Electronics and Computer Science
Faculty of Physical and Applied Sciences
University of Southampton

Arjun Patel (ap5g14)

02-05-2017

Design, Simulation and Analysis of Iterative Learning Control for a Multi-Axis Gantry Robot

Project supervisor: Professor Eric Rogers
Second examiner: Dr Nicolas G Green

A project report submitted for the award of
MEng Electromechanical Engineering

Abstract

Robots may repeat a procedure thousands of times a day, so any control algorithm implemented must be robust and reliable. This project will investigate different control algorithms mainly focusing on iterative learning controls (ILC). Utilising the power of Matlab and SIMULINK to run the dynamic systems. The simulation of the simple controllers like proportional phase-lead ILC to much more complex processes such as Fast Norm-Optimal ILC will be carried out on a model of the 3 axes gantry robot at the University of Southampton. Analysing the algorithms against one another and comparing the different strengths and weaknesses will help determine which controller has superior performance.

Contents

List of Figures

Acknowledgements and statement of originality

I would like to thank my supervisor Professor Eric Rogers for the guidance, making this project possible.

I confirm that all of the work in this project is my own. Plus any sources of other work referenced, quoted or discussed have been correctly acknowledged within the report.

Notation

This notation is used throughout this report.

t	Time
k	Iteration
u_k	Input to current trial
u_{k+1}	Input to next trial
e_k	Error in current trial
ref	Reference/desired output
y_k	Output of current trial

1 Introduction

Multi-axis gantry robots are stationary machines but have an arm or element which can be moved within a given region. The module at the end is free to move in all three axes (X, Y and Z) which makes them useful in the industry and can be employed for numerous tasks, such as moving objects from point A to point B in assembly lines.

With all real world systems, there is a delay between the input and output which is expressed by the system's transfer function. The complex transfer functions of the robot introduce delays and dampened oscillations in the output when responding to the input signal. As the gantry robot may repeat this task numerous times, it is essential to track its position to make sure the robot performs as desired every time. However, due to errors in the robot returning to the initial starting position and introduction of noise, an error between the desired position and actual position could be introduced. Iterative learning control, or ILC, is one of the various methods used in control system designs to eliminate error and to achieve high performance.

Comparing ILC to other systems performing the same procedure of repeatedly carrying out a task, a control system without any learning will always give the same trajectory, meaning errors are repeated. Each time the control operates, error signals from previous trials provide a lot of data which can be used to prevent errors such as overshoot, rise time, settling time and steady-state error. Iterative learning control can be highly effective in these situations to improve the transient response of the system so that the robot can keep performing its tasks. In contrast to adaptive learning control, which aims to modify the controller and thus the system to improve the output, ILC modifies the control signal to generate a feedforward control. Feedback control reacts to inputs therefore a lag is constantly present, but feedforward control acts in an anticipatory fashion to remove the lag for known inputs.

ILC requires the system to carry out a repetitive process where each iteration cycle takes the same time to complete and at the end of each task it resets to the same initial state. ILC utilises the result from previous iterations [1] to remove repetitive inaccuracies which is tremendously useful for robots performing repetitive tasks. Figure 1 shows a general diagram of ILC, where the input, $u_k(t)$, and output, $y_k(t)$, of the system plant are stored in some short term memory, allowing the iterative learning controller to calculate a new input signal for the next trial.

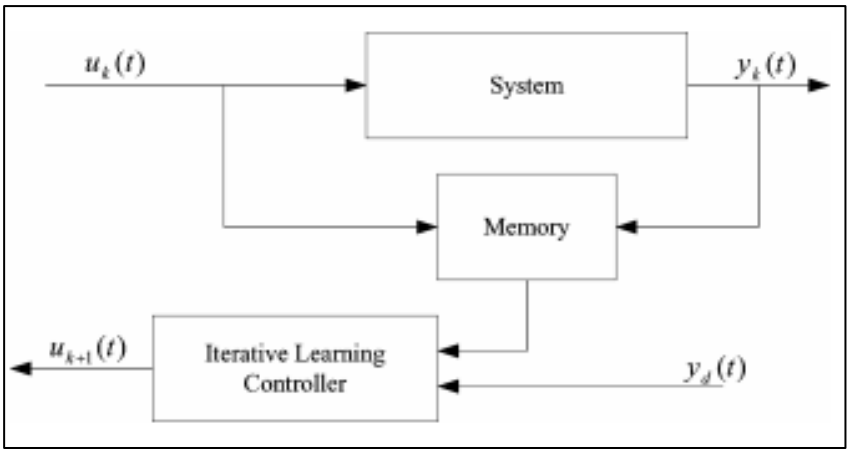


Figure 1: General iterative learning control (sourced from Iterative Learning Control: Brief Survey [1])

The algorithm uses the error between the desired output, $y_d(t)$, and the actual output to compute how the input signal needs to be changed. This means the controller finds the input needed to produce the output that is desired. This iterative process keeps repeating with the objective of creating the optimal input so that the error converges to zero. Conversely, for the initial trials the error between the output and the desired signal will be large, but within relatively few trials this should tend to zero if the control is correctly implemented.

An advantage of iterative learning control is the input is generated by the algorithm so it will become optimal over the course of the iterations. Furthermore, the control will remove errors which occur every cycle, but ILC is not able to anticipate for random zeroing errors or effects of noise in the system. To remove non-repeating disturbances ILC can be used in combination with a feedback controller for better results.

Through the project, an investigation will be undertaken into a handful of iterative learning control algorithms as well as Deadbeat control which aims to completely remove all effects of the plant in the system. The focus will be on the use of the variety of controls on a 3-axes gantry robot at the University of Southampton, of which a model has been made. There is a range of algorithms for ILC, and the best must be selected based on both performance and complexity. The control would ideally reduce tracking error to zero quickly and smoothly without passing through a period of instability, or requiring large control signal resulting in high demand from the actuators of the robot. In this project, the algorithms which were tested on the model include Phase lead ILC, Phase lead-lag ILC and Fast - Norm-Optimal ILC. The simulation and tuning of each of the different control algorithms was undertaken on Matlab with the SIMULINK package. In this report the control mechanisms will be evaluated on a set of three performance aspects; error convergence speed, long term stability and sensitivity to randomly occurring errors in the form of an initial state error.

2 Background research

2.1 Model

A model has been made of the gantry robot at the University of Southampton which includes each of its three axes of operation (X, Y and Z). The Z axis and Y axis are both stable third order dynamic systems whereas the X axis is a stable seventh order plant, this means that the control of the X axis is the hardest to control. As each of the axes has different plants the control algorithms will need to be tuned differently for each of them.

The transfer function for the Z axis;

$$\frac{15.887(s + 850.3)}{s\,(s^2 + 707.6s + 3.377 \times 10^5)}$$

The transfer function for the Y axis;

$$\frac{23.736(s + 661.2)}{s\,(s^2 + 426.7s + 1.744 \times 10^5)}$$

The transfer function for the X axis;

$$\frac{1.3077 \times 10^7\,(s + 113.4)(s^2 + 30.28s + 2.13 \times 10^4)}{s\,(s^2 + 61.57s + 1.125 \times 10^4)(s^2 + 227.9s + 5.647 \times 10^4)(s^2 + 466.1s + 6.142 \times 10^5)}$$

The desired path which this robot should follow is a simple and smooth pick and place task which lasts two seconds. To complete the operation the gantry robot first collects a stationary object, moves it over a fixed distance, places the object in another position and finally returns to the initial location. This task repeats for as long as required by the robot. The reference signal is split into the paths each of the three axes should follow so each of the axes of the model can be simulated and tested individually.

In real world systems, it is almost impossible to return after every trial to exactly the same initial condition, but if the homing system returns the robot to an initial position where the error is relatively small compared to the desired path then it can be assumed to be zero. The pick and place path fits within a 40mm by 160mm by 10mm area so to see the effect of initial state error ±0.1mm, ±1mm and ±5mm offsets will be used.

2.2 Matlab

Throughout this project, Matlab and the additional SIMULINK package was the software used. Matlab is widely used and it performs extremely well simulating and analysing dynamic systems with SIMULINK. The user-friendly interface and comprehensive tutorial archive [2], combined with prior experience with Matlab, made SIMULINK ideal for this project. Doing the project in simulation allows the testing of different algorithms and tuning variables without causing any damage to the robot and provides further methods to analyse the performance of each of the trails.

2.3 Deadbeat

Deadbeat control is a discrete time control theory where all the poles of the transfer function are cancelled out or are at zero, which eliminates all transient effects of the plant. The method relies on plant inversion which requires exact knowledge of the dynamics of

the system plant. When inverting the plant for the controller, any variances between the actual plant and the controller dynamics will hinder the performance of the system.

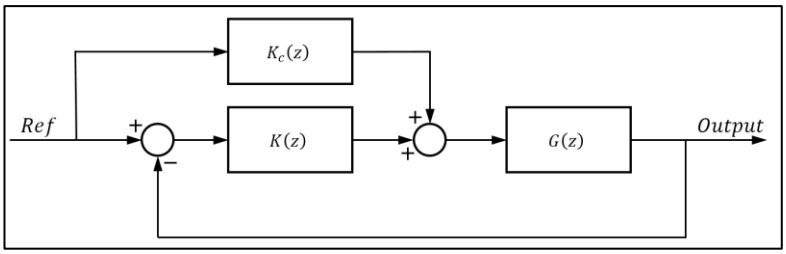


Figure 2: Block diagram of Deadbeat control

Deadbeat uses the above block diagram as the control where $G(z)$ is the system plant and $K(z)$ is the feedback with $K_c(z)$ is the controller. To remove all transient effects, deadbeat shows that if $K_c(z)$ is set to $1/G(z)$ then the transfer function of the whole system can be written as equation (1) below which simplifies to 1 when $K_c(z)$ is properly implemented. When the transfer function is 1 when the system has the perfect response where the input is directly mapped to the output.

$$Transfer\ function = \frac{K_c(z)G(z) + K(z)G(z)}{1 + K(z)G(z)} \quad (1)$$

However, for this model, $K_c(z)$ is not realisable therefore enough unit delays will need to be added until there are the same number of poles as zeroes. This will add a unit delay to the perfect response. The control forces the error to zero in a finite number of steps depending on the number of delays added to the system.

The instantaneous tracking provided, which only has a very small delay, makes deadbeat control very useful. In a real world situation, building a perfect inverse of a complex system is extremely complicated. When running the simulation on Matlab it is relatively simple to create the inverse and use it to control the plant. Additionally the deadbeat control demands a very high control signal output in order to stabilize the output.

2.4 Phase lead

This is a very simple form of iterative learning control where the new input into the system for each new iteration is defined by equation (2) and (3) from [3];

$$u_{k+1}(t) = u_k(t) + K \times e_k(t + \lambda) \quad (2)$$
$$e_k(t) = ref(t) - y_k(t) \quad (3)$$
$$where\ K, \lambda \in \mathbb{R}^+$$

Using calculated error from the previous trials, a phase shift is added, and the error scaled proportionally. This is used to derive the input to the next iteration, and the process repeats until the error converges to zero.

With this type of ILC, the values of the gain and phase shift need to be tuned for each system it is controlling. Due to this, the controller is not very robust as any change in the plant dynamics may cause the system to become unstable. Furthermore finding the optimal combination of both K and λ can be very time-consuming, as they have to be

experimentally adjusted. The phase lead acts as a predictive component to combat the transient delay behaviour of the plant itself so the desired output is reached in time. The gain value adjusts how quickly the convergence to zero error takes and whether it will over correct. Setting the gain too small will mean the error will take a long time to reach zero but too high will mean it will overshoot which depending on the process might be unacceptable.

This simple method of ILC is easy to tune but can take numerous trials to reach the desired output and will not even converge for systems which are too complex. To improve the convergence time a more elaborate iterative learning control needs to be implemented, which can be done by adding an additional element into the calculation.

2.5 Phase lead and lag

Building on the phase lead ILC algorithm, phase lag can also be added into the equation. The new component needs different constants as shown in equation (4)

$$u_{k+1}(t) = u_k(t) + K_1 \times e_k(t+\lambda) + K_2 \times e_k(t-\sigma) \quad (4)$$

$$e_k(t) = ref(t) - y_k(t) \quad (5)$$

$$where\ K_1, K_2, \lambda, \sigma \in \mathbb{R}^+$$

The process in which this algorithm eliminates error is much like that of the previous iterative learning control method. With Phase lead-lag ILC there are four unknown constants (K_1, K_2, λ and σ) which need to be manually modified so tuning becomes much more challenging and if incorrectly tuned it can cause poor response times or instability. Conversely, the Phase lead-lag iterative learning algorithm will have superior performance compared to the Phase lead algorithm if properly implemented.

The addition of the phase lag part has a reactive effect on the iterative routine and when adjusting the values of K_1 and K_2 a balance needs to be struck between the effect of the previous error and future error on the input to the next trial. This iterative learning control does not need information about the plant to run so can be used on systems where measurements of the plant are hard to take. This means it has to be experimentally tuned which can be time consuming and the optimal values are laborious to discover.

Phase lead and Phase lead and lag ILC both require tuning of the variables, but an accurate model of the plant is not necessary for the implementation of algorithms. Furthermore, a high-powered processor and large quantities memory are not required to run this more simple control.

2.6 Fast – Norm-Optimal Iterative Learning Control (F-NOILC)

Fast – Norm-Optimal Iterative Learning Control, from Ratcliffe et al. [4], is a revision of Norm-Optimal Iterative Learning Control, shown in [5], where less computation is needed per iteration cycle.

For a system modelled in state space format, given in equations (6) and (7), the cost function that is shown in equation (8) can be considered.

$$\dot{x}_k(t) = Ax_k(t) + Bu_k(t) \quad (6)$$

$$y_k(t) = Cx_k(t) \tag{7}$$

$$J_{k+1}(u_{k+1}) = \frac{1}{2}\sum_{t=0}^{N}\{e_{k+1}^T(t)\boldsymbol{Q}e_{k+1}(t) + (u_{k+1} - u_k)^T\boldsymbol{R}(u_{k+1} - u_k)\} \tag{8}$$

Where Q and R are symmetric weighting matrices and N is the number of samples per trial of the system. The value of Q affects the impact of the error on the new input and R affects the rate of which the input can change. Therefor depending on the system to improve responsiveness or stop overshoot the weighting matrices can be adjusted. This will affect the error convergence speed and control signal thus the amount of strain put on the actuators of the robot.

The matrix gain equation, predictive component equation and input update equation can be shown to be equations (9), (10) and (11) respectively as shown in [6]. Where the values $K(N) = 0$ and $\xi_{k+1}(N) = 0$ are the terminal conditions for equations (9) and (10).

$$K(t) = A^TK(t+1)A + C^TQ(t+1)C - \begin{bmatrix} A^TK(t+1)B \times \\ \{B^TK(t+1)B + R(t+1)\}^{-1} \\ \times B^TK(t+1)A \end{bmatrix} \tag{9}$$

$$\xi_{k+1}(t) = \{I + K(t)BR^{-1}(t)B^T\}^{-1} \times \begin{Bmatrix} A^T\xi_{k+1}(t+1) + \\ C^TQ(t+1)e_k(t+1) \end{Bmatrix} \tag{10}$$

$$u_{k+1}(t) = u_k(t) - \begin{bmatrix} \{B^TK(t)B + R(t)\}^{-1}B^TK(t) \\ \times A\{x_{k+1}(t) - x_k(t)\} \end{bmatrix} + R^{-1}(t)B^T\xi_{k+1}(t) \tag{11}$$

NOILC uses the characteristics of the plant to optimise the control so no manual tuning is needed. The state space matrices (A, B and D in (6) and (7)) are used in the calculation of the matrix gain (9), and predictive component (10) which are used to update the input to each trial. Unlike the Phase Lead and Lag ILC the gain and predictive component are time dependent so vary during each trial. This more complex system can work for higher order plants than the previous iterative learning controls.

With F-NOILC algorithm, these equations can be generalised so that fewer calculations need to be processed every iteration. The matrix gain equation (9) has no components which are required to be solved every iteration thus can be calculated before the system is running. The predictive component equation (10) can be simplified to equation (12) where each of its constants, alpha, beta and gamma, can be computed prior to running the simulation.

$$\xi_{k+1}(t) = \beta(t)\xi_{k+1}(t+1) + \gamma(t)e_k(t+1) \tag{12}$$
$$\alpha(t) = \{I + K(t)BR^{-1}(t)B^T\}^{-1} \tag{13}$$
$$\beta(t) = \alpha(t)A^T \tag{14}$$
$$\gamma(t) = \alpha(t)C^TQ(t+1) \tag{15}$$

The same type of simplification can also be implemented with the input update equation (11) with the constants omega and lambda.

$$u_{k+1}(t) = u_k(t) - \lambda(t)\{x_{k+1}(t) - x_k(t)\} + \omega(t)\xi_{k+1}(t) \tag{16}$$
$$\omega(t) = R^{-1}(t)B^T \tag{17}$$

$$\lambda(t) = \left(B^T K(t) B^T + R(t)\right)^{-1} B^T K(t) A \quad (18)$$

Therefore to implement F-NOILC, equations (12) and (16) are the only calculations that need to occur while the system is running and the values of $\alpha(t)$, $\beta(t)$, $\gamma(t)$, $\omega(t)$ and $\lambda(t)$ will be predetermined. This method requires no manual tuning as all values are calculated from the system's dynamics. If the dynamics were to change, the constants could be re-designed for the system.

The faster control is achieved by performing the majority of calculations before starting the iterations of the system. Additionally, the calculations are relatively less demanding meaning they require less time, so sampling rate can be improved. However, F-NOILC requires much more memory as it is static whereas NOILC can reuse memory each iteration. As for upgrading the processor, it can be much more troublesome and expensive than expanding the memory of the operating system, and as F-NOILC can improve performance, as long as there is adequate memory, it is more desirable. Like all iterative learning controls F-NOILC will not work perfectly from the very first trial, but should reach the desired output like the previously discussed ILC algorithms.

