Controller

the unit circle border. This is because a deadbeat controller leads to extreme velocities – a state we do not capture in the output but is factored into the cost. Without the presence of input weights, the controller solely focuses on states - all equally weighted in our case.

1.6 Iterative Learning Control

Switching gears, we will introduce now another form of system. The previous models have all been iterative in time (i.e. adjust next input based on last samples state), but there is a form of control that focuses on trials. This is logical for and applied in the manufacturing process, where the desired output is not a ‘zero’ state, but rather zero error. Iterative Learning Control (“ILC”) is particularly useful for its ability to factor out repeated noise and function to produce machined outputs regardless of initial condition or repeated disturbances.

Iterative Learning Control employes a system representation is expanded to factor in the temporal element of control steps. Instead of each step (k) trying to send the states to zero, we now want each trial (j) to send the error on our outputs to zero. The first step is to define our output, denoted as y (“y-bar”), occurring over p time steps. That is, y is a $pm \times 1$. Similarly, there is then a sequence of inputs, denoted as u (“u-bar”) that when applied, will get us here – it will be $pr \times 1$.

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

$$\underline{y} = P \cdot \underline{u} + \underline{d} \tag{1.46}$$

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

$$\underline{y} = \begin{bmatrix} y\,(1) \\ y\,(2) \\ \vdots \\ y\,(p) \end{bmatrix} \quad \underline{u} = \begin{bmatrix} u\,(0) \\ u\,(1) \\ \vdots \\ u\,(p-1) \end{bmatrix} \tag{1.47}$$

$$P = \begin{bmatrix} CB & D & 0 & 0 & 0 \\ CAB & CB & D & 0 & 0 \\ CA^2B & CAB & CB & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & D \\ CA^{p-1}B & CA^{p-2}B & CA^{p-3}B & \cdots & CB \end{bmatrix} \tag{1.48}$$

Note that in the ILC process we do not try to control $y\,(0)$ or even model it. We cannot control initial conditions and therefore do not worry about them. As ILC is the pursuit of a desired output, we will mark that as $\underline{y}^*$ and it stands that the input that gets us there is marked as $\underline{u}^*$. That is:

$$\underline{y}^* = P\underline{u}^* + \underline{d} \tag{1.49}$$

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

$$\delta_j x = x_j - x_{j-1} \tag{1.50}$$

Applying this δ operator to Eq. 1.46

$$\delta_j \underline{y} = P \cdot \delta_j \underline{u} + \delta_j \underline{d} \tag{1.51}$$

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

$$\delta_j \underline{y} = P \cdot \delta_j \underline{u} \tag{1.52}$$

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

$$e_j = \underline{y}^* - \underline{y}_j \tag{1.53}$$

Applying the δ operator to this equation

$$\delta_j e = \delta_j \underline{y}^* - \delta_j \underline{y} \tag{1.54}$$

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

$$\delta_j e = -\delta_j \underline{y} \tag{1.55}$$

which expands to

$$e_j - e_{j-1} = -\delta_j \underline{y} \tag{1.56}$$

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

$$e_{j+1} - e_j = -\delta_{j+1}\underline{y} \tag{1.57}$$

Through re-arrangement of Eq. 1.57 and substitution of Eq, 1.52, we arrive upon the ILC Equation

$$e_{j+1} = Ie_j - P\delta_{j+1}\underline{u} \tag{1.58}$$

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

$$\delta_{j+1}\underline{u} = \mathcal{L}e_j \tag{1.59}$$

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

1.6.1 Example — ILC

The first step in setting up an ILC problem is to establish the goal, or y^*. For simplicity, we will work through this example trying to draw a circle. That is, x_1 would ideally trace out one period of a cosine, and x_2 will follow one period of a sine wave. We can set the resolution of this circle by our choice of p. Supposed we set $p = 100$, meaning we want to draw a circle over p discrete time steps. We will define the goal for our first output (the position of x_1) as y_1^* and the second output (position of x_2) as y_2^*:

$$y_1^* = \cos\left(\frac{2\pi k}{p}\right) \quad y_2^* = \sin\left(\frac{2\pi k}{p}\right) \tag{1.60}$$

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

$$\underline{y}^* = \begin{bmatrix} y_1^*(1) \\ y_2^*(1) \\ \vdots \\ y_1^*(p) \\ y_2^*(p) \end{bmatrix} \tag{1.61}$$

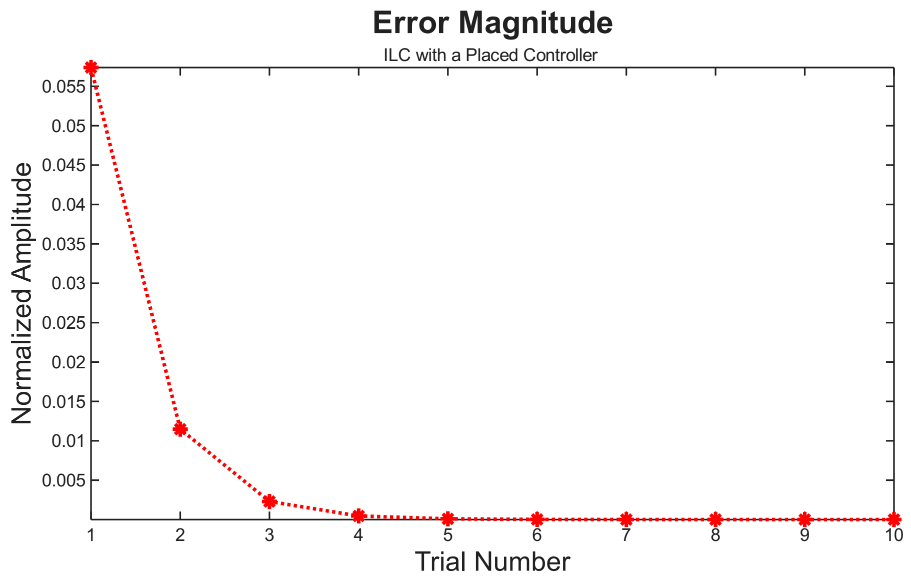

Figure 1.29: Simple ILC Controller Error Progression

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

$$\mathcal{L} = 0.8P^+ \tag{1.62}$$

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

Further showing the progression of the individual outputs and inputs (Figures 1.30 - 1.33) as well as the shaped output in Figure 1.34 (where x_1's position is the x-axis and x_2's position is the y-axis), you can see as the system 'learns' to draw the circle. Trial 1 matches our open-loop response, but even Trial 2 much more closely matches our goal (marked by the dotted red line). It is convenient here that the initial conditions of the system match those of the initial goal outputs, but we will shortly show that it is not necessary.