3 Design and Testing

3.1 Matlab

3.1.1 Testing dampened simple harmonic motion (SHM)

As I had no prior experience with SIMULINK, utilising the online tutorials from Mathworks, I initially created a dampened SHM system plant with a step input and proportional negative feedback, to gain an understanding of the program. Using Matlab, I was able to run the simulation for multiple gain values, see Appendix A for the code.
In SIMULINK the transfer function is very simple to implement using the standard Transfer Function block already in the library. Adding the other blocks from the library, I created a proportional feedback to a step response simulation as shown in Figure 3.

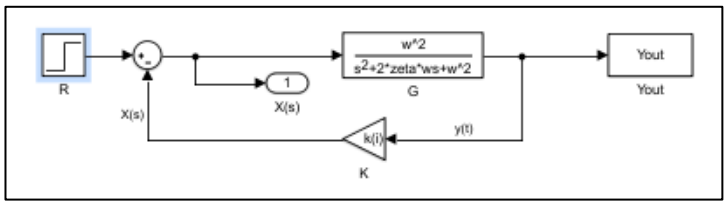


Figure 3: SIMULINK model of P control of SHM

Designing the blocks with no numerical values facilitates the constants of the plant and gain being taken directly from variables in the workspace of Matlab. This allows the variables to be easily altered and used by the Matlab script. The output of the simulations can either be set as a variable type called timeseries or as an array. The advantage of timeseries is that they include a time component to each element. The process of manipulating the timeseries adds a level of complexity but allows the shifting of values in time, which will be useful later in the project when calculating the new input in Proportional Phase Lead and Lag ILC.

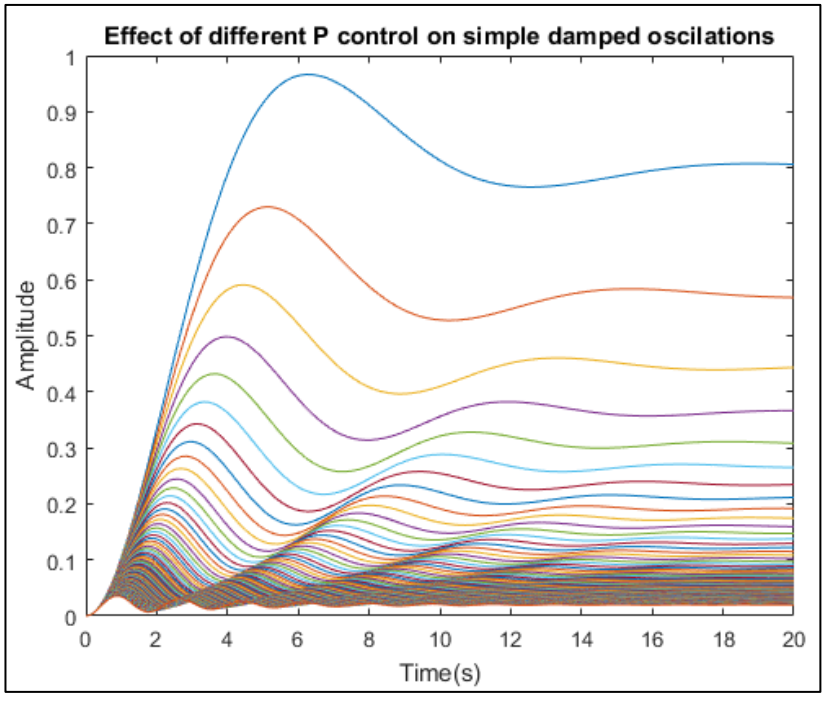


Figure 4: 2D plot of Time against output amplitude for different gain values

The Matlab code initially sets the values of the constants then runs the simulation for twenty seconds for different proportional gain values. For a small number of trials the output from the simulation can be plotted on the same set of 2D axes but for larger and more complex trails it becomes problematic. As shown in Figure 4 it is difficult to see how varying the gain affects the responses in the simulation.

Representing the results in a 3D graph, as shown in Figure 5, allows you to see the trends and patterns when increasing the value of the gain. Applying the 'plot3' function for each trial creates a 3D plot using the gain value as the X axis coordinates, with time on the Y axis and amplitude as the Z axis. Applying the 'hold on' function means that each trial would be plot onto the same set of axes.

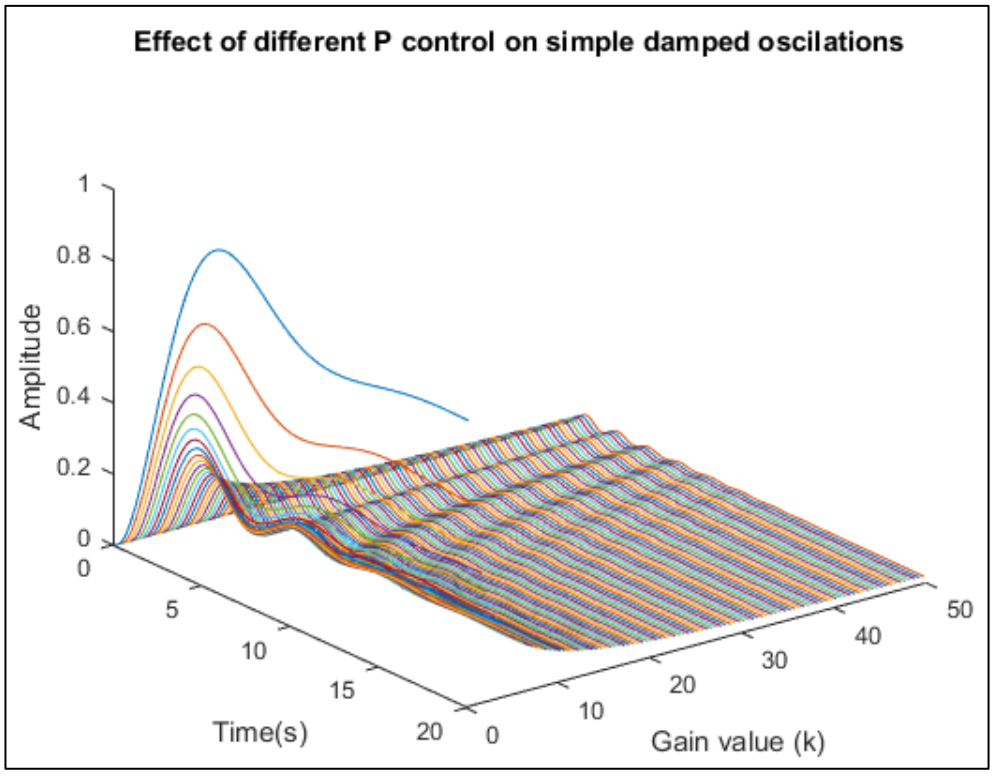


Figure 5: 3D plot of time against output amplitude against different gain values

As graphed above, the simulation of the dampened simple harmonic motion system behaves as expected, where increasing the proportional control decreases the settling time and lowers the final amplitude. With a better grasp of Matlab and SUMILINK and operation could progress onto simulations which are more demanding.

3.2 Deadbeat

Deadbeat control is simple to implement in Matlab, code and SIMULINK model is shown in Appendix B, as creating the inversion of the plant can be done perfectly through the code. As the model for all of the axes have more poles than zeroes, adequate unit delays (poles at zero) must be added to the inverted plant so it becomes realisable. The equations (19) and (20) below shows the Z axis model in discrete time and the transfer function $K_c(z)$, for the Y axis and X axis similar controls were used.

$$sysZ(z) = \frac{8.0647 \times 10^{-6}\ (z + 1.037)\ (z - 0.4268)}{(z - 1)\ (z^2 - 1.257z + 0.4928)} \tag{19}$$

$$K_c(z) = \frac{1.24 \times 10^{5}\ (z - 1)\ (z^2 - 1.257z + 0.4928)}{z\ \ (z + 1.037)\ (z - 0.4268)} \tag{20}$$

The Z axis deadbeat system resulted in the output shown in Figure 6 where the system becomes unstable very quickly. The system plant has a zero outside the unit circle (at $z = -1.037$) and as $K_c(z) = 1/G(z)$ the controller has a pole outside the unit circle. The unstable pole causes the output of the controller to oscillate with an exponentially increasing amplitude. The unstable behaviour is translated to the output in spite of the zero in the plant which should cancel it. The same happens for the Y and Z axis controllers as they too have unstable poles.

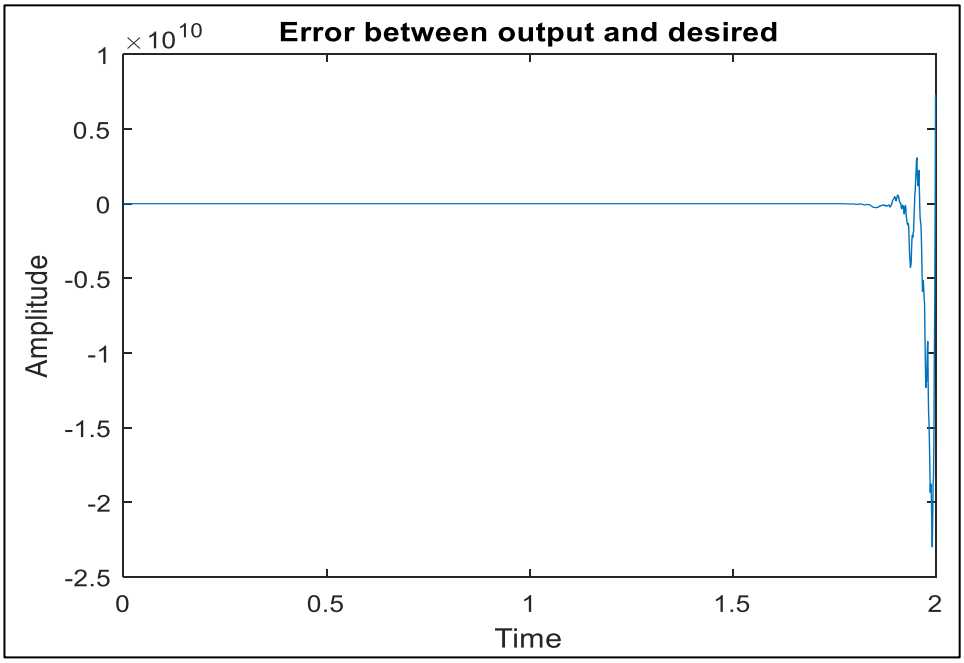


Figure 6: Amplitude against time for unchanged deadbeat control

If the $K_c(z)$ is edited to remove the unstable pole and replace it with a pole at zero as shown below in equation (21). The edited version of the Deadbeat control, also shown in Appendix B, is able to produce an output close to the reference signal as shown in Figure 7, but it is not a perfect response. While the control instantaniously brings down the error, it does not adapt with time, thus it does not regulate any error which may arise.

$$K_c(z) = \frac{1.24 \times 10^{5}\ (z - 1)\ (z^2 - 1.257z + 0.4928)}{z\ \ (z + 1.037)\ (z - 0.4268)} \tag{21}$$

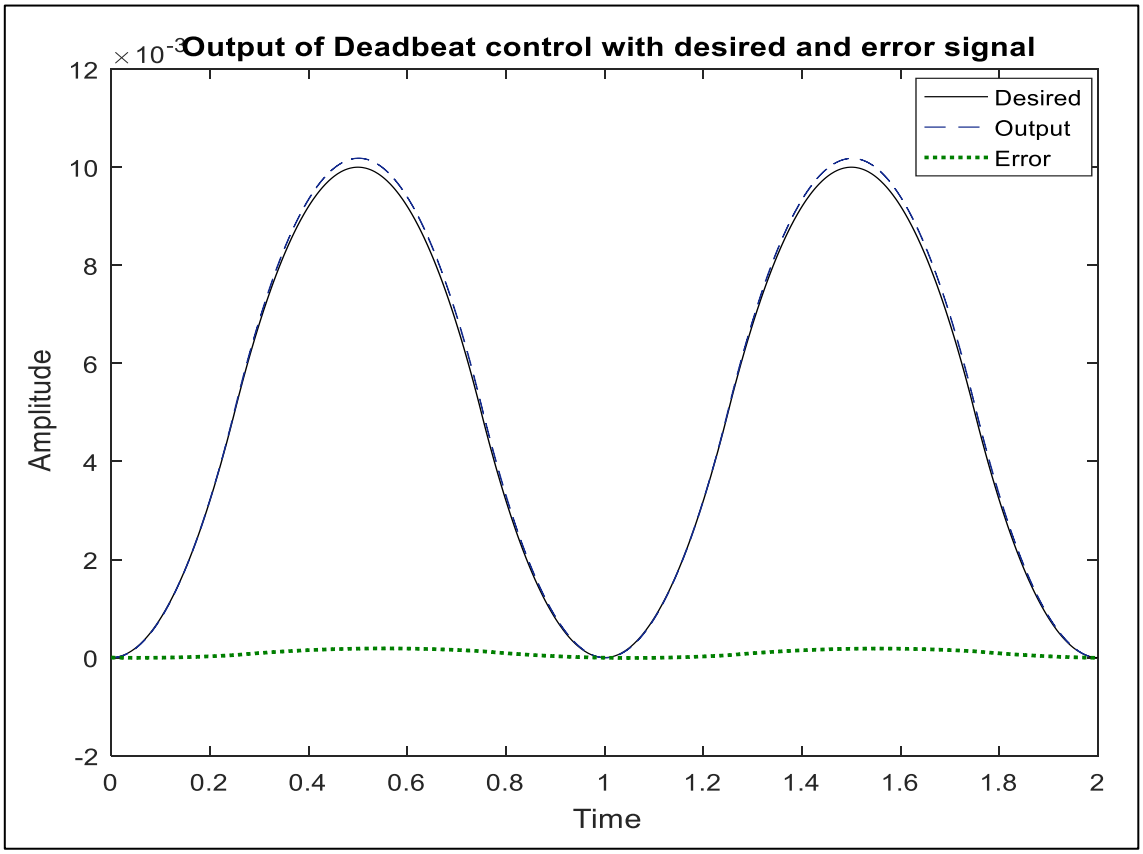


Figure 7: Amplitude against time for adjusted deadbeat control on Z axis

3.3 Iterative learning control

3.3.1 Proportional error ILC

3.3.1.1 Design

Proportional error control is a very basic form of iterative learning control where the algorithm is given is by the equation (22) below.

$$u_{k+1}(t) = u_k(t) + K \times e_k(t) \quad \{e_k(t) = ref(t) - y_k(t)\} \tag{22}$$

Where the error gain, K, is the only constant which needs to be adjusted. The control operated on the dampened SHM where the desired output was a sinusoidal wave of period 6π and amplitude 1.

To create the iterative learning control, building on the previous SIMULINK model and Matlab code, Appendix C shows the code. Using blocks from the SIMULINK library, all the calculations could be run in the simulation (see Figure 8 for the model) as a result the Matlab code was only required to pass over the input to the next trial and store the output data to be plotted.

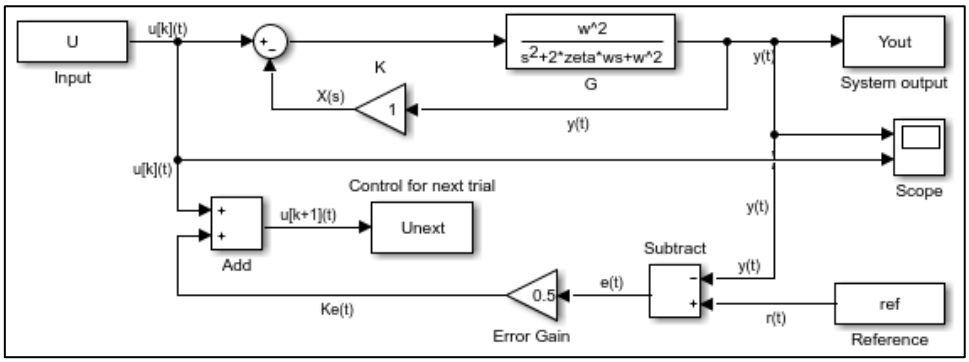


Figure 8: Model of proportional ILC

In the Matlab code, I added the feature to extract the output data from any chosen single test, an example is shown in Figure 9. The 2D graph output allows you to visualise the output from a single test to see if there are any errors compared to the desired output.

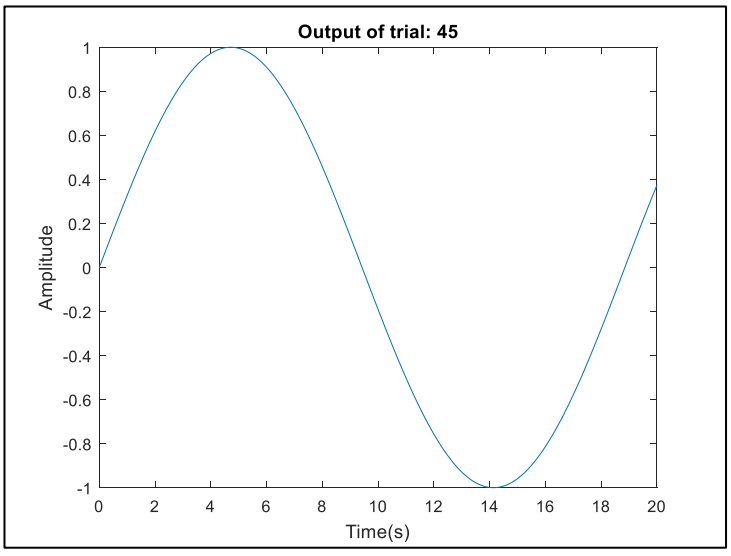


Figure 9: Output of proportional ILC from

Furthermore, to improve the way in which the data is presented, I changed the format of the 3D graph from multiple 3D line plots to a single 3D surface plot. To achieve this effect the simulation data from each iteration needed to be added into different individual matrices. I made output, input, time and trial matrices so that both input and output can be seen with respect to time and trial number. Using the 'surface' function to plot trials on the X axis, time on the Y axis and the amplitude on the Z axis produced the 3D graphs, shown in Figure 10 and 11.

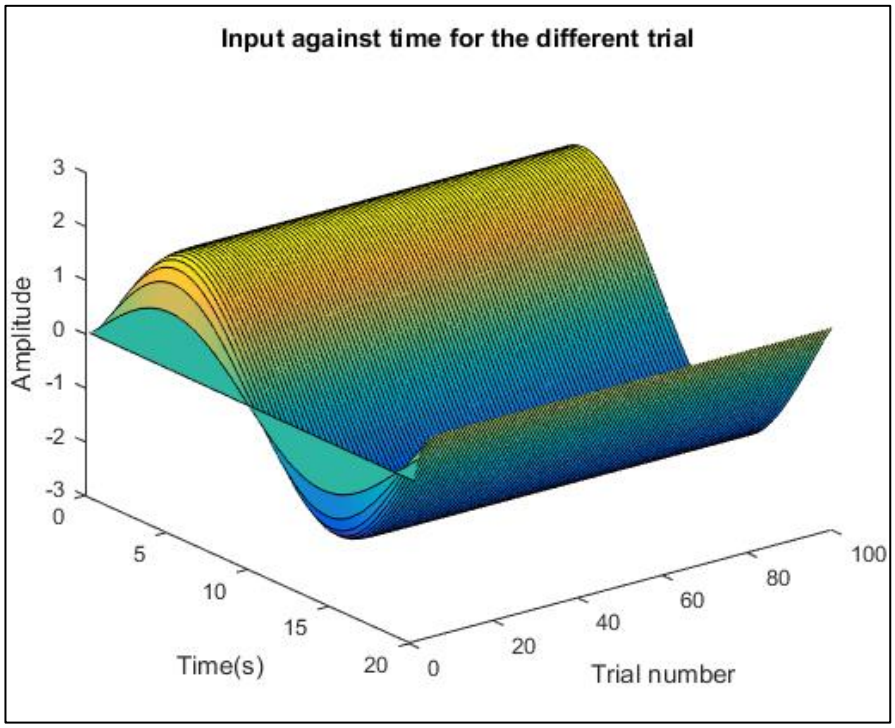


Figure 10: Surface plot of input to simulation for each trial

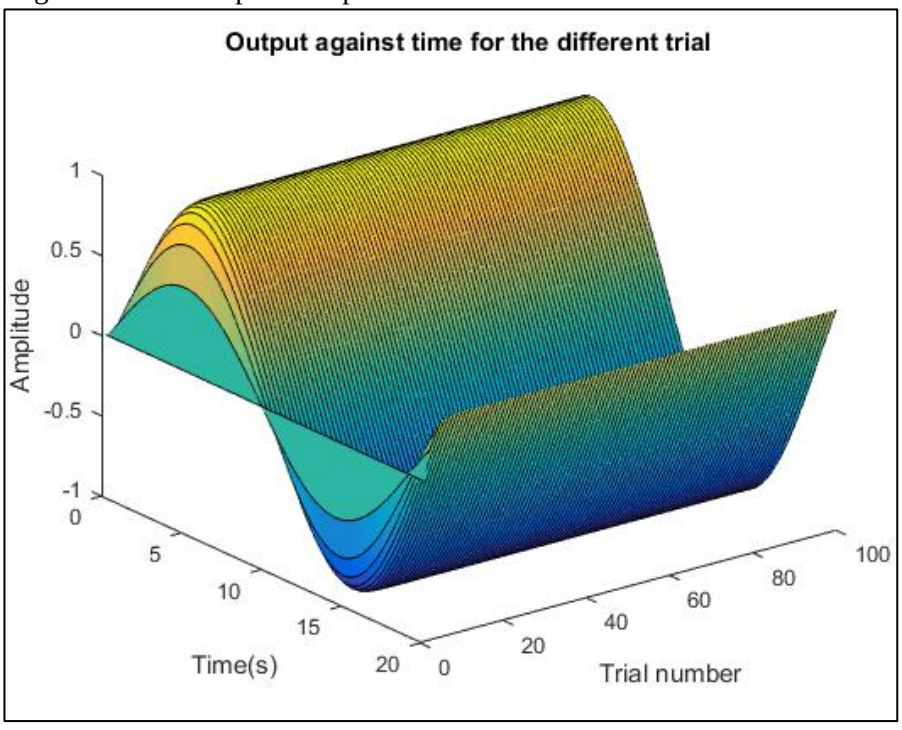


Figure 11: Surface plot of output of simulation for each trial

3.3.1.2 Tuning and Results

Tuning is relatively straightforward as the error gain is the only variable that needs to be changed. If the error gain is too small, then the output will have a long response time and will take a long time to reach the reference input. Conversely, if the error gain value is too large (approximately greater than eight), then the system becomes unstable for this particular system. The rudimentary ILC control implemented here falls apart if the system is more complicated than the dampened SHM it is controlling.

3.3.2 Phase lead ILC

3.3.2.1 Design

Moving away from the low order plant of the dampened SHM, the model of the gantry robot was then used, starting with the plant of the Z axis as it is simpler where amplitude is in meters. The desired trajectory for the axis to carry out the pick and place operation was given.

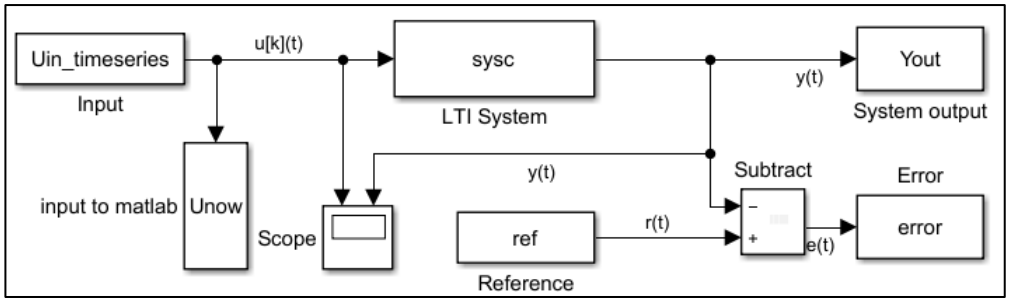


Figure 13: SIMULINK model the outputs unaltered error

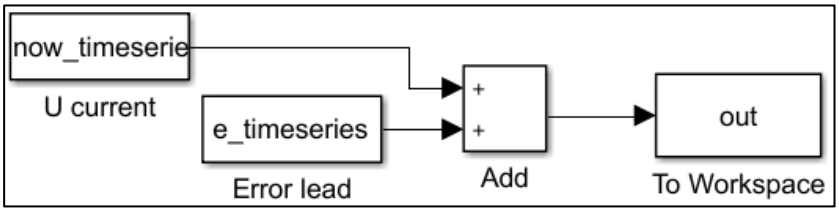


Figure 12: SIMULINK model to add timeseries and create input to next iteration

There is no simple way in the SIMULINK library to add in a phase lead into the error timeseries so using the model in Figure 12 I simulated the system plant and calculated the error. The phase lead ILC algorithm was carried out by the Matlab code which is in Appendix D. Utilising the fact that the error variable is a timeseries type only the time component is manipulated to represent the phase lead shift. To carry out the addition of the error to the current input to create the input to the next trial, I used a second SIMULINK file, Figure 13. The reason for this is the Matlab code was unable to directly add timeseries with different time components.

I further improved the code by optimising some of the functions and pre-allocating the memory used to improve the time taken for the simulation the run. The addition of the surface plot of the error creates a visual representation of the way in which the error varies over time and how it decreases over each iteration, shown in Figure 14.

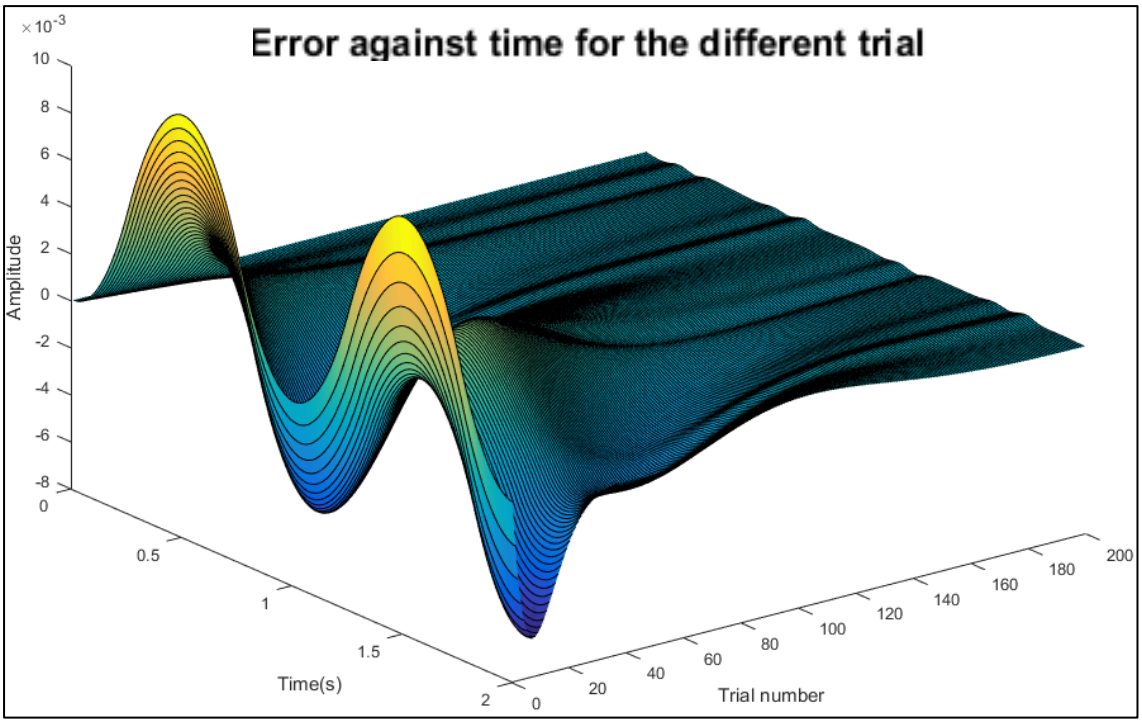


Figure 14: Error with phase lead control for Z axis over 200 trials

The code also computes and stores the mean-squared error (MSE) value of each trial relative to the initial trial. These values are plotted against trials for a visualisation of the amount of error after each iteration and how it is increasing or decreasing. When MSE error is in the order of 10^{-3} then the output and the reference are very close to one another.

Gantry robots often have an initial state error where the axes are not always reset back to the exact position for the start of each trial. To simulate this I added an offset which is created from a pseudorandom number. The random number generator uses the trial number as a seed so that over different tests of other magnitude and ILC controls, the same number is generated for each trial but each is random compared to the previous trial. The bounded random number generated is then added to the whole error array to simulate a shift in the starting position. The offset will show the robustness of the algorithm to variation in homing error in each iteration of the control.

3.3.2.2 Tuning and Results

The gain and lambda are the two variables which need to be tuned to stabilise the output. As previously, the gain affects the speed in which the output responds to the error. Including lambda means the error acts as a predictive component. If lambda is too small, the controller will just work like the previous ILC; also the ILC algorithm will not function if lambda is too big.

In addition to the Z axis system function, the Y axis function could be stabilised relatively quickly as they are both 3rd order systems. The X axis function is a 7th order system and was the hardest to control as it is of a higher order. The values of $gain = 5$ and $lambda = 0.01$ are not optimal values but can be used to control all three axes. Figure 15 and 16 shows the input and output of the X axis with this control implemented after 300 iterations

plotted in dotted blue and reference signal plotted in dashed red, where the two signals are almost identical.

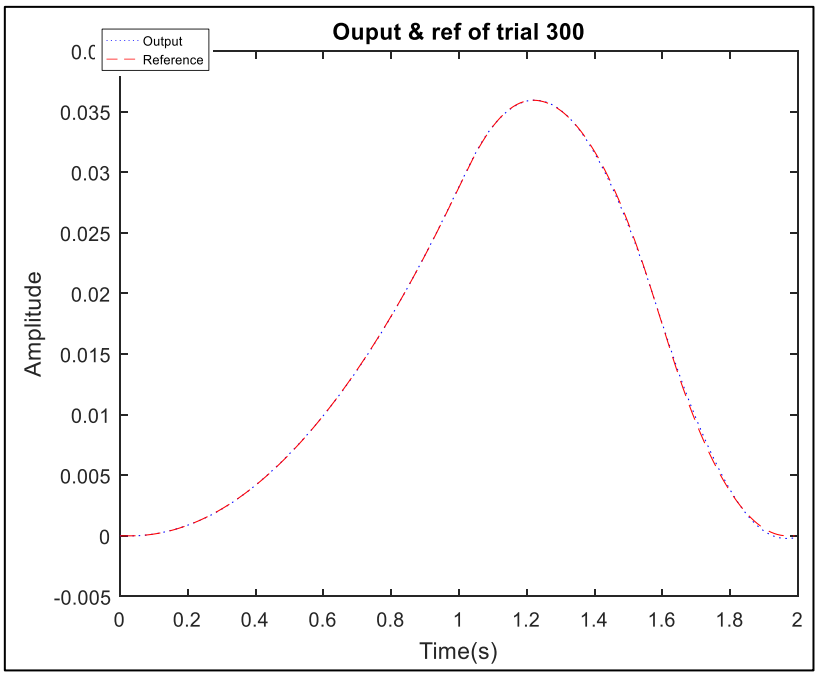


Figure 16: Output from trial 300 of Proportional lead ILC for X axis

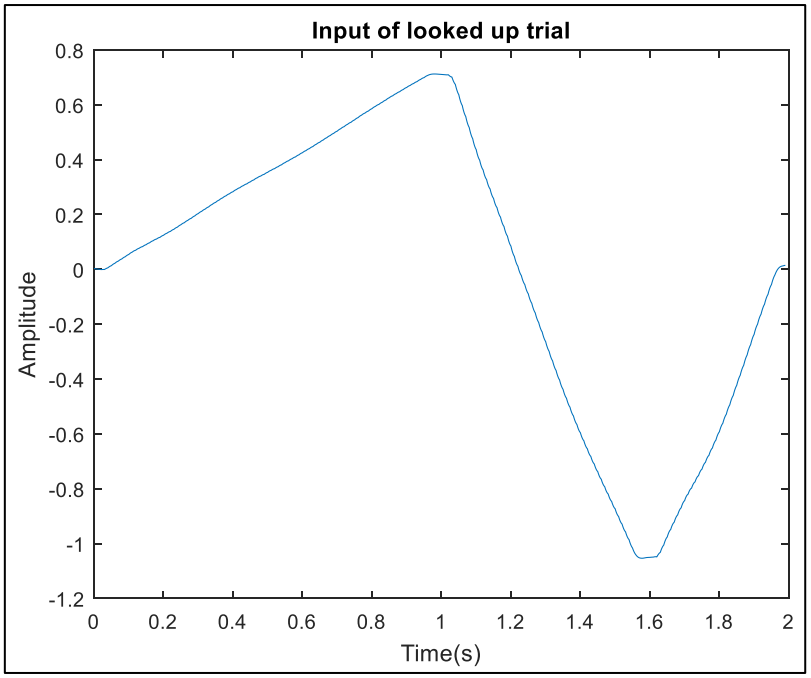


Figure 15: Input from trial 300 of Proportional lead ILC for X axis

However if allowed to run for a lot more iterations the MSE starts to increase again, as shown by the plot in Figure 17. The increase indicates the control is unstable and is diverging away from zero error. Rising MSE occurs from the small short oscillations which are amplified by the iterative algorithm when trying the control the system. The input for trial 4000 is shown in Figure 18, which shows the start of the oscillations which only increase with trials as the controller overcompensates. This causes problems for the long term performance and I am unable to find any values of the gain and lambda to solve the problem.

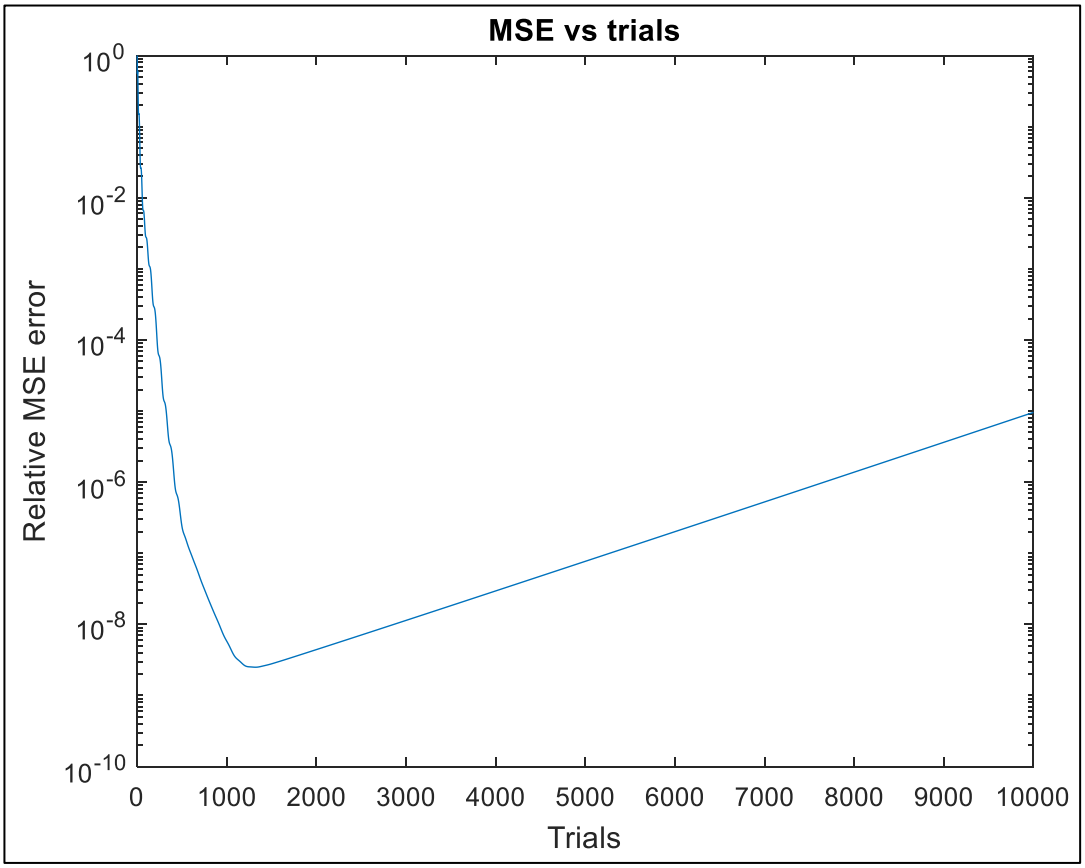


Figure 18: Effect of Proportional lead ILC on MSE value run for 10000 trials on the X axis