To demonstrate ILC's ability to learn arbitrary shapes, a $p = 500$ point

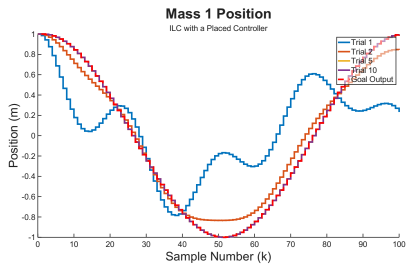

Figure 1.30: Mass 1 Position Sequence over select Trials

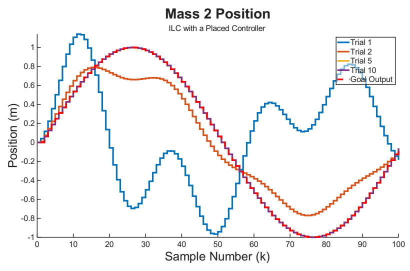

Figure 1.31: Mass 2 Position Sequence over select Trials

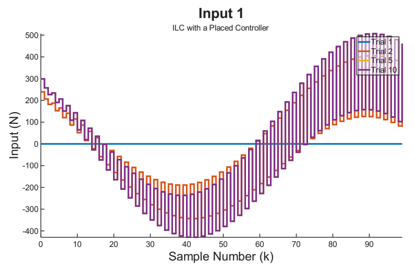

Figure 1.32: Input 1 Magnitude Sequence over select Trials

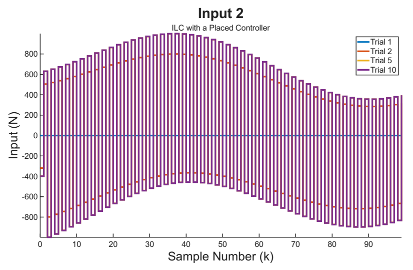

Figure 1.33: Input 2 Magnitude Sequence over select Trials

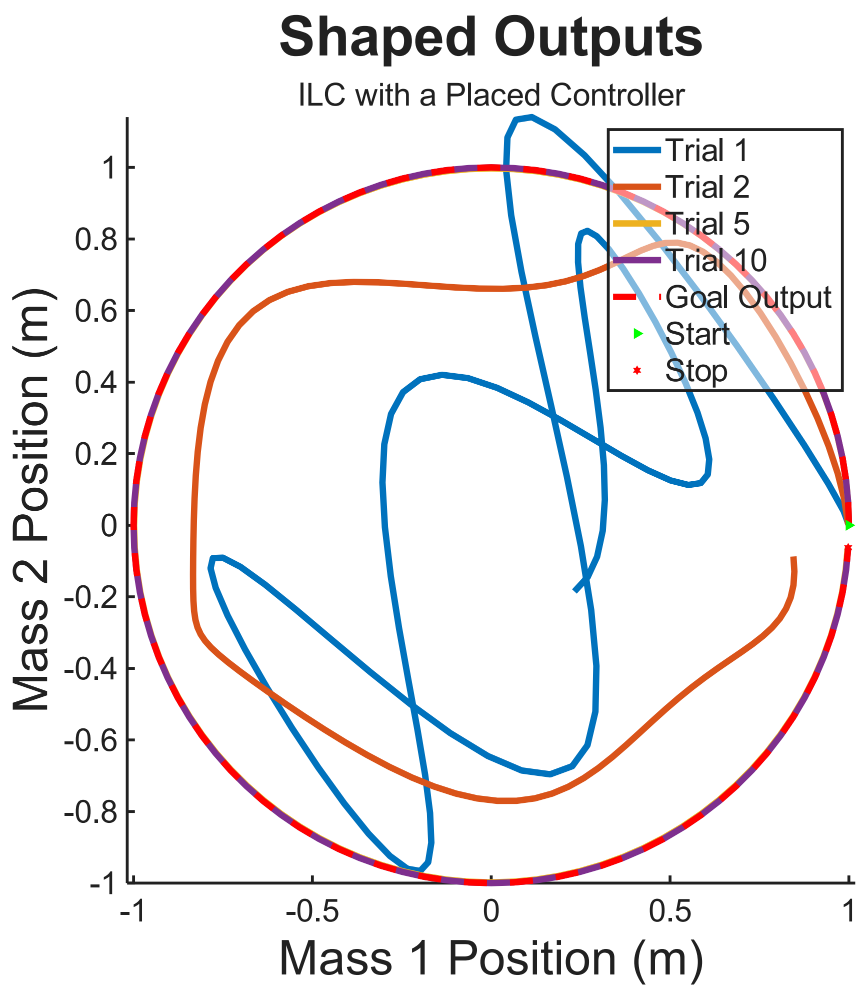

Figure 1.34: Simple ILC Controller Progression to draw a Circle

resolution, mouse-drawn version of the word 'Dartmouth' was introduced as the $\underline{y}^*$. Utilizing the same controller shown in Eq. 1.62, we can similarly learn the exact inputs required to generate our desired output. The sharp jump from the starting position (1, 0) is the point $(y_1(0), y_2(0))$ which we have previously mentioned we cannot control. As such, the $\underline{y}^*$ does not start at that point, but rather at the green 'play' symbol

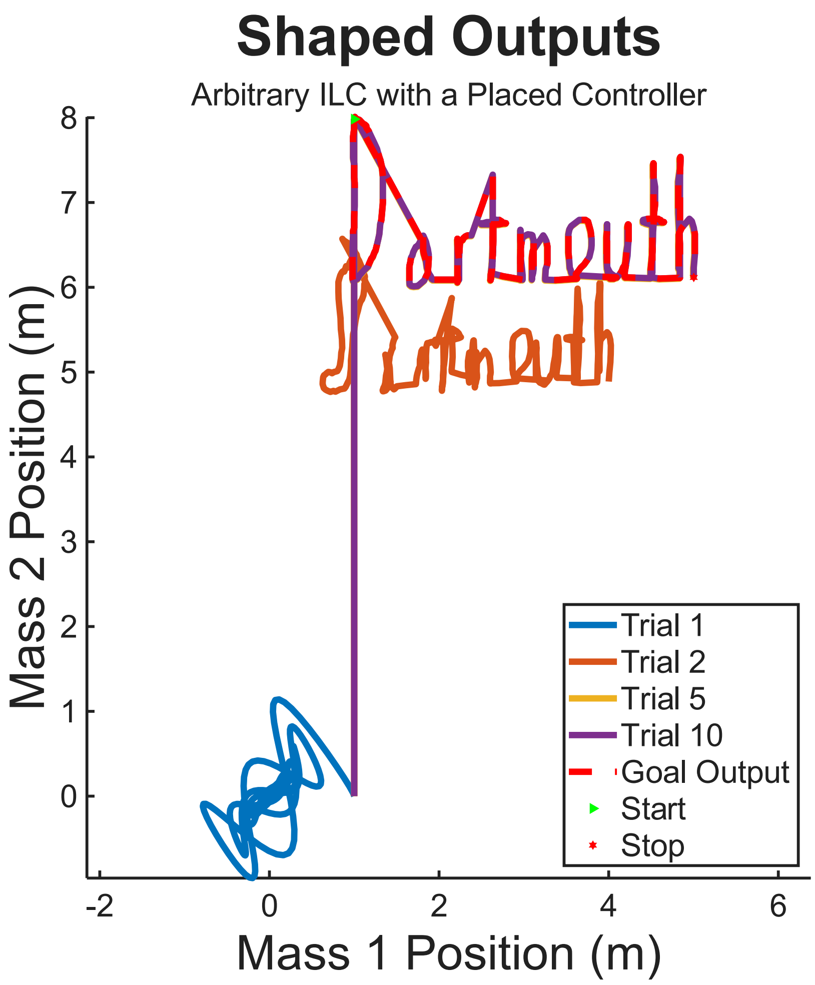

Figure 1.35: Simple ILC Controller learning 'Dartmouth'

1.7 Reinforcement Learning

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

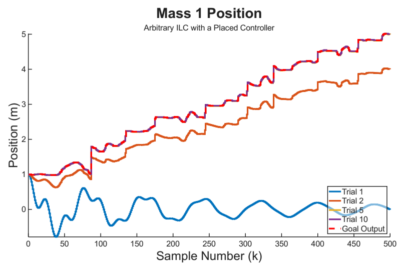

Figure 1.36: Mass 1 Position Sequence over select Trials learning ‘Dartmouth’

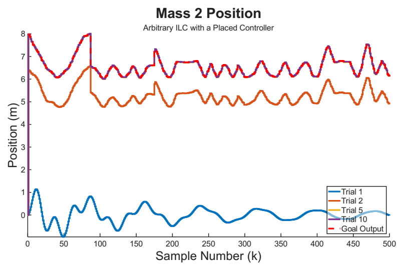

Figure 1.37: Mass 2 Position Sequence over select Trials learning ‘Dartmouth’

then create a cost-to-go function $V(k)$

$$V\left(k\right)=\sum_{i=0}^{s-1}\gamma^{i}U\left(k+1\right)=U\left(k\right)+\gamma U\left(k+1\right)+\cdots+\gamma^{s-1}U\left(k+s-1\right) \tag{1.63}$$

As from our principle of optimality, it similarly follows that whatever controller minimizes $V\left(k\right)$ will then also minimize J. This agrees with an important feature of the cost-to-go function, and that is recurrence. If we multiply Eq. 1.63 by γ and increment k by 1, we get

$$\gamma V\left(k+1\right)=\gamma U\left(k+1\right)+\cdots+\gamma^{s}U\left(k+s\right) \tag{1.64}$$

Substituting Eq. 1.64 into 1.63, the relationship between $V\left(k\right)$ and $V\left(k+1\right)$ can be shown

$$V\left(k\right)=\gamma V\left(k+1\right)+U\left(k\right)-\gamma^{s}U\left(k+s\right) \tag{1.65}$$

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

$$V\left(k\right)=\gamma V\left(k+1\right)+U\left(k\right) \tag{1.66}$$

It can then be useful to express $V\left(k\right)$ in a supervector format for later descriptions. Substituting the utility function described in Eq. 1.39 into Eq. 1.65

$$\begin{aligned}V\left(k\right)=\,&u^{T}\left(k\right)Ru\left(k\right)+x^{T}\left(k\right)Qx\left(k\right)\\&+\gamma u^{T}\left(k+1\right)Ru\left(k+1\right)+\gamma x^{T}\left(k+1\right)Qx\left(k+1\right)\\&+\cdots+\\&+\gamma^{s-1}u^{T}\left(k+s-1\right)Ru\left(k+s-1\right)\\&+\gamma^{s-1}x^{T}\left(k+s-1\right)Qx\left(k+s-1\right)\end{aligned} \tag{1.67}$$

And we can define supervectors for state and input histories

$$x_s(k) = \begin{bmatrix} x(k) \\ x(k+1) \\ \vdots \\ x(k+s-1) \end{bmatrix} \quad u_s(k) = \begin{bmatrix} u(k) \\ u(k+1) \\ \vdots \\ u(k+s-1) \end{bmatrix} \tag{1.68}$$