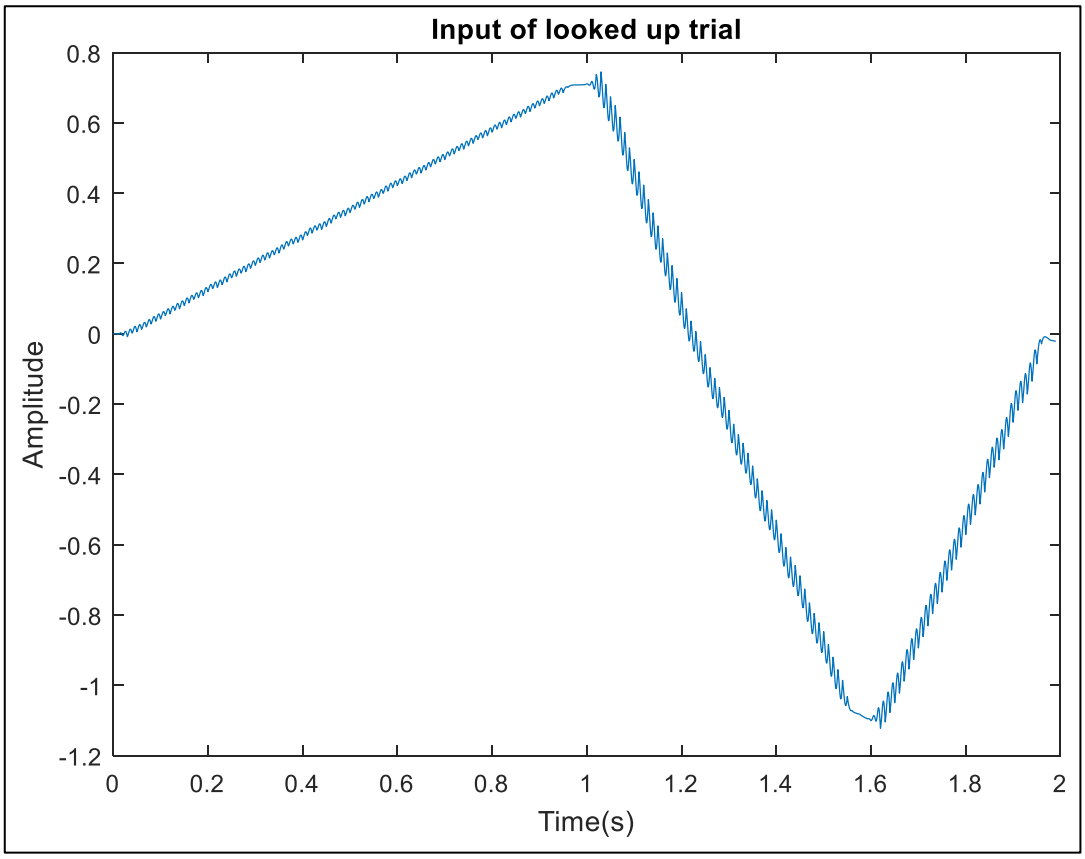


Figure 17: Input from trial 4000 of Proportional lead ILC for X axis

The plots shown in Figure 19 and 20 demonstrate the MSE error with different tuning values of both gain and lambda. Figure 19 shows the effect of the gain changing from 3 through to 30 with a lambda value of 0.1 to amplify the effect. As illustrated, as the gain increases the Phase lead ILC reduces the error faster but introduces an increase in error for the first few trials. Moreover, the higher the gain value the faster the system will reach the point where it begins to become unstable.

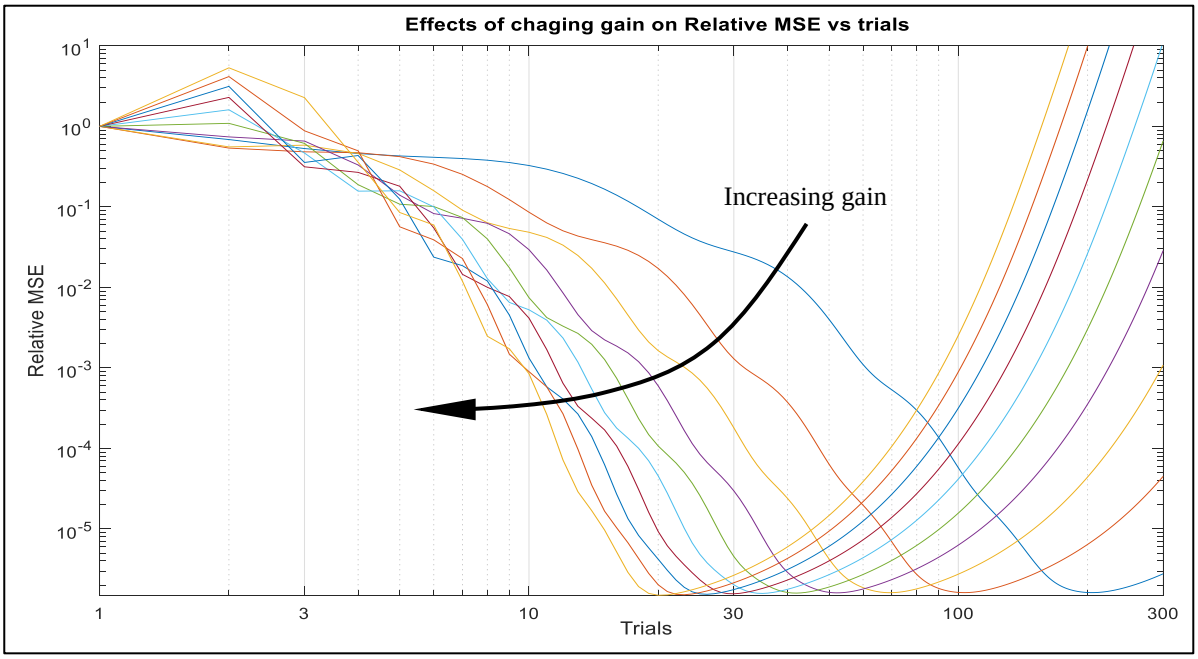


Figure 20: Effects of increasing the gain value on Proportional lead ILC for lambda equals 0.1

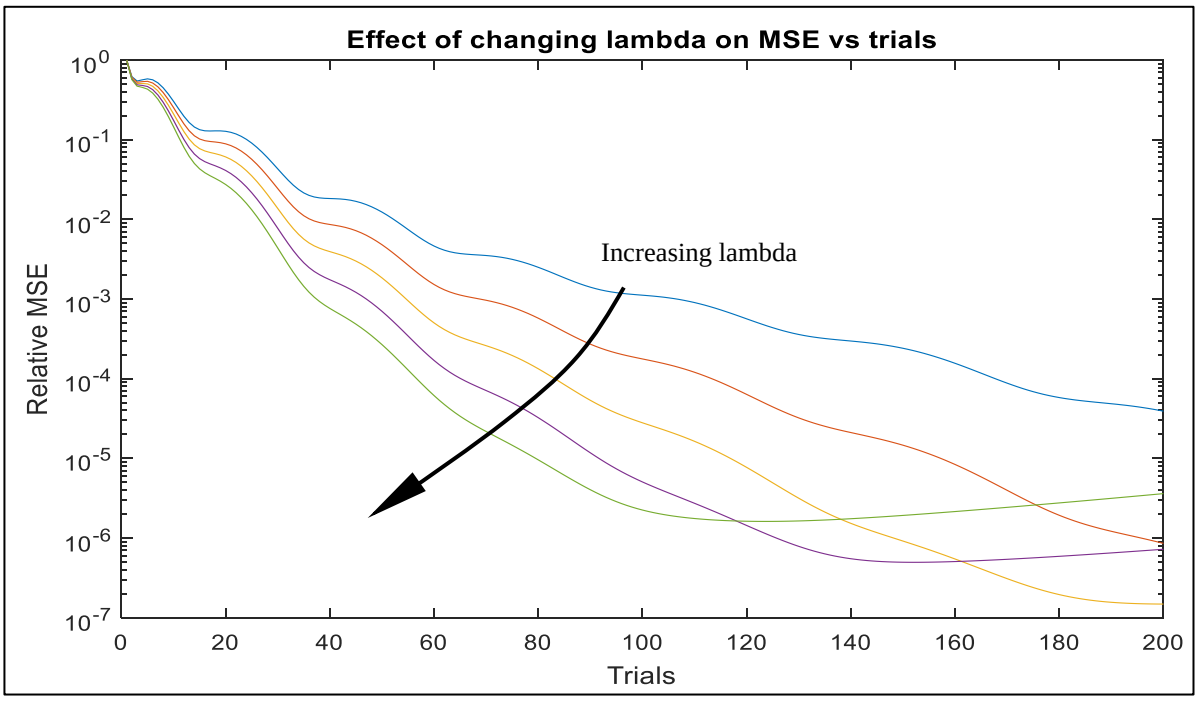


Figure 19: Effects of varying the gain value on Proportional lead ILC

Figure 20 shows the effect of increasing lambda from 0.02 to 0.10 with the gain value set to 5. As lambda increases the MSE falls faster but the point where the control becomes unstable occurs later for the lower values of lambda. The plots show the trade offs which need to be made for each system when adjusting the variables for the ILC.

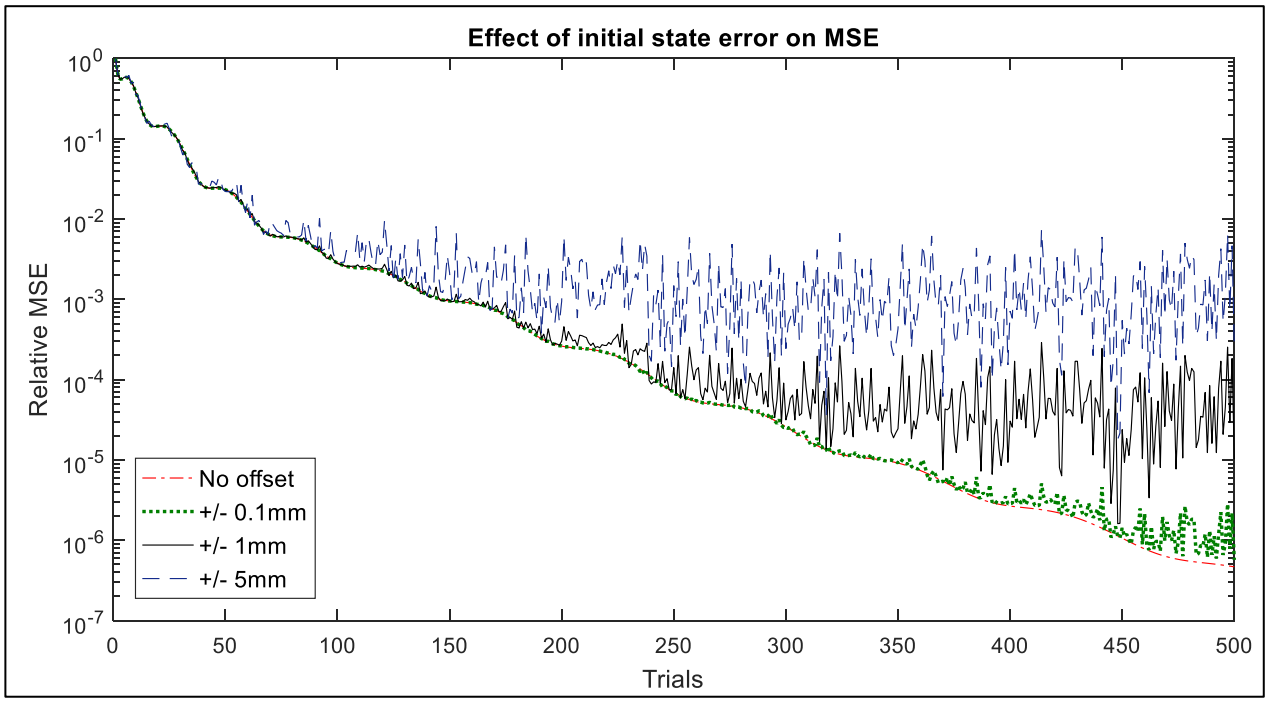


Figure 21: Effect of different maximum offsets on MSE value of Proportional lead ILC for 500 trials

The addition of initial state errors of maximum sizes of ±0.1mm, ±1mm and ±5mm are shown in Figure 21. Even for the relatively large offset of ±5mm, the MSE error stays approximately lower than 10^{-2} and as the offset decreases the MSE error decreases. The offset does not affect the convergence speed of the iterative learning control but increases the minimum MSE error reached.

3.3.3 Phase lead and lag ILC

3.3.3.1 Design

To improve the iterative learning control the phase lag element was added. Building on the Phase lead ILC algorithm it was not difficult to add the phase lag element. To implement the new control the SIMULINK model in Figure 13 could be kept the same, and only a single input block was needed in the additional simulation model. The only advancements to the Matlab code necessary are the new variables and two lines of code for the manipulation of the error, see changed code in Appendix E.

3.3.3.2 Tuning and Results

The process of tuning this ILC algorithm involved attempting different combinations of the four constants (K_1, K_2, λ and σ). As aforementioned, the gain values K_1 and K_2 affect the impact of the lead and lag elements respectively. The phase shifts do not need to be of an equal size for stability. The magnitude of lambda, however, needs to be larger than sigma for stability to be achieved. The Z axis and Y axis were simple to stabilise, and as

expected the higher order X axis was more time consuming. All three axes could be stabilised by the values of $K_1 = 10, K_2 = 5, \lambda = 0.005$ and $\sigma = 0.001$. As shown by Figure 22, there are no large spikes in the error and converges to near zero after 50 trials.

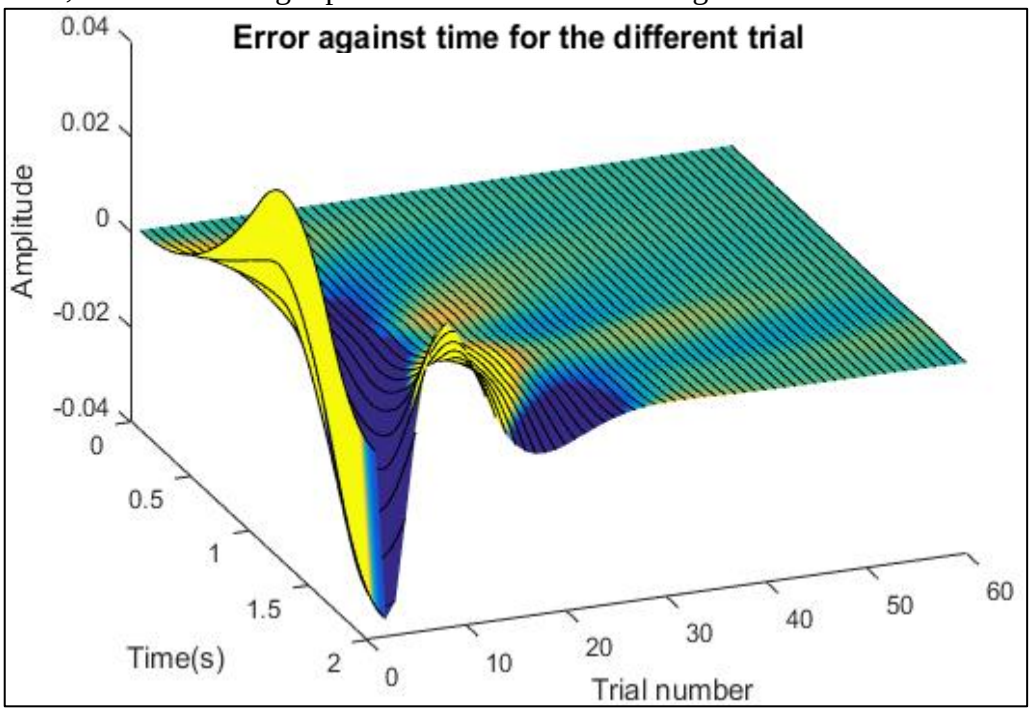


Figure 22: Error surface plot for X axis Phase lead-lag ILC algorithm

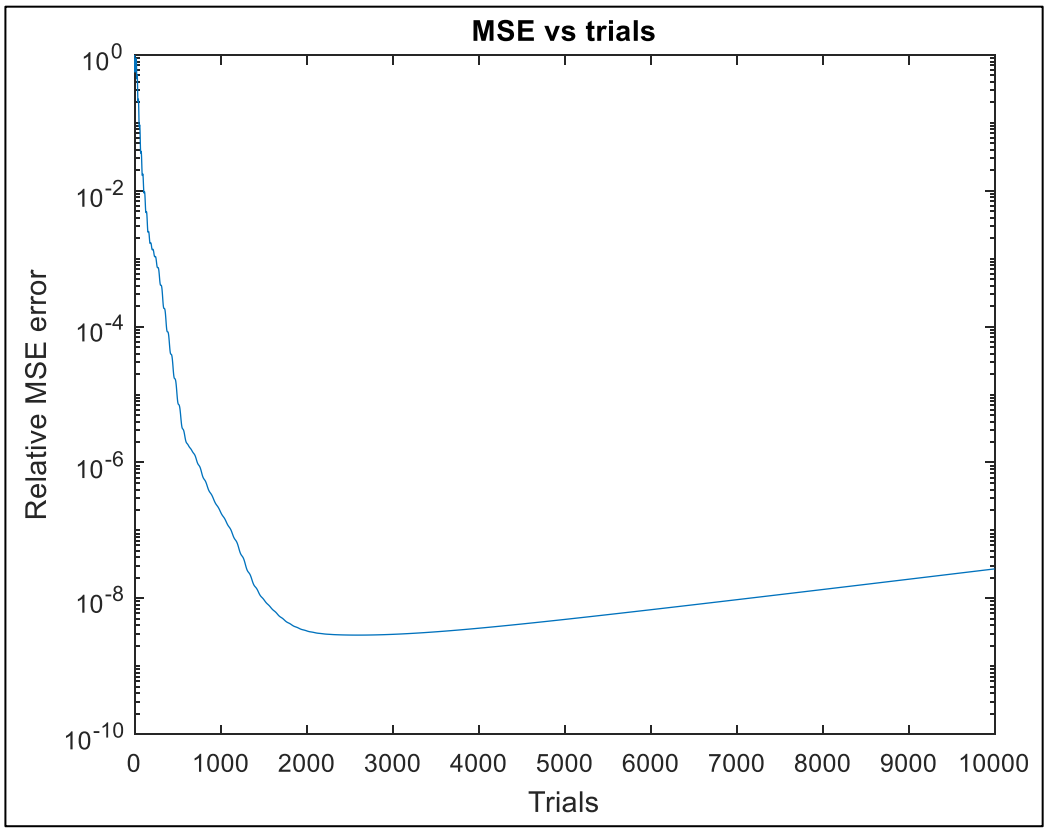


Figure 23: Effect of Proportional lead and lag ILC on MSE value run for 10000 trials on the X axis

Figure 23 shows the convergence speed and long term performance of the phase lead and lag ILC. Much like the phase lead iterative learning control if the iterative learning

algorithm is run for a long time then the error increases. The cause for the divergence away from zero tracking error comes from the high frequency oscillations which occur after a large number of iterations which grow in amplitude.