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

$$\mathbf{Q} = \begin{bmatrix} Q_{n\times n} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & Q_{n\times n} \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} R_{r\times r} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & R_{r\times r} \end{bmatrix} \tag{1.69}$$

and the Λ matrices are defined as

$$\Lambda_n = \begin{bmatrix} I_{n\times n} & & & \\ & \sqrt{\gamma} I_{n\times n} & & \\ & & \ddots & \\ & & & \left(\sqrt{\gamma}\right)^{s-1} I_{n\times n} \end{bmatrix} \tag{1.70}$$

$$\Lambda_r = \begin{bmatrix} I_{r\times r} & & & \\ & \sqrt{\gamma} I_{r\times r} & & \\ & & \ddots & \\ & & & \left(\sqrt{\gamma}\right)^{s-1} I_{r\times r} \end{bmatrix} \tag{1.71}$$

Now Eq. 1.67 can be re-written as

$$V(k) = u_s^T(k)\, \mathbf{R}_\gamma u_s(k) + x_s^T(k)\, \mathbf{Q}_\gamma x_s(k) \tag{1.72}$$

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

$$x_s(k) = P_1 x(k) + P_2 u_s(k) \tag{1.73}$$

where

$$P_1 = \begin{bmatrix} I \\ A \\ \vdots \\ A^{s-1} \end{bmatrix} \quad P_2 = \begin{bmatrix} 0 & & & \\ B & 0 & & \\ \vdots & \ddots & \ddots & \\ A^{s-2}B & \cdots & B & 0 \end{bmatrix} \tag{1.74}$$

Substituting these into our cost-to-go expression of Eq. 1.72, we get a cost-to-go defined only in terms of present state and all inputs from now until s

$$V(k) = u_s^T(k)\,\mathbf{R}_\gamma u_s(k) + \left[P_1 x(k) + P_2 u_s(k)\right]^T \mathbf{Q}_\gamma \left[P_1 x(k) + P_2 u_s(k)\right] \tag{1.75}$$

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

$$\mathbf{S} \;=\; \begin{bmatrix} P_1^T \mathbf{Q}_\gamma P_1 & P_1^T \mathbf{Q}_\gamma P_2 \\ P_2^T \mathbf{Q}_\gamma P_1 & \mathbf{R}_\gamma + P_2^T \mathbf{Q}_\gamma P_2 \end{bmatrix} \tag{1.76}$$

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

$$V(k) = \begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix}^T \mathbf{S} \begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix} \tag{1.77}$$

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

$$u_s(k) = \begin{bmatrix} u(k) \\ u_{s-1}(k+1) \end{bmatrix} \quad u_{s-1}(k+1) = \begin{bmatrix} u(k+1) \\ u(k+2) \\ \vdots \\ u(k+s-1) \end{bmatrix} \tag{1.78}$$

Given our system model and control law, all future inputs can be tracked back to $x(k)$ and $u(k)$. By repeated substitution, it can be shown

$$\begin{aligned} u(k+1) &= Fx(k+1) = F\left[Ax(k) + Bu(k)\right] \\ u(k+2) &= Fx(k+2) = F(A+BF)\left[Ax(k) + Bu(k)\right] \\ &\vdots \\ u(k+s-1) &= F(A+BF)^{s-2}\left[Ax(k) + Bu(k)\right] \end{aligned} \tag{1.79}$$

We can now rewrite 1.78

$$u_{s-1}(k+1) = F_{x}x(k) + F_{u}u(k) \tag{1.80}$$

where

$$F_x = \begin{bmatrix} FA \\ F(A+BF)A \\ \vdots \\ F(A+BF)^{s-2}A \end{bmatrix} \qquad F_u = \begin{bmatrix} FB \\ F(A+BF)B \\ \vdots \\ F(A+BF)^{s-2}B \end{bmatrix} \tag{1.81}$$

Then

$$\begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix} = \begin{bmatrix} x(k) \\ u(k) \\ u_{s-1}(k+1) \end{bmatrix} = \begin{bmatrix} I_{n\times n} & 0_{n\times r} \\ 0_{r\times n} & I_{r\times r} \\ F_x & F_u \end{bmatrix} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \tag{1.82}$$

Definining **R**

$$\mathbf{R} = \begin{bmatrix} I_{n\times n} & 0_{n\times r} \\ 0_{r\times n} & I_{r\times r} \\ F_x & F_u \end{bmatrix} \tag{1.83}$$

We can now write

$$\begin{bmatrix} x(k) \\ u_s(k) \end{bmatrix} = \mathbf{R} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \tag{1.84}$$

If we create matrix[4] P $= \mathbf{R}^T\mathbf{S}\mathbf{R}$, we can substitute Eq. 1.84 into Eq. 1.77 to get a new cost-to-go function

$$Q(x(k), u(k)) = \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \mathbf{P} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \tag{1.85}$$

We may now utilize this formulation, along with the recurrence equation, to find the optimal Q-function. As the Q-function is defined for a given controller F, and we constructed $Q(x(k), u(k))$ in our observance with the cost function that defined our LQR controller, the Q-function will be optimized by the LQR controller. The optimal controller F will produce $u(k+1)$ that minimizes our cost-to-go, now defined in Eq. 1.85, starting from $x(k)$. That is, it will give us input $u(k+1)$ defined as:

$$u(k+1) \;=\; \arg\min_{u(k+1)} Q(x(k+1),\; u(k+1)) \tag{1.86}$$

[4]Note this is not the same P matrix defined in the ILC problem

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

$$Q(x(k+1), \arg\min_{u(k+1)} Q(x(k+1),\ u(k+1))) = \min_{u(k+1)} Q(x(k+1),\ u(k+1)) \tag{1.87}$$

So we can now write the recurrence equation for the deterministic Q-Learning-based RL method. By taking Eq. 1.66, substituting in our new cost-to-go defined in Eq. 1.85 and the logic in Eq. 1.87, we arrive upon the relationship that the optimal Q-function (as defined by the optimal controller) must satisfy:

$$Q\left(x\left(k\right),u\left(k\right)\right) = \gamma \min_{u(k+1)} Q\left(x\left(k+1\right),u\left(k+1\right)\right) + U\left(k\right) \tag{1.88}$$

Any Q-function will satisfy

$$Q\left(x\left(k\right),u\left(k\right)\right) = \gamma Q\left(x\left(k+1\right),u\left(k+1\right)\right) + U\left(k\right) \tag{1.89}$$

but only Eq. 1.88 is satisfied by the optimal Q-function, as defined by the optimal controller F

To extract the controller from a given Q-function, we return to Eq. 1.85. Recall matrix $\mathbf{P}$ is symmetric. We can re-write it as

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{xx} & \mathbf{P}_{xu} \\ {\mathbf{P}_{xu}}^T & \mathbf{P}_{uu} \end{bmatrix} \tag{1.90}$$

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

$$F = -\left(\mathbf{P}_{uu}\right)^{-1}\left(\mathbf{P}_{xu}^T\right) \tag{1.91}$$

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

1.7.1 Policy Iteration

The first method which we will demonstrate is that of Policy Iteration. To do this, we need data triplets of $x(k), u(k)$ and $x(k+1)$. By manipulating enough of these triplets, we can episodically solve for the $\mathbf{P}$ that parametrizes a Q-function, while simultaneously optimizing the Q-function. We begin with Eq. 1.89, but re-arrange it to

$$Q(x(k), u(k)) \; - \; \gamma Q(x(k+1), u(k+1)) = U(k) \tag{1.92}$$

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

$$H \; = \; \begin{bmatrix} h_1 & h_2 & \cdots & h_w \end{bmatrix} \tag{1.93}$$

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

$$H^S = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_w \end{bmatrix} \tag{1.94}$$

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

$$A \; = \; \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} \tag{1.95}$$

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

$$A \otimes B \; = \; \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \tag{1.96}$$

Now to see how these operators will be useful, we return to Eq. 1.85, re-written below

$$Q\left(x\left(k\right),u\left(k\right)\right)=\begin{bmatrix}x\left(k\right)\\u\left(k\right)\end{bmatrix}^{T}\mathbf{P}\begin{bmatrix}x\left(k\right)\\u\left(k\right)\end{bmatrix}\tag{1.85}$$

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

$$x\left(k\right)=x_1\quad u\left(k\right)=u_1\quad \mathbf{P}=\begin{bmatrix}\mathbf{P}_{11}&\mathbf{P}_{12}\\\mathbf{P}_{21}&\mathbf{P}_{22}\end{bmatrix}\tag{1.97}$$

Manually working out Eq. 1.85, we can re-write it as

$$\begin{aligned}Q\left(x\left(k\right),u\left(k\right)\right)&=\begin{bmatrix}x_1&u_1\end{bmatrix}\begin{bmatrix}\mathbf{P}_{11}&\mathbf{P}_{12}\\\mathbf{P}_{21}&\mathbf{P}_{22}\end{bmatrix}\begin{bmatrix}x_1\\u_1\end{bmatrix}\\&=\begin{bmatrix}x_1\mathbf{P}_{11}+u_1\mathbf{P}_{21}&x_1\mathbf{P}_{12}+u_1\mathbf{P}_{22}\end{bmatrix}\begin{bmatrix}x_1\\u_1\end{bmatrix}\\&=x_1^2\mathbf{P}_{11}+x_1u_1\mathbf{P}_{21}+x_1u_1\mathbf{P}_{12}+u_1^2\mathbf{P}_{22}\end{aligned}\tag{1.98}$$

It can be shown that the results of Eq. 1.98 can then be expressed by stacking $\mathbf{P}$ as

$$Q\left(x\left(k\right),u\left(k\right)\right)=\begin{bmatrix}x_1^2&x_1u_1&x_1u_1&u_1^2\end{bmatrix}\begin{bmatrix}\mathbf{P}_{11}\\\mathbf{P}_{21}\\\mathbf{P}_{12}\\\mathbf{P}_{22}\end{bmatrix}\tag{1.99}$$

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

$$\begin{aligned}\begin{bmatrix}x_1^2&x_1u_1&x_1u_1&u_1^2\end{bmatrix}&=\begin{bmatrix}x_1&u_1\end{bmatrix}\otimes\begin{bmatrix}x_1&u_1\end{bmatrix}\\&=\begin{bmatrix}x_1\\u_1\end{bmatrix}^{T}\otimes\begin{bmatrix}x_1\\u_1\end{bmatrix}^{T}\end{aligned}\tag{1.100}$$

Putting it all together, we can re-write Eq. 1.98 as

$$Q\left(x\left(k\right),u\left(k\right)\right)=\begin{bmatrix}\begin{bmatrix}x_1\\u_1\end{bmatrix}^{T}\otimes\begin{bmatrix}x_1\\u_1\end{bmatrix}^{T}\end{bmatrix}\mathbf{P}^{S}\tag{1.101}$$

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

$$Q\left(x\left(k\right), u\left(k\right)\right) = \left[\begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix}^T \otimes \begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix}^T\right] \mathbf{P}^S \tag{1.102}$$

For controller F_j, we have a given Q-function parametrized by $\mathbf{P}_j$. Recall that $x\left(k\right)$ and $u\left(k\right)$ can be arbitrary[5], $x\left(k+1\right)$ will be produced by nature/the system, and all inputs from $k+1$ onward are defined by our control law. Thus we can re-write Eq. 1.92 as

$$\left[\begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix}^T \otimes \begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix}^T - \gamma \begin{bmatrix} x\left(k+1\right) \\ F_j x\left(k+1\right) \end{bmatrix}^T \otimes \begin{bmatrix} x\left(k+1\right) \\ F_j x\left(k+1\right) \end{bmatrix}^T\right] \mathbf{P}_j^S = U\left(k\right) \tag{1.103}$$