When offsets are introduced into the simulation of the Phase lead and lag ILC, the results (shown in Figure 24) show sensitivity to the initial state error added to the system. When a ±0.1mm offset is added the relative MSE becomes less than 10^{-4} but when the ±5mm offset is added the minimum error reached is much higher.

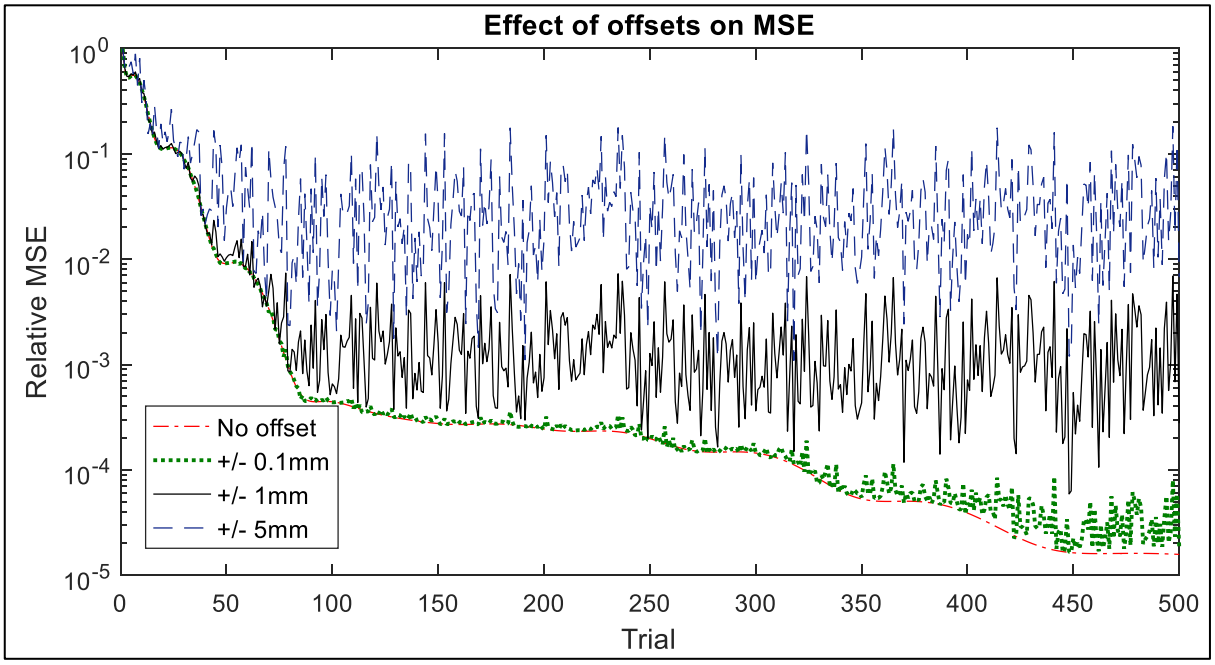


Figure 24: Effect of different maximum offsets on MSE value of Proportional lead and lag ILC for 500 trials

3.3.4 F-NOILC algorithm

3.3.4.1 SIMULINK model

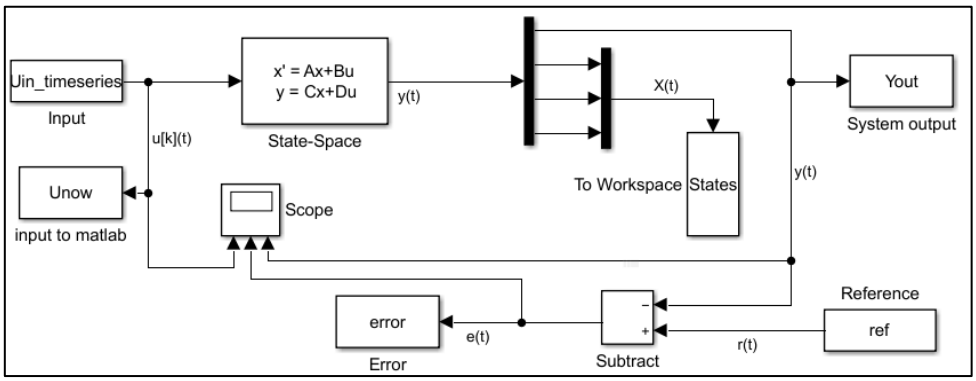


Figure 25: SIMULINK model for F-NOILC

The Fast – Norm-Optimal iterative control algorithm requires values from the system plant taken from the matrices from the state space form. The whole SIMULINK model needed to be updated, see Figure 25. Replacing the transfer function plant with the state space model block which has the inputs of the A, B, C and D matrices. However, I altered block input variables of C plus D to be zeroes of the size to match, both are shown below. The

reason for the change was so that the states of the system are output from the block so they can be used for the calculations of the next input.

$$C = \begin{bmatrix} sys.C(1,1) & sys.C(1,2) & sys.C(1,3) \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \& \quad D = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The second state space equation is shown below in equation (23) which means the addition of the identity matrix to C produces the effect shown below in equation (24).

$$y(t) = C \times x(t) + D \times u(t) \tag{23}$$

Substituting the values into the equation;

$$output = \begin{bmatrix} C_{(1,1)} \times x_{(1,1)}(t) & C_{(1,2)} \times x_{(2,1)}(t) & C_{(1,3)} \times x_{(3,1)}(t) \\ x_{(1,1)}(t) & 0 & 0 \\ 0 & x_{(2,1)}(t) & 0 \\ 0 & 0 & x_{(3,1)}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{24}$$

$$output = \begin{bmatrix} y(t) \\ x(t) \end{bmatrix}$$

With the use of a demultiplexer and multiplexer in unison, the signal $y(t)$ can be separated from $x(t)$ and both can be passed to the Matlab code.

3.3.4.2 Attempt to use existing Riccati equation to create gain matrix

3.3.4.2.1 Code Design

The new ILC required the Matlab code to be completely rewritten, see Appendix F for the first attempt. The system plant initially needed to be converted into a discrete time domain state space system which is carried out by Matlab. The gain matrix K is solved by the existing built discrete-time algebraic Riccati equation function 'dare'. The matrix is then used with the plant matrices to find alpha, beta, gamma, omega and lambda. After each trial, the code calculates backwards in time through the trial to work out all the values for the new input, as the predictive component $\xi_{k+1}(t)$ has a terminal condition.

3.3.4.2.2 Tuning and results

The tuning of F-NOILC only involves choosing values for Q and R so that the system is stable. Q effects the weighting of the error function and R the amount the new input can change from the previous trial. When the algorithm computes 100 iterations with Q=10^13 and R=0.001, it can be seen that the output is moving closer to the reference, as shown in the surface plot in Figure 26. However between trials 0 and 10, very large spikes are generated to drive the output signal. If ran in real life, the system may break down before it reaches stability.

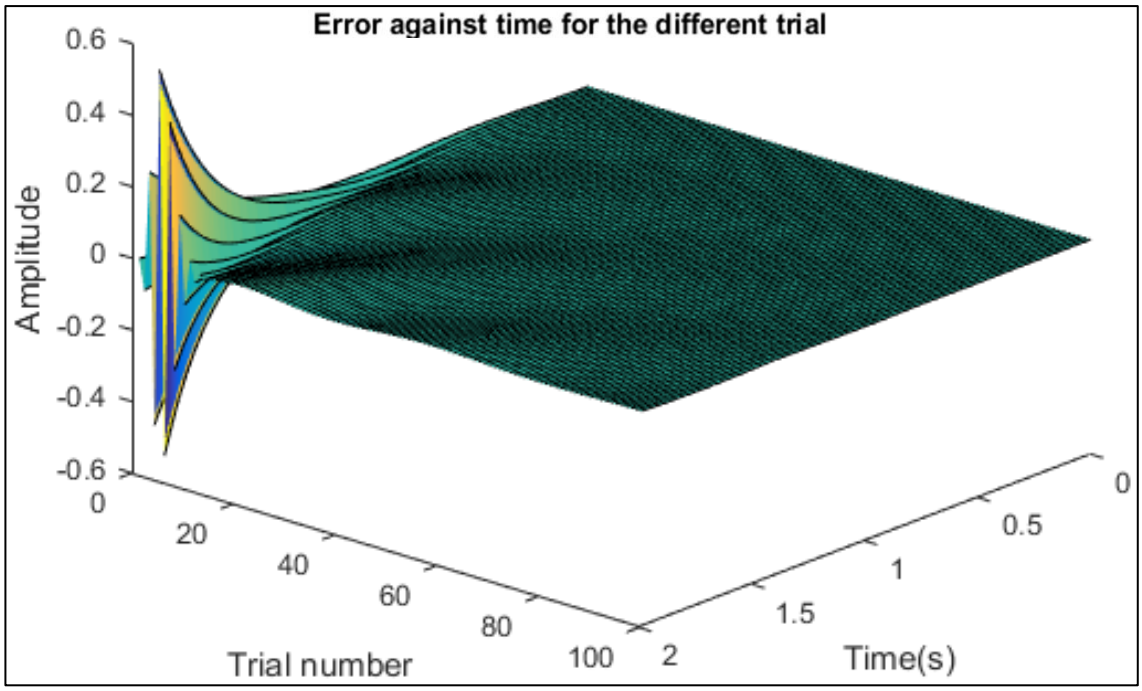


Figure 26: Surface plot of error for F-NOILC X axis over 100 trials

The plot of relative to initial MSE vs. trials, shown in Figure 27, shows an initial large increase in error. The large increase in the error would not be acceptable in commercial use and points to problem within the code.

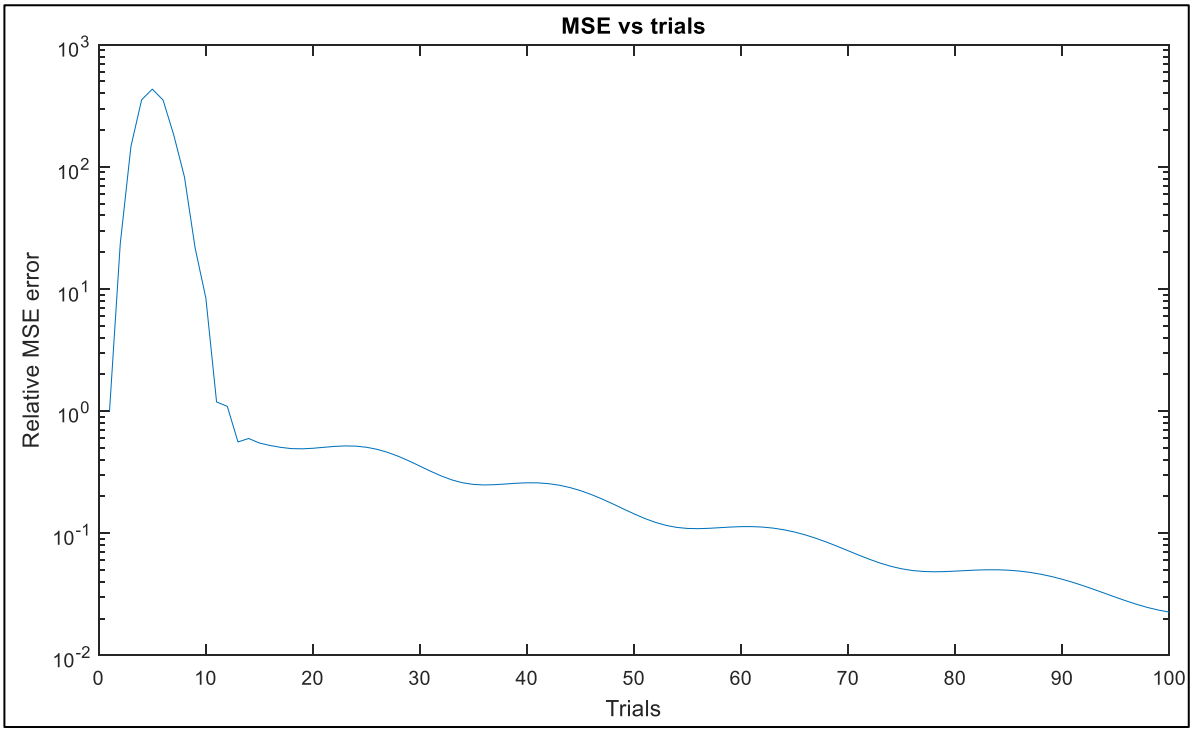


Figure 27: Effect of F-NOILC on MSE value run for 100 trials on the X axis

When the cost function values, Q and R, are altered the algorithm may not produce a stable system. The lowest value of mean squared error for each value of Q and R, for 50 trials, provides a measure of how much the error has been reduced compared to the initial MSE. Figure 28 shows a surface plot of the effect of MSE for different values of Q and R, and it can be seen that there are large areas where no improvement was made.

Comparing the data from Ratcliffe et al. [4], in Figure 29, to the data I produced for varying Q and R, Figure 28, the magnitude of the values are vastly different.

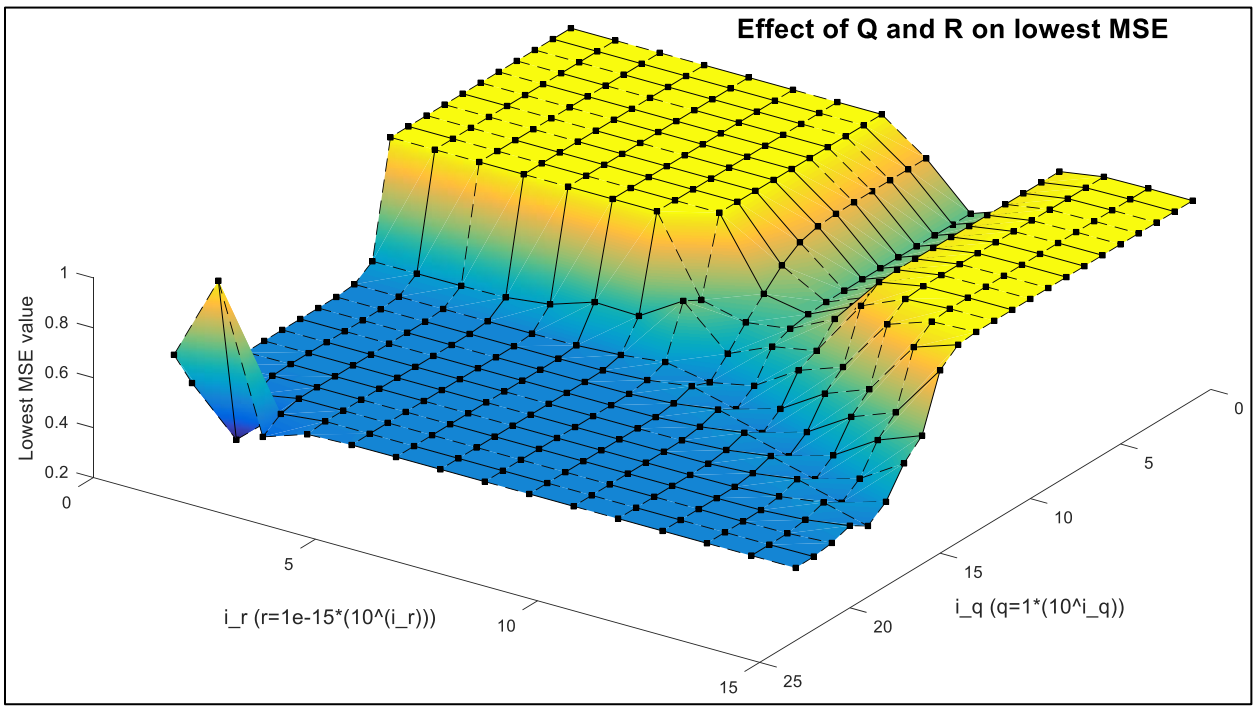


Figure 28: Surface plot of effects of Q and R on lowest MSE value over 500 trials of F-NOILC

Furthermore, the shapes of the surface plot do not match up with that with Figure 29. Moreover from the MSE data in [4] the error should not reach such a high values like the data in Figure 27. I concluded that F-NOILC was not correctly executed in my code and is needed to be adjusted.