To simplify equations, write $X_j\left(k\right)$

$$X_j\left(k\right) = \begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix}^T \otimes \begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix}^T - \gamma \begin{bmatrix} x\left(k+1\right) \\ F_j x\left(k+1\right) \end{bmatrix}^T \otimes \begin{bmatrix} x\left(k+1\right) \\ F_j x\left(k+1\right) \end{bmatrix}^T \tag{1.104}$$

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

$$\begin{bmatrix} X_j\left(k\right) \\ X_j\left(k'\right) \\ X_j\left(k''\right) \\ \vdots \end{bmatrix} \mathbf{P}_j^S = \begin{bmatrix} U\left(k\right) \\ U\left(k'\right) \\ U\left(k''\right) \\ \vdots \end{bmatrix} \tag{1.105}$$

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

$$\mathbf{P}_j^S = \begin{bmatrix} X_j\left(k\right) \\ X_j\left(k'\right) \\ X_j\left(k''\right) \\ \vdots \end{bmatrix}^+ \begin{bmatrix} U\left(k\right) \\ U\left(k'\right) \\ U\left(k''\right) \\ \vdots \end{bmatrix} \tag{1.106}$$

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

In the noise free scenario, we need $(n+r)^2$ collections to solve for $\mathbf{P}_j^S$. Anything less, our matrix will be poorly conditioned and the pseudo-inverse operator will have more than one solution – unlikely to be optimal. By unstacking $\mathbf{P}_j^S$, we can update controller F_j using Eq. 1.91 such that iteratively

$$F_{j+1} = -(\mathbf{P}_{uu})^{-1}\left(\mathbf{P}_{xu}^T\right) \tag{1.107}$$

Since Eq. 1.103 is derived from the optimal condition outlined in Eq. 1.88, the controller we derived must also be optimal. It may take several iterations, but this process will alternate updating $\mathbf{P}$ and F until the optimal controller is found.

Example — Policy Iteration

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

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

Our system has four states ($n = 4$) and two inputs ($r = 2$). As such, each $X_j(k)$ will be 1×36 ($(n+r)^2 = (4+2)^2$). That means that each controller 'update' marked by the process shown in Eq. 1.106 requires 36 triplets of state, input, and next state data $(x(k), u(k), x(k+1))$. Knowing this, we will set parameters dictating the number of controllers we will try and the number of data points we will collect per controller. We must also define a controller to start with, which we will set to be all zeros. For every sample k, we first compute our $u(k)$. When learning, it is not enough to just use our classic $u(k) = Fx(k)$, we must add some random excitation term for exploration. That is

$$u(k) = Fx(k) + v(k) \tag{1.108}$$

Where $v(k)$ is some random value intentionally added to the classic $u(k)$. Terminology-wise, this differentiates it from noise - which we would not know the value of. Due to the arbitrary nature of the triplets, it is also possible to make the input purely random and not based in any way on the current state. The input under that approach would be

$$u(k) = v(k) \tag{1.109}$$

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

Whichever input approach we chose, we then apply it to the system in state $x(k)$. Nature then produces $x(k+1)$ for us. We now have our $x(k)$,$u(k)$,$x(k+1)$ triplet. Following Eq. 1.104 we formulate $X_j(k)$. Note that the $F_j x(k+1)$ term does not include an exploration term. At the same time, we will compute the utility $U(k)$ as defined in Eq. 1.39

For our system, we will repeat this process 35 more times before we can update the controller once. We compute $\mathbf{P}_j^S$ from Eq. 1.106, and undo the stack operator by reshaping it into a 6×6 matrix. The way in which we do this does not matter, (rows to column or column to rows) as $\mathbf{P}_j$ is symmetric. Numerical operators are not exact, however, and we can accelerate the learning process by imposing symmetry. That is after computing a P_j which is semi-symmetric, we set $\mathbf{P}_j$ as

$$\mathbf{P}_j = \frac{1}{2}\left(P_j + P_j^T\right) \tag{1.110}$$

Now we refer to Eq. 1.91 to extract the components to solve for our next controller as shown in Eq. 1.107. In this example, we grab the bottom right 2×2 block of $\mathbf{P}_j$ as $\mathbf{P}_{uu}$, and the bottom left 2×4 block as $\mathbf{P}_{xu}^T$. We then use those parameters to update our controller, and repeat the process. After you have iterated through all the controller you wish to learn, it is often useful to run out a few trials without an exploration on the input to verify to yourself that your controller does indeed work. The system will stabilize as you go under this approach, but only so much when the input is distorted. In this case, after five controllers (of 36 trials each) we produce the controller F_{policy}

$$F_{policy} = \begin{bmatrix} 9.9546 & -1.8952 & -2.6156 & -0.3501 \\ -25.2185 & 29.3267 & -0.8026 & -3.767 \end{bmatrix} \tag{1.111}$$

which matches our F_{LQR}^{γ} exactly to atleast 4 decimal places. Its application can be seen in Figures 1.38 - 1.41 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 1.42 and 1.43 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.