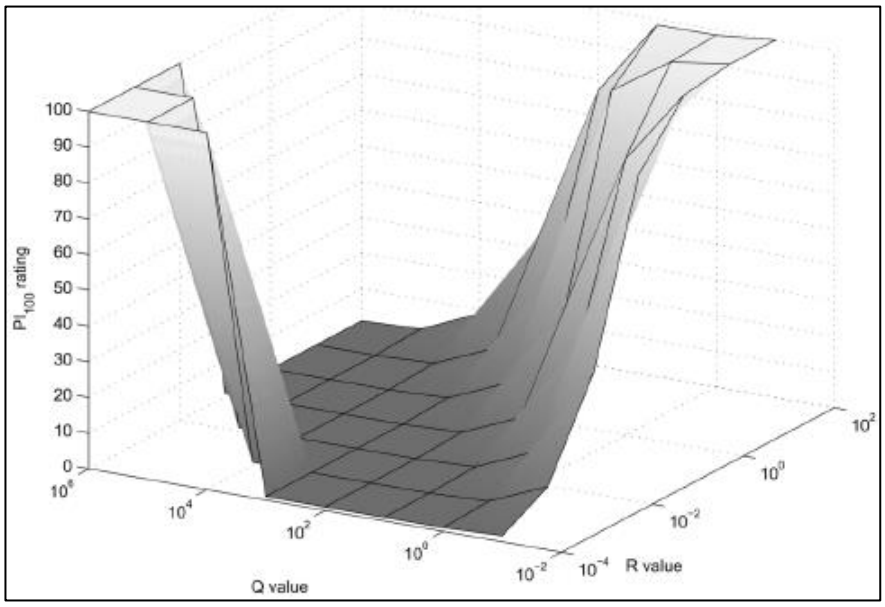


Figure 29: PI for x-axis over 100 trials varying Q and R (sourced from Ratcliffe et al. [4])

3.3.4.3 Revision of F-NOILC

3.3.4.3.1 Code Design

To accomplish F-NOILC, I moved from using the Matlab ‘dare’ function to solving the values for the gain matrix manually in the code. To do this I ran a loop backwards through time starting from K terminal condition and stored the values in an array of three by three matrices (increasing to seven by seven for X axis). Furthermore, I improved the code run times by simplifying a lot of the processes and adjusted the way in which the predictive component is calculated, code is shown in Appendix F

3.3.4.3.2 Tuning and results

As shown in Figure 30, with the improved code the effect of R and Q on the minimum relative MSE value after 100 trials matches the shape of Figure 29. The plot for the Y and X axis are the same shape but the R and Q are shifted slightly. The valley profile demonstrations there are two regions where the error does not decrease and a middle area for which the algorithm works. The diagonal profile shows that the exact values of R and Q are not important but it is the relative magnitudes, Q roughly a hundred times larger than R.

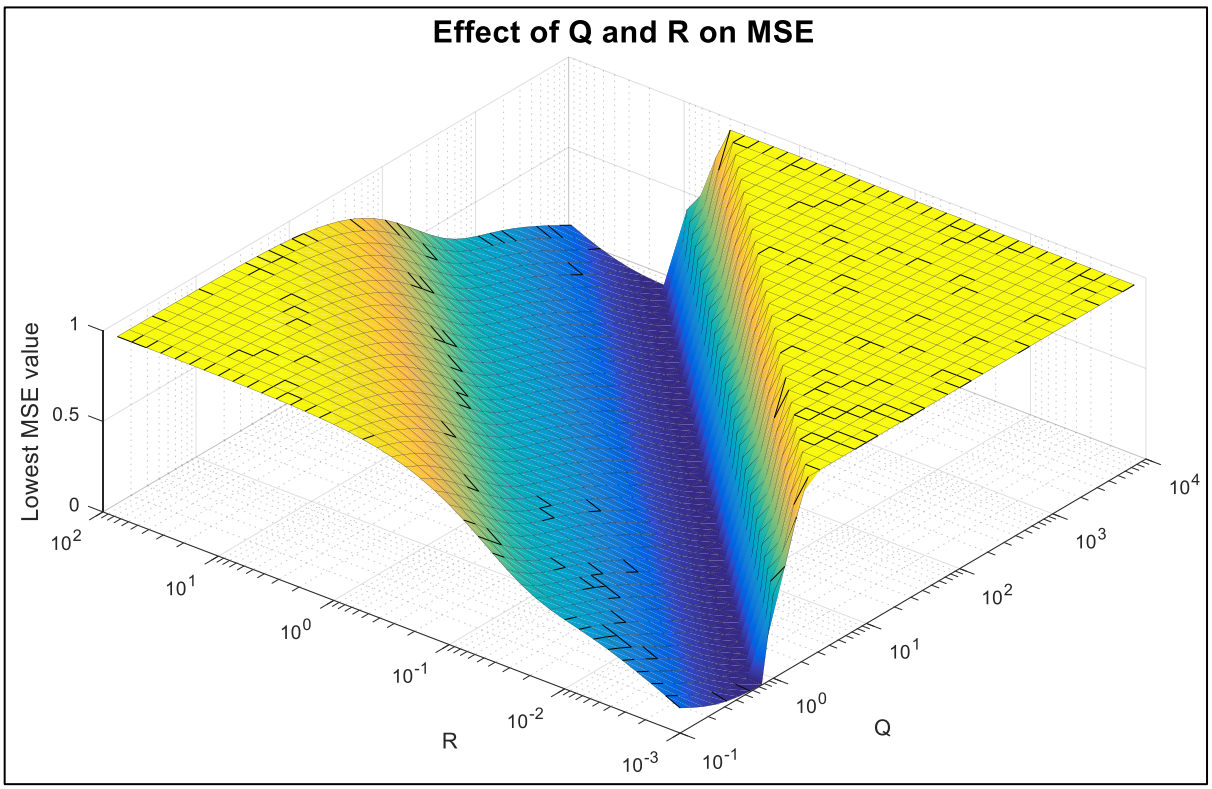


Figure 30: Surface plot of effects of Q and R on lowest MSE value over 1000 trials of F-NOILC

Looking more closely at the break point, Figure 31, shows for the lowest MSE the R and Q values should be chosen to be close to the boundary where it becomes unstable. To quickly stabilise the Z axis ‘Q’ is set to 300 and ‘R’ to 0.5 and for the other two axes ‘Q’ and ‘R’ are 100 and 1 respectively.

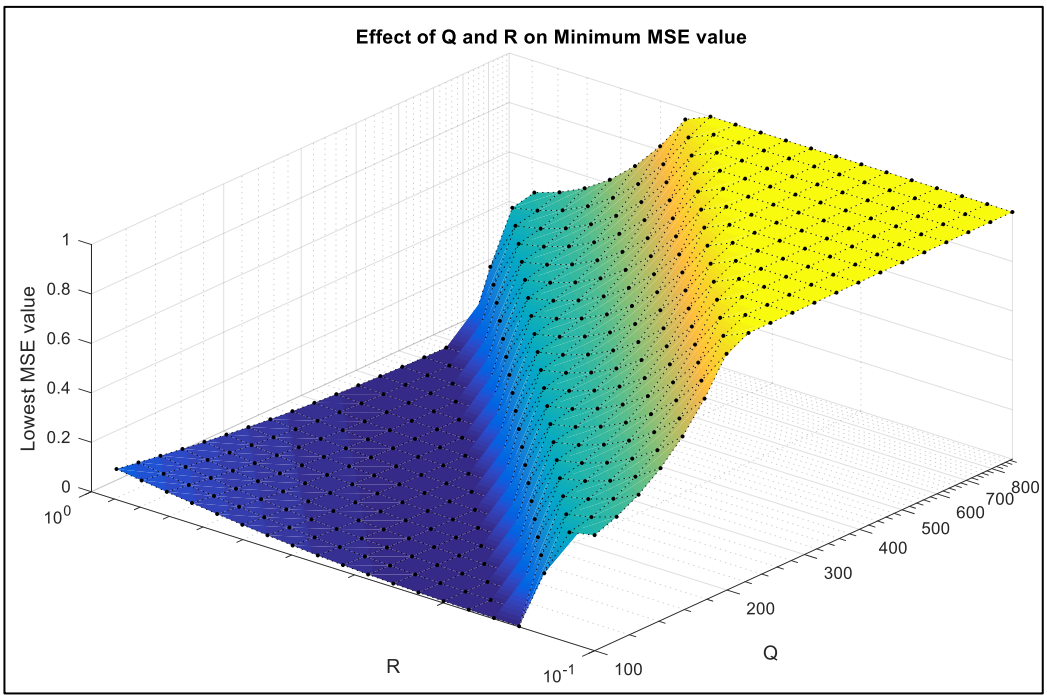


Figure 32: Detail of break point of F-NOILC minimum MSE value against Q and R

Unlike the Phase lead and lag iterative learning control this algorithm does not become unstable as the trials progress which is much more useful. This version of the F-NOILC no long has a large spike in the MSE at the start and the error decreases much faster than previously, shown by Figure 32. The lack of increasing MSE suggests that the algorithm is stable. The 10000 trials carried out does not guarantee the F-NOILC will perform for an infinite amount of trials but shows its long term performance.

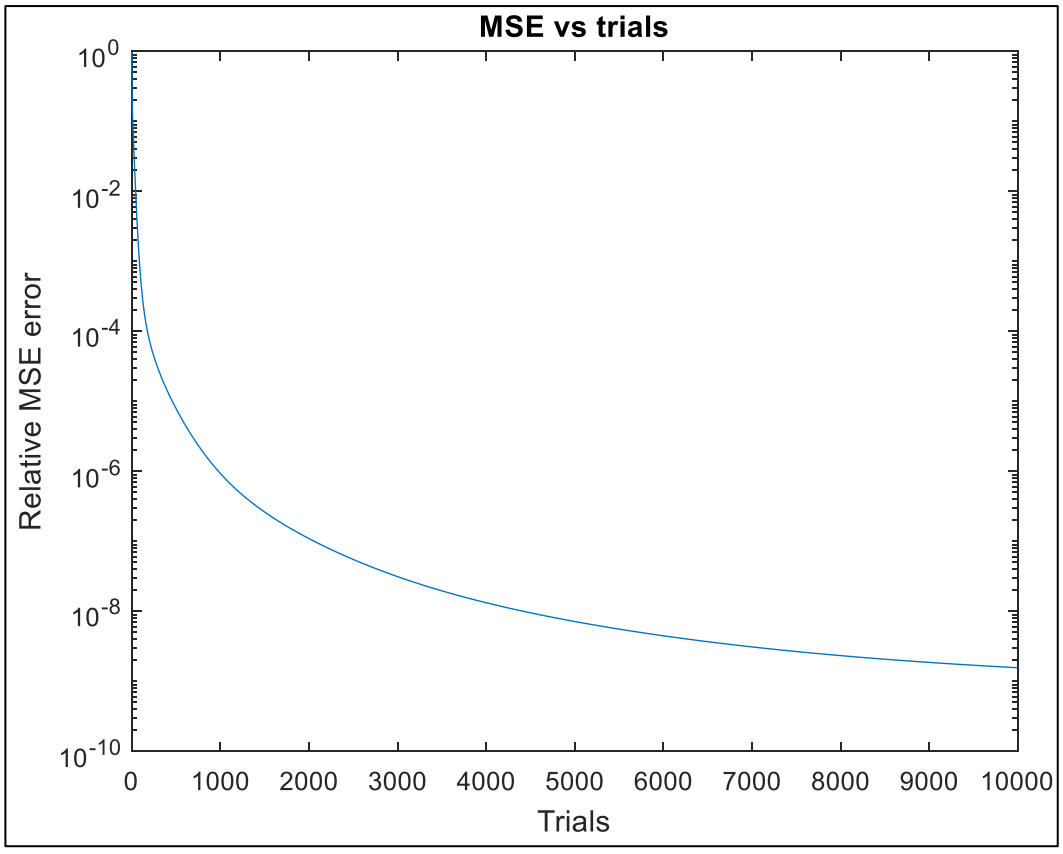


Figure 31: Effect of F-NOILC on MSE value run for 10000 trials on the X axis

To simulate a zeroing error of the robot, an offset is added to the error at the end of each trial and the results for 500 trials on the X axis are shown in Figure 33. The offsets added were ±0.1mm, ±1mm and ±5mm with no offset plotted for reference. The plot shows the offset does not affect the convergence but limits the minimum error which the F-NOILC can reach as there is no way for it to predict the offset in the next trial.

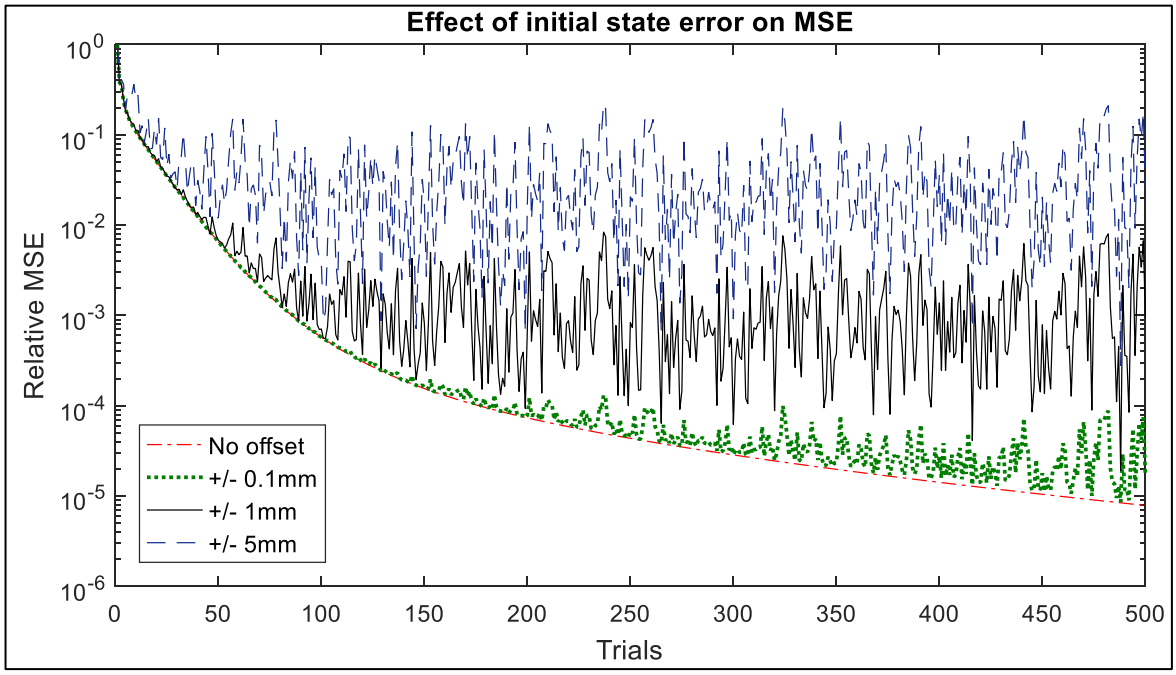


Figure 33: Effect of different maximum offsets on MSE value of F-NOILC for 500 trials

4 Evaluation of Controls

The different control algorithms can be evaluated on three aspects of the performance which are error convergence, long term stability and sensitivity to initial state error. Figure 34 shows how the relative MSE value changes over 10,000 trials for each of the iterative control algorithms for the X axis of the robot.

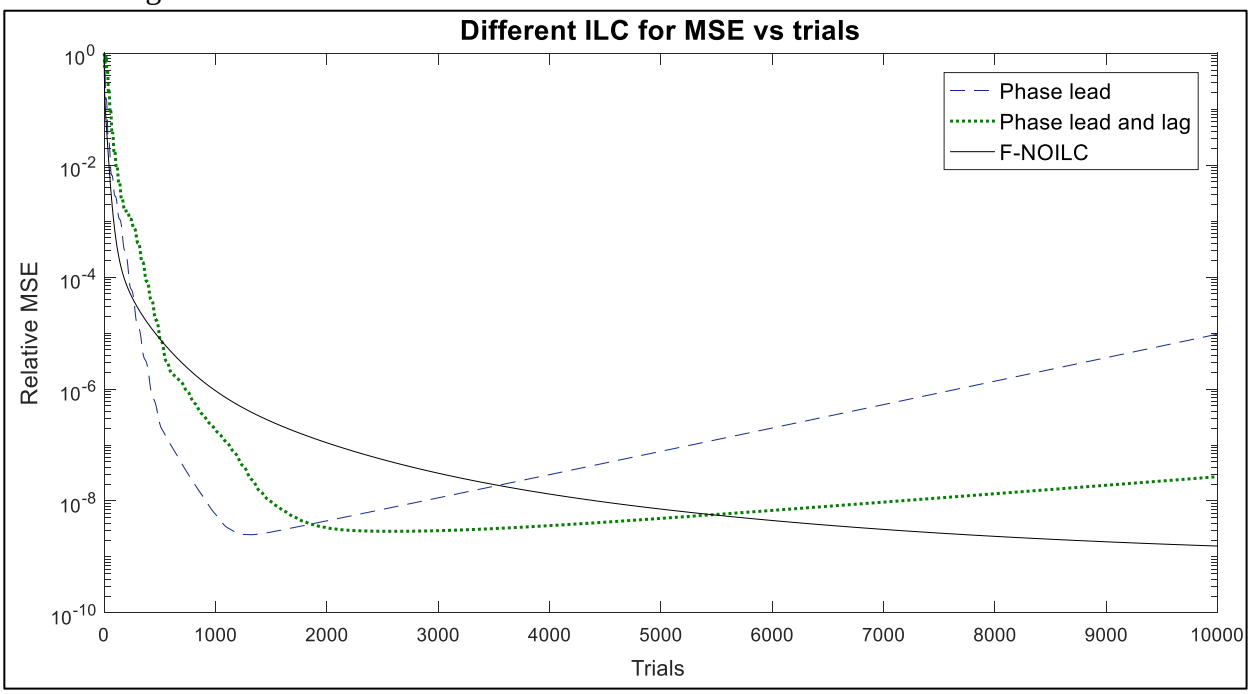


Figure 34: Effect of different ILC on MSE value for 10000 trials on the X axis

Deadbeat control does not adapt with trials and immediately reaches the minimum error from the first run on the robot. This means if it is able to be implemented with a low enough minimum error for the application then Deadbeat will outperform all other tested controllers in error convergence.

As shown the initial error convergence is fastest for the F-NOILC and slowest for Phase lead and lag system. Setting the requirement for adequate converge at an MSE value of 10^{-3}, F-NOILC pass this condition in 86 trials whereas phase lead takes 148 trials, and Phase lead and lag requires 250 trials. F-NOILC has a much faster initial convergence but is passed by the other algorithms before the 1000^{th} trial. However the tuning variables of the Phase lead and Phase lead and lag algorithms are not optimal and the speed of the error convergence can be greatly increased or decreased. Tuning the variables to increase the speed of error convergence does come at a cost, as shown by Figure 19 where an increase in error is induced in the first few trials. Moreover increasing the convergence speed has an effect on the long term performance.