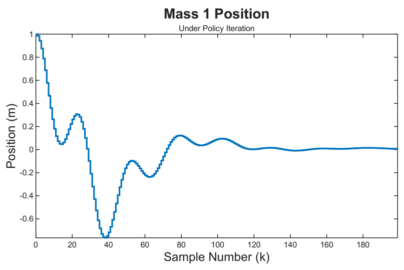

Figure 1.38: Mass 1 Position through Policy Iteration Trials

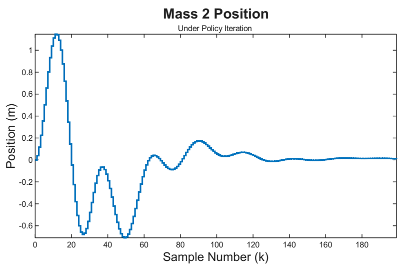

Figure 1.39: Mass 2 Position through Policy Iteration Trials

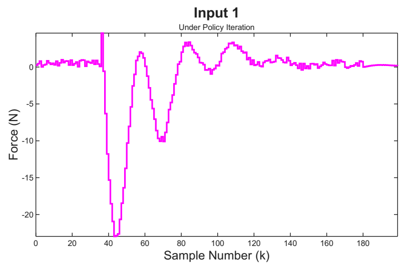

Figure 1.40: Input 1 Magnitude through Policy Iteration Trials

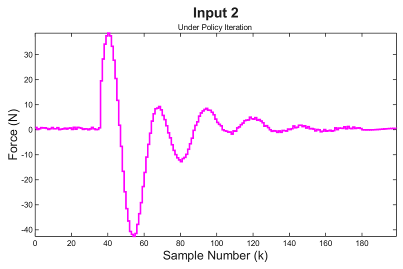

Figure 1.41: Input 2 Magnitude through Policy Iteration Trials

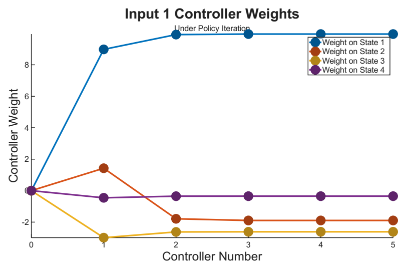

Figure 1.42: Input 1 Controller Weights through Policy Iteration Trials

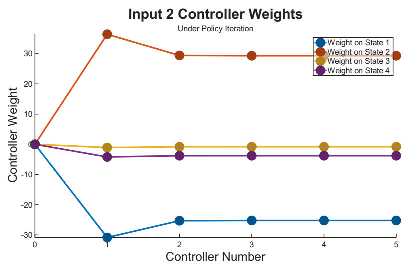

Figure 1.43: Input 2 Controller Weights through Policy Iteration Trials

1.7.2 Input Decoupling

In Policy Iteration, we learn every input at a time which has an exponential relationship with the number of trials we must complete. Input Decoupling allows us to learn one input at a time, reducing the number of collections per controller from $(n+r)^2$ to $(n+1)^2$, but at the cost of needing to complete r times as many learning trials. It can be easily shown that when $n^2 \geq r$ Policy Iteration learns faster / in less trials. Input Decoupling will always reduce the number of trials needed for one input to learn, but rarely the whole controller. However, no learning-optimality is lost, and it is often that control on one input sends all states to zero (though at a sub-optimal rate). Recall our cost-to-go function (as defined in Eq. 1.85)

$$Q\left(x\left(k\right), u\left(k\right)\right) = \begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix}^T \mathbf{P} \begin{bmatrix} x\left(k\right) \\ u\left(k\right) \end{bmatrix} \tag{1.85}$$

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

$$u\left(k\right) = \begin{bmatrix} u_1\left(k\right) \\ u_2\left(k\right) \\ \vdots \\ u_r\left(k\right) \end{bmatrix} \tag{1.112}$$

Which can be produced by our similarly-represented stacked controller

$$F = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_r \end{bmatrix} \tag{1.113}$$

Where each F_i is a$1 \times n$ vector. In general, for input $u_i\left(k\right)$, we can re-write Eq. 1.112 and 1.113 as

$$u(k) = \begin{bmatrix} u_{Ti}(k) \\ u_i(k) \\ u_{Bi}(k) \end{bmatrix} \tag{1.114}$$

$$F = \begin{bmatrix} F_{Ti} \\ F_i \\ F_{Bi} \end{bmatrix} \tag{1.115}$$

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

$$\mathbf{G}_i = \begin{bmatrix} I_{n\times n} & 0_{n\times 1} \\ F_{Ti} & 0_{t\times 1} \\ 0 & 1 \\ F_{Bi} & 0_{b\times 1} \end{bmatrix} \tag{1.116}$$

We can write the state-input stack as:

$$\begin{aligned} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} &= \begin{bmatrix} x(k) \\ u_{Ti}(k) \\ u_i(k) \\ u_{Bi}(k) \end{bmatrix} \\ &= \begin{bmatrix} I_{n\times n} & 0_{n\times 1} \\ F_{Ti} & 0_{t\times 1} \\ 0 & 1 \\ F_{Bi} & 0_{b\times 1} \end{bmatrix} \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix} \\ &= \mathbf{G}_i \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix} \end{aligned} \tag{1.117}$$

By defining a $\mathbf{P}$ akin to the one from Eq. 1.90

$$\mathbf{P}_i = \mathbf{G}_i^T \mathbf{P} \mathbf{G}_i \tag{1.118}$$

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

$$Q_i(x(k), u_i(k)) = \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix}^T \mathbf{P}_i \begin{bmatrix} x(k) \\ u_i(k) \end{bmatrix} \tag{1.119}$$

Once again we have a function of current state $x(k)$, but now the only concern is a single input variable $u_i(k)$. In our previous Q-function the controller that was built in was $r \times n$, now we are looking for a $1 \times n$ controller F_i. Just as we did for Policy Iteration, we now define a recurrence equation like that seen in Eq. 1.89.

$$Q_i(x(k), u_i(k)) = \gamma Q_i(x(k+1), u_i(k+1)) + U(k) \tag{1.120}$$

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

$$Q_i(x(k), u_i(k)) = \gamma \min_{u_{i(k+1)}} Q_i(x(k+1), u_i(k+1)) + U_i(k) \tag{1.121}$$

The analogies to Policy Iteration continue where instead of Eq. 1.85 we now have

$$Q_i\left(x\left(k\right),u_i\left(k\right)\right)=\begin{bmatrix}x\left(k\right)\\ u_{i(k)}\end{bmatrix}^T\mathbf{P}_i\begin{bmatrix}x\left(k\right)\\ u_{i(k)}\end{bmatrix}\tag{1.122}$$

and instead of Eq. 1.90

$$\mathbf{P_i}=\begin{bmatrix}\mathbf{P_i}_{xx} & \mathbf{P_i}_{xu}\\ \mathbf{P_i}_{xu}{}^T & \mathbf{P_i}_{uu}\end{bmatrix}\tag{1.123}$$

Where the F_i assosciated with Q_i is captured as it is in Eq. 1.91

$$F_i=-\left(\mathbf{P_i}_{uu}\right)^{-1}\left(\mathbf{P}_{i_{xu}}{}^T\right)\tag{1.124}$$