As shown by Figure 34, the iterative learning controls which use the phase shift become unstable and diverge away from the minimum error value as indicated by the increasing MSE value. I was unable to find values of the tuning variables to prevent this increase on MSE value but it does not guarantee there is not a set of variables to control the instability. On the other hand, the F-NOILC algorithm does not show any signs of divergence from the minimum error. This strongly suggests that the long term performance of the F-NOILC is stable. For a gantry robot which could be in continuous use, performing the two second

pick and place task, each week it would perform over 300,000 operations which means the long term stability of the algorithm if very important.

The effects of initial state errors is similar for all three of the iterative learning controls, where the increase of maximum offset from ±1mm to ±5mm forces MSE value to increase by roughly an order of 10. Although the Phase lead algorithm is the simplest, it appears to perform much better than the other ILC algorithms at handling the initial state error. With all three offset magnitudes the average MSE values are much lower for the Phase lead ILC and are of approximately equal value for the other algorithms. When the offsets are added the minimum MSE value reached by the ILC increases, reducing the effectiveness of the algorithms. Figure 35 shows how the effect of the long-term performance break down in Phase lead ILC interacts with the raised MSE due to the initial state errors, plus the Phase lead and lag ILC shows a similar pattern.

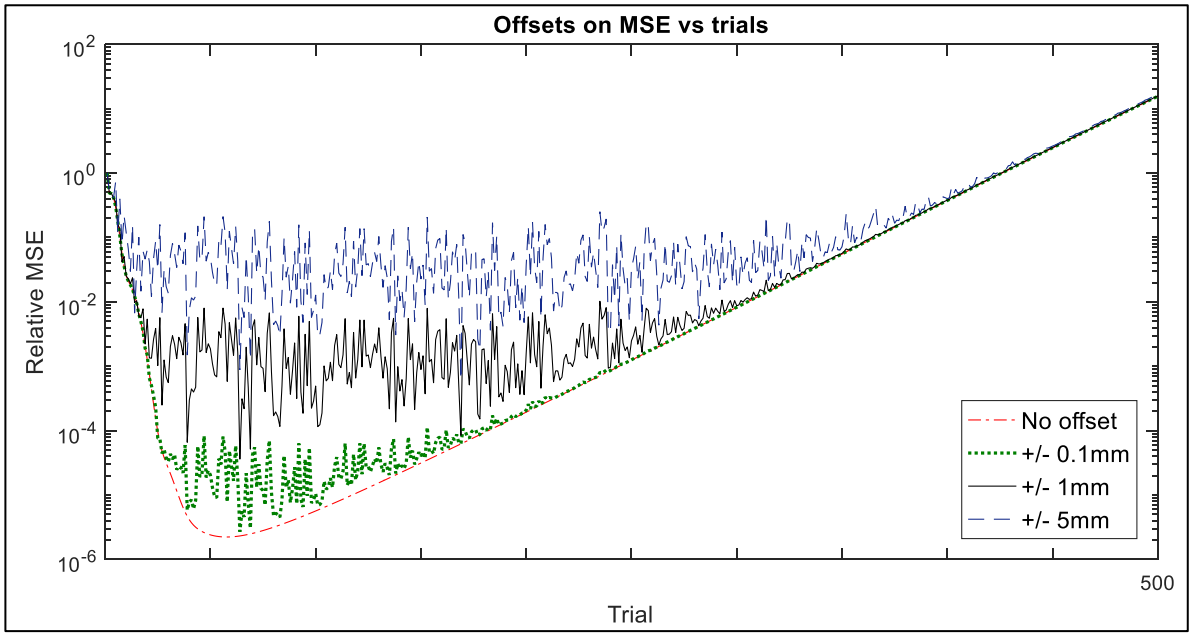


Figure 35: Effect of the long-term performance break down in Phase lead ILC on the raised MSE due to the initial state error

Due to the fact Deadbeat controller has no learning component, the effect of the addition of an initial state error would be a shift in output where the size is governed by the summative value of all the previous starting offsets.

Implementation the each controller requires a different level of complexity. Deadbeat is by far the most challenging as requires an exact model of the systems plant and for you to create the inverse which can be demanding and laborious. Likewise, Deadbeat control requires exact parts to create the controller and could involve a lot of maintenance for the exact inverse plant. For systems where repetitive errors are likely to arise, one of the iterative learning controls would be much better suited.

The Fast – Norm-Optimal Iterative Learning control performs much better in the long-term compared to the other ILC test but requires knowledge of the plant dynamics in order to create the state space model which in turn is used to make the gain matrix and predictive component. Moreover the F-NOILC algorithm can only be realised if the processer of the controller is fast enough and if there is adequate memory. For a system which has a faster

processor but not enough memory then NOILC can be implemented to the same effect, as stated by Ratcliffe et al. [4].

For a system where the plant dynamics cannot be exactly measured then the simpler Phase lead, and Phase lead and lag ILC can be used. Although these ILC algorithms come at the cost of the need to tune the variables of the algorithm which greatly increases in difficulty for the more complex system. The long term performance of these algorithms may cause problems but could be combated by the incorporation of a high frequency filter or even more optimal tuning of the variables. Additionally Phase lead ILC, and Phase lead and lag ILC require much less memory and processing power than F-NOILC.

5 Future Work

To further analyse the differences in the range of iterative learning controls the effect of other types of noise in the system, like random noise disturbances in the signals or repeating errors. Furthermore the effect if the system plant were to change during operation could be investigated to find how exact a model is needed to create a stable algorithm. This will give greater insight into the performance of the controls being tested.

To improve the controls tested an exploration can be carried out into adding extra elements onto the Phase lead and lag control which could help improve the long term performance problem. Furthermore, the Phase lead and lag control could be improved with the addition of an element which stops the error convergence once a certain error is reached and only resumes when the output is sufficiently far away from the desired signal.

The combination of multiple different control systems could provide greater performance. For example if some type of feedback system was added to the ILC to dampen the oscillation which causes the instability in the Phase lead algorithm. Moreover the Deadbeat control could be included in some type of iterative learning control to further its effectiveness.

In terms of designing for the University gantry robot, the ILC methods could be tested on the real robot to show the effectiveness outside of the controlled simulation.

6 Project Management

6.1 Time management

Managing time was vital to completing this project on time. To ensure I had enough time I started work on Matlab and SIMULINK over the summer of 2016 to ensure I could start as soon as possible. As the simulations of each control progressed in complexity, enough time needed to be allocated if problem were to arise. To schedule time a Gantt chart was used to provide a visual representation of the task needed to be completed and time scale. Estimating the time taken to complete tasks was judged on the apparent complexity but as the project progresses and new problems became apparent the timeline shifted. However the initial Gantt chart did not factor in the time off I would need to take to complete my January exams which pushed the schedule back by a month. The Gantt chart operated as a guide for progress which should be made and is shown in Appendix G with the actual timeline of events

6.2 Risks

The project includes risks of varying impact, which need to be identified and precautions need to be made. Due to the nature of this project, completed individually and all on software, all the risks pertain to my personal ability to use Matlab and write this report. The table below shows a risk matrix of the main possibilities which would cause this project not to be completed. The likelihood and impact are both scored on a scale of 1 to 5 and multiplied to provide the risk to the project if no precautions are taken. The risk will then be a value between 1 and 25 where 1-7 is low risk, 8-15 is medium risk and 16-25 is high risk.

Problem	Likelihood	Impact	Risk
Personal issue or illness causing delays	3	3	**9**
Unable to use MATLAB on personal computer	2	4	**8**
Personal computer breaking or loss of all data and completed work	3	5	**15**

Figure 36: Table showing likely problems and risk involved

To reduce these risks I took safety measures for each of the problems.

- Understanding and usage of the compensation system of the University decreases the impact of personal illness.
- The impact of not being able to use MATLAB on my personal computer is reduced as Zepler Labs has computers with Matlab installed.
- To reduce the risk of data loss, all files are saved thrice. On my computer, on a physical external hard drive in my house and in a Google Drive account. Additionally I versioned my code saving new files at regular intervals.

7 Conclusion

This report shows the design and simulation of 3 different ILC algorithms and an inversion control method, demonstrating the different strengths and weaknesses of each. Furthermore it demonstrates the speed and robustness which the ILC algorithms are able to control the three different axes of the gantry robot.

Although Fast Norm-Optimal ILC yields the best long term performance, it will not always be able to be implemented for a system due to the much higher computational power needed. The proportional Phase lead ILC is more ideal where the systems homing mechanism is not very accurate so the effect of initial state error is high. Showing the range of Iterative control algorithms can be implemented in different situations and for varying systems depending on the requirements. Deadbeat provides the fastest control but in the real world is much more complex to implement and has no method for combating any differences between the outputs and desired path which will occur.

For the control of the gantry robot at the University of Southampton, I would recommend the F-NOILC as the plant dynamics is known and the system has enough processing power and memory. Also compared to deadbeat it can be implemented with relative ease and the iterative learning gives it the advantage when handling errors.

References

[1] H.-S. Ahn, C. Y and M. K. L, “Iterative learning control: brief survey and categorization.,” *IEEE Transactions on Systems Man and Cybernetics Part C Applications and Reviews.,* vol. 37, no. 6, p. 1099, 2007.

[2] N. M. A, “Mathworks Matlab and SIMULINK tutorials,” Mathworks, [Online]. Available: https://uk.mathworks.com/support/learn-with-matlab-tutorials.html. [Accessed September 2016].

[3] C. T. Freeman, P. Lewin and E. Rogers, “Experimental evaluation of iterative learning control algorithms for non-minimum phase plants,” *International Journal of Control,* vol. 78, no. 11, pp. 826-846, 2005.

[4] J. D. Ratcliffe, P. L. Lewin, E. Rogers, J. J. Htnen and H. Owens D, “Norm-Optimal Iterative Learning Control Applied to Gantry Robots for Automation Applications,” *IEEE Transactions on Robotics,* vol. 22, no. 6, pp. 1303 - 1307, 2006.

[5] N. Amann, O. D. H and R. E, “Predictiveoptimal iterative learning control,” *International Journal of Control,* vol. 69, no. 2, pp. 203-226, 2010 (online).

[6] N. Amann, D. H. Owens and E. Rogers, “Iterative learning control for discrete-time systems with exponential rate of convergence,” *IEE Proceedings - Control Theory and Applications,* vol. 143, no. 2, pp. 217 - 224, 2002.

Appendix A.

Appendix includes code for proportional code on SHM

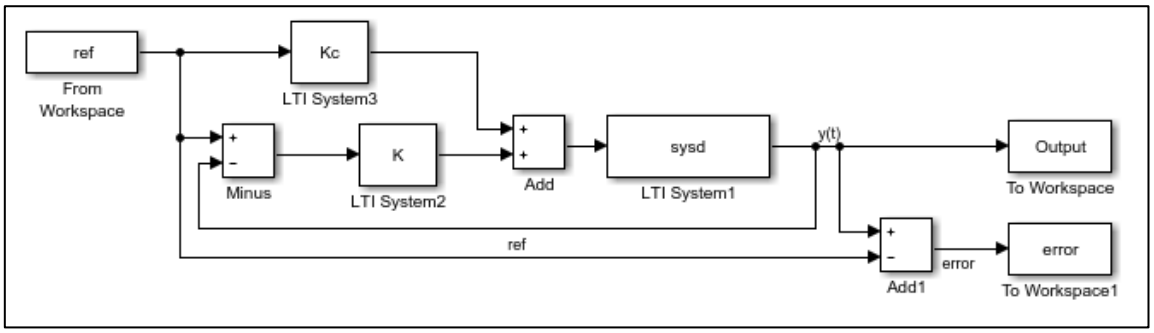
```
%set values of SHM
w=500;
zeta=5;
k=5;

%setup reference and start U
t=[0:0.0001:20]';
x=0*ones(length(t),1);
U=timeseries(x,t);
ref=timeseries(sin(t/3),t);

%simulation parameters
paramNameValStruct.SimulationMode = 'normal';
paramNameValStruct.AbsTol         = '1e-3';
paramNameValStruct.SaveState      = 'on';
paramNameValStruct.StateSaveName  = 'xout';
paramNameValStruct.SaveOutput     = 'on';
paramNameValStruct.OutputSaveName = 'yout';
paramNameValStruct.SaveFormat = 'Dataset';

%trial numbers and lookup trial
trials=100;
LookUp=45;

%loop for trials
for i=1:trials

    %run sinulation
    simOut = sim('shm_ILC',paramNameValStruct);
    %get output and input data
    outputs = simOut.get('Yout');
    Outputs =outputs.get('Data');
    Unext = simOut.get('Unext');
    Unow = U.get('Data');
    g = i*ones(length(Outputs),1);

    %save data to matrices
    if i==1
        Time =(outputs.get('Time'));
        T=Time;
        Yout=[Outputs];
        Trial=[g];
        Ucurrent=[Unow];
        Lookout = -pi;
        Looktime = pi;
    else
        Yout=[Yout Outputs];
        Trial=[Trial g];
        T=[T,Time];
        Ucurrent=[Ucurrent,Unow];
    end

    %save data of lookup trial
    if i==LookUp
        Lookout = Outputs;
        Looktime = Time;
    end

    %set next U
    U=Unext;

end
```

```
%plot of output setting view and lables
figure
surface(Trial,T,Yout,'MeshStyle','column')
view(3)
set(gca,'ydir','reverse')
title('Output against time for the different trial');
xlabel('Trial number');
ylabel('Time(s)');
zlabel('Amplitude');

%plot of input setting view and labels
figure
surface(Trial,T,Ucurrent,'MeshStyle','column')
view(3)
set(gca,'ydir','reverse')
title('Input against time for the different trial');
xlabel('Trial number');
ylabel('Time(s)');
zlabel('Amplitude');

%plot of looked up trial
figure
plot(Looktime,Lookout);
title(['Output of trial:',' ',num2str(LookUp)]);
xlabel('Time(s)');
ylabel('Amplitude');
```

Appendix B.

```
%load system and reference
load('trajectories_30upm_1KHz_no_offset.mat')
load('z-axis_3rd_order_model.mat')

%print transfer fn
format compact
sysc

%steps for time
t_step=0.001;

%convert to descrete time domain
sysd = c2d(sysc,t_step,'zoh');

%set z as descrete time variable
z=zpk('z',t_step);
inverse_z=1/z;

%set upt ref signal
t=z_profile_30upm(:,1);
profile=z_profile_30upm(:,2);
ref=timeseries(profile,t);

%set up Kc with sufficent delays
Kc=inverse_z*(1/(sysd));

%set up K
K=1/(z*(z+0.5));

%run for when Kc is not adjusted
%simulation
sim('discrete_deadbeat');

%simulation data
Time(:,1)=Output.Time;
Yout(:,1)=Output.Data;
Errors(:,1)=error.Data;

%plot of output
figure
plot(Time,Yout,'b:',ref.time,ref.data,'r--')
title('Ouput & reference for Kc=1/G')
xlabel('Time(s)');
ylabel('Amplitude');

%plot of error
figure
plot(Time,Errors)
title('Error for Kc=1/G')
xlabel('Time(s)');
ylabel('Amplitude');


%fix Kc removing unstable pole and adding plot at z=-1
Kc=zpk(sysd.P,[0,-1,sysd.Z{1,1}(2,1)],1/(sysd.K),t_step);

%run for when Kc is adjusted
%simulation
sim('discrete_deadbeat');

%simulation data
Time(:,1)=Output.Time;
```

```
Yout(:,1)=Output.Data;
Errors(:,1)=error.Data;

%plot of output
figure
plot(Time,Yout,'b:',ref.time,ref.data,'r--')
title('Ouput & reference for changed Kc')
xlabel('Time(s)');
ylabel('Amplitude');

%plot of error
figure
plot(Time,Errors)
title('Error for changed Kc')
xlabel('Time(s)');
ylabel('Amplitude');
```



Simulink model used for deadbeat control

Appendix C.