A similar stacking computation as seen in Eq. 1.105 can be done, except now we use X_{i_j}, defined as

$$X_{i_j}(k)=\begin{bmatrix}x\left(k\right)\\ u_i\left(k\right)\end{bmatrix}^T\otimes\begin{bmatrix}x\left(k\right)\\ u_i\left(k\right)\end{bmatrix}^T-\gamma\begin{bmatrix}x\left(k+1\right)\\ F_{i_j}x\left(k+1\right)\end{bmatrix}^T\otimes\begin{bmatrix}x\left(k+1\right)\\ F_{i_j}x\left(k+1\right)\end{bmatrix}^T\tag{1.125}$$

Note that we still compute $U\left(k\right)$ in the complete form, using all the inputs on the system not just the current one of interest (u_i). We can solve for $\mathbf{P}_i$ in the exact same manner as we do in Eq. 1.107, with the iterative equation

$$F_{i_{j+1}}=-\left(\mathbf{P}_{i_{j_{uu}}}\right)^{-1}\left(\mathbf{P}^T_{i_{j_{xu}}}\right)\tag{1.126}$$

Example — Input Decoupling

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

$$F^{\gamma}_{LQR}=\begin{bmatrix}9.9546 & -1.8952 & -2.6156 & -0.3501\\ -25.2185 & 29.3267 & -0.8026 & -3.767\end{bmatrix}\tag{1.45}$$

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

$$u_i(k)=F_ix\left(k\right)+v\left(k\right)\tag{1.127}$$

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

$$u_i(k) = v(k) \tag{1.128}$$

We then apply it to the to produce $x(k+1)$. With our $u_i(k), x(k)$, $x(k+1)$ triplet. Following Eq. 1.125 we formulate $X_{i_j}(k)$. At the same time, we will compute the utility $U(k)$ as defined in Eq. 1.39 For our system, we will repeat this process 24 more times before we can update the controller once. We compute $\mathbf{P}_{i_j}^S$ from Eq. 1.106 (using $X_{i_j}(k)$ in place of $X_j(k)$), and undo the stack operator by reshaping it into a 5×5 matrix. Symmetry is imposed once again.

Refer to Eq. 1.123 to extract the components to solve for our next controller as shown in Eq. 1.126. In input decoupling, we grab the bottom right scalar of $\mathbf{P}_{i_j}$ as $\mathbf{P}_{i_{uu}}$, and the bottom left 1×4 block as ${\mathbf{P}_{i_{xu}}}^T$. We then use those parameters to update our controller and repeat the process. In this case, after five controllers (of 25 trials each) we produce the controllers seen in Eq. 1.129

$$\begin{aligned} F_1 &= \begin{bmatrix} 9.9546 & -1.8952 & -2.6156 & -0.3501 \end{bmatrix} \\ F_2 &= \begin{bmatrix} -25.2185 & 29.3267 & -0.8026 & -3.767 \end{bmatrix} \end{aligned} \tag{1.129}$$

Which once again matches our LQR controller exactly. The learning process and application can be seen in Figures 1.44 - 1.47

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 1.48 and 1.49 show how the various parameters converged through the learning process. Notice how it is only every other controller number that parameters change for each input.

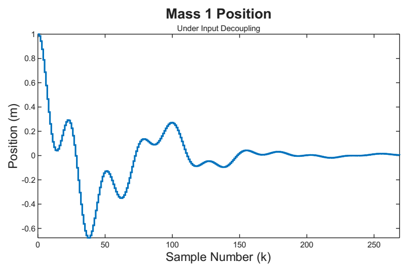

Figure 1.44: Mass 1 Position through Input Decoupling Trials

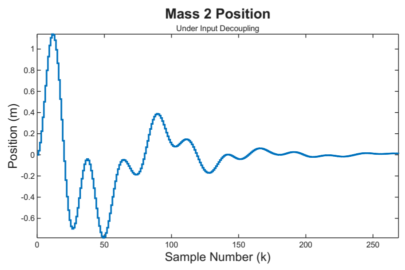

Figure 1.45: Mass 2 Position through Input Decoupling Trials

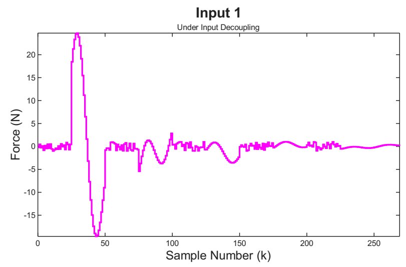

Figure 1.46: Input 1 Magnitude through Input Decoupling Trials

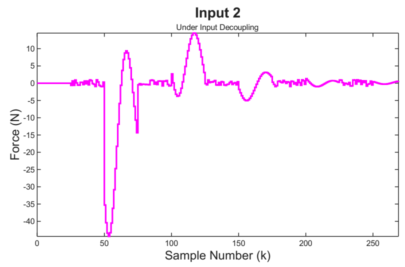

Figure 1.47: Input 2 Magnitude through Input Decoupling Trials

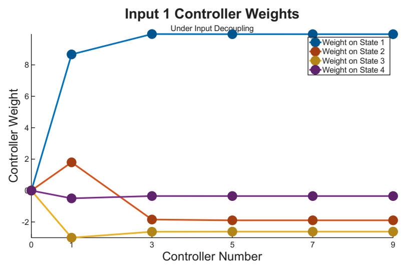

Figure 1.48: Input 1 Controller Weights through Input Decoupling Trials

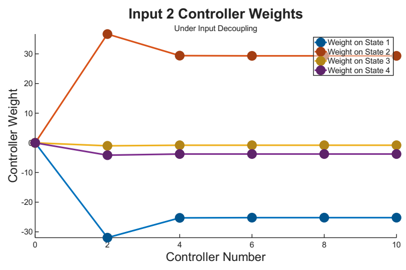

Figure 1.49: Input 2 Controller Weights through Input Decoupling Trials