Appendix includes code for phase lead ILC

```
%load model and references
load('trajectories_30upm_100Hz_no_offset.mat')
load('x-axis_7th_order_model.mat')
%print model (to visually quickly see correct model)
sysc

%initialisation of reference into a timeseries
t=x_profile_30upm(:,1);
z=x_profile_30upm(:,2);
ref=timeseries(z,t);

%initial U setting
t=0:0.001:1.99;
t=t';
x=(ones(length(t),1))/200;
Uin_timeseries=timeseries(x,t);

%set simulation paramerters
paramNameValStruct.SimulationMode = 'normal';
paramNameValStruct.AbsTol         = '1e-3';
paramNameValStruct.SaveState      = 'on';
paramNameValStruct.StateSaveName  = 'xout';
paramNameValStruct.SaveOutput     = 'on';
paramNameValStruct.OutputSaveName = 'yout';
paramNameValStruct.SaveFormat = 'Dataset';

%-------------------------------------------------------
%number of trials, look up trail and other variables
trials=1000;
LookUp=500;
lambda=0.01;
gain=4;
x=5;
maxoffset=0.001;
%-------------------------------------------------------

%set up matrices for interval figure
interval=zeros(1,x);
int_out = zeros(1991,x);
int_time = zeros(1991,x);
int_in = zeros(1991,x);
for i=1:x
   interval(i)=(trials/x)*i;
end
y=1;

%preallocation of varrialbes used to store data to save simulation time
Trials=zeros(1991,trials);
Time=zeros(1991,trials);
Yout=zeros(1991,trials);
Ucurrent=zeros(1991,trials);
Errors=zeros(1991,trials);
mse_err=zeros(1,trials);
Offsets=zeros(1,trials);

%loop for trials
for i=1:trials

    %seed random number gen with trial number
    rng(i)
    %set offset between bounds with random number
    offset=(maxoffset*2*rand)-maxoffset;
```

```
    %store offset data
    Offsets(1,i)=offset;

    %run simulation
    simOut = sim('z_axis_ILC',paramNameValStruct);

    %output and input extraction
    outputs_timeseries = simOut.get('Yout');
    outputs_array =outputs_timeseries.get('Data');
    Unow_timeseries=simOut.get('Unow');
    Unow_array = Unow_timeseries.get('Data');
    %form trial and time array
    trial_array = i*ones(length(outputs_array),1);
    time_array =(outputs_timeseries.get('Time'));
    %get error and add offset
    e_timeseries = simOut.get('error');
    e_array=e_timeseries.get('Data')+offset;

    %arrays into matrix to plot
    Trials(:,i)=trial_array;
    Time(:,i)=time_array;
    Yout(:,i)=outputs_array;
    Ucurrent(:,i)=Unow_array;
    Errors(:,i)=e_array;

    %run error through phase lead and gain
    e_timeseries.time=e_timeseries.time-lambda;
    e_timeseries.data=(e_timeseries.data)*gain;

    %create Unext with 2nd simulation
    SIMOUT = sim('addingsimulationusedinlead',paramNameValStruct);
    %get U next from data
    Uin_timeseries=SIMOUT.get('out');

    %save looked up trial data
    if i==LookUp
        Lookout = outputs_array;
        Looktime = time_array;
        Lookin = Unow_array;
    end

    %save interval data
    if i==interval(y)
        int_out(:,y) = outputs_array;
        int_time(:,y) = time_array;
        int_in(:,y) = Unow_array;
        y=y+1;
    end

    %find initial MSE value and set array to 1 for trial 1 to make
relative
    if i==1
        err_start=immse(zeros(length(t),1),e_array);
        mse_err(1,i)=1;
    %find and add other relative mse values
    else
        err_hold=immse(zeros(length(t),1),e_array);
        mse_err(1,i)=err_hold/err_start;
    end

end


%surface plot of output against time against trial
figure
%do not show lines along trials for >600 trials and set view of surface
if trials>600
```

```
    surface(Trials,Time,Yout,'MeshStyle','none')
else
    surface(Trials,Time,Yout,'MeshStyle','column')
end
view(3)
set(gca,'ydir','reverse')
%add labels to plot
title('Output against time for the different trial');
xlabel('Trial number');
ylabel('Time(s)');
zlabel('Amplitude');

%surface plot of input against time against trial
figure
%do not show lines along trials for >600 trials and set view of surface
if trials>600
    surface(Trials,Time,Ucurrent,'MeshStyle','none')
else
    surface(Trials,Time,Ucurrent,'MeshStyle','column')
end
view(3)
set(gca,'ydir','reverse')
%add labels to plot
title('Input against time for the different trial');
xlabel('Trial number');
ylabel('Time(s)');
zlabel('Amplitude');

%surface plot of input against time against trial
figure
%do not show lines along trials for >600 trials and set view of surface
if trials>600
    surface(Trials,Time,Errors,'MeshStyle','none')
else
    surface(Trials,Time,Errors,'MeshStyle','column')
end
view(3)
set(gca,'ydir','reverse')
%add labels
title('Error against time for the different trial');
xlabel('Trial number');
ylabel('Time(s)');
zlabel('Amplitude');

%plot of lookup in
if LookUp==(-1)
    disp('No trial looked up')
else
    %plot of lookup input
    figure
    plot(Looktime,Lookin)
    title('Input of looked up trial')
    xlabel('Time(s)');
    ylabel('Amplitude');
    %plot of lookup out and reference
    figure
    plot(Looktime,Lookout,'b:',ref.time,ref.data,'r--')
    title(['Ouput & ref of trial ', num2str(LookUp)])
    xlabel('Time(s)');
    ylabel('Amplitude');
end

%plot of intervals
figure
for y=1:x
    %plot of input on top row
    subplot(2,x,y)
    plot(int_time(:,y),int_in(:,y))
```

```
    title(['Input of trial ', num2str(interval(y))])
    xlabel('Time(s)');
    ylabel('Amplitude');
    %plot of output and ref below
    subplot(2,x,(y+x))
    plot(int_time(:,y),int_out(:,y),'b',ref.time,ref.data,'r--')
    title(['Ouput & ref of trial ', num2str(interval(y))])
    xlabel('Time(s)');
    ylabel('Amplitude');
end

%plot MSE vs trails on semi log(y) graph
figure
semilogy(Trials(1,:),mse_err)
title('MSE vs trials')
xlabel('Trials');
ylabel('Relative MSE error');
```

Appendix D.

Appendix includes additions to phase lead ILC code to make phase lead and lag ILC code

{added additional variables at the start}

```
%number of trials, look up trail and other variables
trials=10000;
LookUp=3000;
lambda=0.0015;
sigma=0.005;
k1=5;
k2=0.5;
x=5;
maxoffset=0.0001;
```

{inside for loop of trials}

```
    %run error through phase lead and gain
    e1_timeseries=e_timeseries;
    e2_timeseries=e_timeseries;
    e1_timeseries.time=e1_timeseries.time-lambda;
    e1_timeseries.data=(e1_timeseries.data)*k1;
    e2_timeseries.time=e2_timeseries.time+sigma;
    e2_timeseries.data=(e2_timeseries.data)*k2;
```

Appendix E.

Appendix includes 1st attempt code of F-NOILC

```
%load system and reference
load('z-axis_3rd_order_model.mat')
load('trajectories_30upm_100Hz_no_offset.mat')
%print transfer fn
format compact
sysc

%convert to descrete time domain
sysd = c2d(sysc,0.001,'zoh');
%convert to state space
sys=ss(sysd);

n=size(sys.A);
n=n(1,1);

%initialisation of reference into a timeseries
{omitted as identical to previous}

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%number of trials
trials=100;
x=5;
Lookup=149;
LookUp=round(trials/2);

%set cost function values
q=0.1*(10^15);
r=0.00000001*(10^5);
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
disp(q)
disp(r)

%initial U setting and ksi
t=0:0.001:1.99;
t=t';
one_length_t=(ones(length(t),1));
zero_length_t=one_length_t*0;
Uin_timeseries=timeseries(one_length_t*0,t);
U_timeseries=timeseries(one_length_t*0,t);
one_length_t_3_wide=ones(length(t),3);
ksi_old_matrix=one_length_t_3_wide;
ksi_new_matrix=one_length_t_3_wide;
x_old_timeseries=timeseries((one_length_t_3_wide(:,1:3)*0),t);

%initialise interval print
{omitted as identical to previous}
%preallocation of varrialbes used to store data to save time
{omitted as identical to previous}

%set Q,R,S,E
Q=q*eye(1);
R=r*eye(1);
S=0*sys.B;
E=eye(n);
Qn=(sys.C)'*Q*(sys.C);

%descreate Riccatti eqn
[K,L,Gain]=dare(sys.A,sys.B,Qn,R,S,E);

%pre trials calcualtions
alpha=inv(eye(n)+K*(sys.B)*(1/R)*(sys.B'));
beta=alpha*(sys.A');
```

```
gamma=alpha*(sys.C')*Q;
omega=(1/R)*(sys.B');
lambda=inv((sys.B'*K*sys.B)+R)*(sys.B')*K*(sys.A);

%loop for iterations
for i=1:trials
    disp(i)
    %run simulation
    simOut = sim('ss_feedback',paramNameValStruct);

    %output and input extraction
    outputs_timeseries = simOut.get('Yout');
    outputs_array =outputs_timeseries.get('Data');
    Unow_timeseries=simOut.get('Unow');
    Unow_array = Unow_timeseries.get('Data');
    %form trial and time array
    trial_array = i*ones(length(outputs_array),1);
    time_array =(outputs_timeseries.get('Time'));
    %error
    e_timeseries = simOut.get('error');
    e_array=e_timeseries.get('Data');
    %states
    x_current_timeseries=simOut.get('States');

    %arrays into matrix to plot
    Trials(:,i)=trial_array;
    Time(:,i)=time_array;
    Yout(:,i)=outputs_array;
    Ucurrent(:,i)=Unow_array;
    Errors(:,i)=e_array;

    %manipulate error and old ksi for new ksi and Unext for each value
of t
    for l=length(t):-1:1
        %get ksi old from matrix
        ksi_old=[ksi_old_matrix(l,1);ksi_old_matrix(l,2);ksi_old_matrix
        (l,3)];
        %set ksi new using equation
        if l==length(t)
            ksi_new=[0;0;0];
        else
            ksi_new=beta*ksi_old+gamma*e_timeseries.data(l);
        end
        %get single time states from matrix
        state_current=[x_current_timeseries.data(l,1);x_current_timeser
        ies.data(l,2);x_current_timeseries.data(l,3)];
        state_old=[x_old_timeseries.data(l,1);x_old_timeseries.data(l,2
        );x_old_timeseries.data(l,3)];
        %update new U with equation
        U_timeseries.data(l)=Unow_timeseries.data(l)-
        lamda*(state_current-state_old)+omega*ksi_new;
        %add ksi new to matrix format ready to be passed to ksi old
        matrix
        ksi_new_matrix(l,1)=ksi_new(1,1);
        ksi_new_matrix(l,1)=ksi_new(2,1);
        ksi_new_matrix(l,1)=ksi_new(3,1);
    end

    %pass new ksi to old ksi (matrix)
    ksi_old_matrix=ksi_new_matrix;
    %pass new x to old x (timeseries)
    x_old_timeseries=x_current_timeseries;
    %pass new input to old input (timeseries)
    Uin_timeseries=U_timeseries;

    %find lookup trial and save results
    {omitted as identical to previous}
    %find interval trials and save results
```

```
    {omitted as identical to previous}

    if i==1
        err_start=immse(zero_length_t,e_array);
        mse_err(1,i)=1;
    else
        err_hold=immse(zero_length_t,e_array);
        mse_err(1,i)=err_hold/err_start;
    end
end

%plotting

%plot of mse along trials
figure
plot(Trials(1,:),mse_err)
title('MSE vs trials')
xlabel('Trials');
ylabel('Relative MSE error');

%surface plot of output against time against trial
{omitted as identical to previous}
%surface plot of input against time against trial
{omitted as identical to previous}
%surface plot of error against time against trial
{omitted as identical to previous}
%plot of lookup
{omitted as identical to previous}
%plot of intervals
{omitted as identical to previous}
```

Appendix F.

Appendix includes 2nd attempt code of F-NOILC

```
%clear previous variables in workspace
clear

%format style of text in command window
format compact

%load reference signals
load('trajectories_30upm_1KHz_no_offset.mat')
%set which axis to run in simulation and load model and desired
trajectories
Axis=input('Which axis? [X],[Y]or[Z]: ','s');
if Axis=='X'
    load('x-axis_7th_order_model.mat')
    profile=x_profile_30upm(:,2);
elseif Axis=='Y'
    load('y-axis_3rd_order_model.mat')
    profile=y_profile_30upm(:,2);
else
    load('z-axis_3rd_order_model.mat')
    profile=z_profile_30upm(:,2);
end

%print transfer fn for visual check
sysc

%choose if want to vary Q and R
Vary = input('Do you want to vary Q and R values? [Y] or [N]:','s');
if Vary=='N'
    %set if you want plots to be made
    Plot = input('Do you want plot for each Q and R value? [Y] or
[N]:','s');
    if isempty(Plot)
        Plot= 'N';
    end
    iqmax=1;
    irmax=1;
elseif Vary=='Y'
    %dont plot figures for each Q and R
    Plot= 'N';
    %values for vary q and r (range depends on size of ir and iq)
    iqmax=input('iq maximum:');
    irmax=input('ir maximum:');
end

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%set mex offset size (can set to 0 for no offset)
maxoffset=input('Max Offset:');
%number of trials, intervals and lookup (inputs)
trials=input('Number of trials:');
x=input('Internal capture splits:');
LookUp=input('Look up iteration number:');
%steps for time in simulation
t_step=0.001;
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

%initialise mse matries with q and r matries
mse_error_global_end=ones(iqmax,irmax);
mse_error_global_mean=ones(iqmax,irmax);
mse_error_global_min=ones(iqmax,irmax);
mse_error_global_trial1=ones(iqmax,irmax);
mse_error_global_trial5=ones(iqmax,irmax);
mse_error_global_trial20=ones(iqmax,irmax);
mse_error_global_trial100=ones(iqmax,irmax);
```

```
qvalues=ones(iqmax,irmax);
rvalues=ones(iqmax,irmax);

%convert to descrete time domain
sysd = c2d(sysc,t_step,'zoh');
%convert to state space
sys=ss(sysd);
%set n (size of matrix [3 for z and y and 7 for x])
n=length(sys.A);

%set A,b,c,d
A=sys.A;
B=sys.B;
C=sys.C;
D=sys.D;

%initialisation of reference into a timeseries
t=x_profile_30upm(:,1);
ref=timeseries(profile,t);

%two loops for varying Q and R
for iq=1:iqmax
for ir=1:irmax

    %start timer
    tic

    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    %set cost function values dependent for varying Q and R
    if Vary=='Y'
        %set Q and R for current iq and ir if set to vary
        q=10*(10^((iq-1)/2));
        r=0.1*(10^((ir-1)/2));
        %print value out to visual see progress in program
        disp(['iq = ', num2str(iq),' & ir = ', num2str(ir),' so Q = ',
num2str(q),' & R = ', num2str(r)])
    %set Q and R if not varying (set to specific values for each axis)
    elseif Vary=='N'
        if Axis=='Z'
            q=300;
            r=0.5;
        elseif Axis=='Y'
            q=200;
            r=1;
        else
            q=100;
            r=1;
        end
    end
    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    %initialize timeseries for Uin, U, e, states, k and ksi
    t=0:t_step:1.99;
    t=t';
    one_length_t=(ones(length(t),1));
    zero_length_t=one_length_t*0;
    Uin_timeseries=timeseries(one_length_t*0,t);
    e_timeseries=timeseries(one_length_t*0,t);
    U_timeseries=timeseries(one_length_t*0,t);
    Simulation_X_previous=timeseries((ones(length(t),n)*0),t);
    Simulation_X=timeseries((ones(length(t),n)*0),t);
    K=0*ones(n,n,length(t));
    ksi_old_matrix=0*ones(n,1,length(t));
    ksi_new_matrix=0*ones(n,1,length(t));

    %preallocation of varrialbes used to store data (to save simulation
time)
```

```
    Trials=zeros(length(t),trials);
    Time=zeros(length(t),trials);
    Yout=zeros(length(t),trials);
    Ucurrent=zeros(length(t),trials);
    Errors=zeros(length(t),trials);
    mse_err=zeros(1,trials);
    Offsets=zeros(1,trials);

    %initialise interval print matrix
    interval=zeros(1,x);
    int_out = zeros(length(t),x);
    int_time = zeros(length(t),x);
    int_in = zeros(length(t),x);
    y=1;
    for i=1:x
       interval(i)=(trials/x)*i;
    end

    %set Q,R matrix
    Q=q*eye(1);
    R=r*eye(1);

    %assign K values with loop that runs backwards in time from t_final
to 0
    for l=length(t):-1:1
        %set terminal value of K as zeros
        if l==length(t)
            K_hold=zeros(n,n);
            K(:,:,l)=K_hold;
        else
            %use equation to find K(t) from K(t+1)
            K_current=(A'*K_hold*A) + (C'*Q*C) - ( (A'*K_hold*B) *
(inv(B'*K_hold*B + R)) *(B'*K_hold*A));
            K_hold=K_current;
            K(:,:,l)=K_hold;
        end
    end
    %visual check to show K values set
    disp('K values set')

    %assign alpha,beta,gamma,omega and lamda
    alpha=inv(eye(n)+K(:,:,1)*(B)*(1/R)*(B'));
    beta=alpha*(A');          %#ok<MINV>
    gamma=alpha*(C')*Q;       %#ok<MINV>
    omega=(1/R)*(B');
    lamda=(inv((B'*K(:,:,1)*B)+R))*(B')*K(:,:,1)*(A);
    %visual check to show values set
    disp('all other pre-calculations done')

    %loop for iterations
    for i=1:trials

        %seed random number generator with trial number
        rng(i)
        %set offset from psudorandom number withing max offset bounds
        offset=(maxoffset*2*rand)-maxoffset;
        %add offset to array to save them
        Offsets(1,i)=offset;

        %run correct simulink simulation for correct axis
        if Axis=='X'
            sim('ss_discrete_x_1KHz');
        else
            %as z and y both same size can run on same simulation
            sim('ss_discrete_z_1KHz');
        end
```

```
        %get error data from simulation and add offset to whole error
        e_array=Simulation_e.Data+offset;
        %loop to calc predictive component (runs backwards from t_final
to 0)
        for l=length(t):-1:1
            %set terminal condition of ksi
            if l==length(t)
                ksi_previous=zeros((n),1);
            %set ksi using eqn
            else
                ksi_previous=(beta*ksi_current)+(gamma*e_array(l+1));
            end
            %pass ksi(t) to ksi(t+1) to be used on next round of loop
            ksi_current=ksi_previous;
            %add worked out ksi to matrix of new ksi values
            ksi_new_matrix(:,:,l)=ksi_current;
        end

        %loop to calc Uin for next iteration
        for l=1:1:length(t)
            %get single time states from simulation state matrix
            state_current=Simulation_X.Data(l,:);
            %get single time states from previous state matrix
            state_old=Simulation_X_previous.Data(l,:);
            %update to new U with equation
            Uin_timeseries.data(l)=Simulation_Unow.Data(l)-
(lamda*(state_current'-state_old'))+(omega*ksi_new_matrix(:,:,l));
        end
        %pass new state data to old state data (timeseries)
        Simulation_X_previous=Simulation_X;

        %find initial MSE value if trial 1 to make all MSE relative and
set to 1 in array
        if i==1
            err_start=immse(zero_length_t,e_array);
            mse_err(1,i)=1;
        %set MSE value for trial and add to MSE array
        else
            err_hold=immse(zero_length_t,e_array);
            mse_err(1,i)=err_hold/err_start;
        end

        %arrays into matrix to plot for trial number, time, output,
input and errors
        Trials(:,i)=i*ones((1.99/t_step)+1,1);
        Time(:,i)=Simulation_Y.Time;
        Yout(:,i)=Simulation_Y.Data;
        Ucurrent(:,i)=Simulation_Unow.Data;
        Errors(:,i)=e_array;

    end

    %print iterations completed and end MSE as visual check
    disp(['iteration number ',num2str(i),'   MSE: ',
num2str(mse_err(1,i))]);
    %end timer and prints time taken to complete
    toc
    disp(' ');

    %PLOTTING of all figures if set to do so
    if Plot=='Y'
        %surface plot of output against time against trial
        figure
        %sets if more than 500 trials to not add lines along trials of
surface plot
        if trials>500
            surface(Trials,Time,Yout,'MeshStyle','none')
        else
```

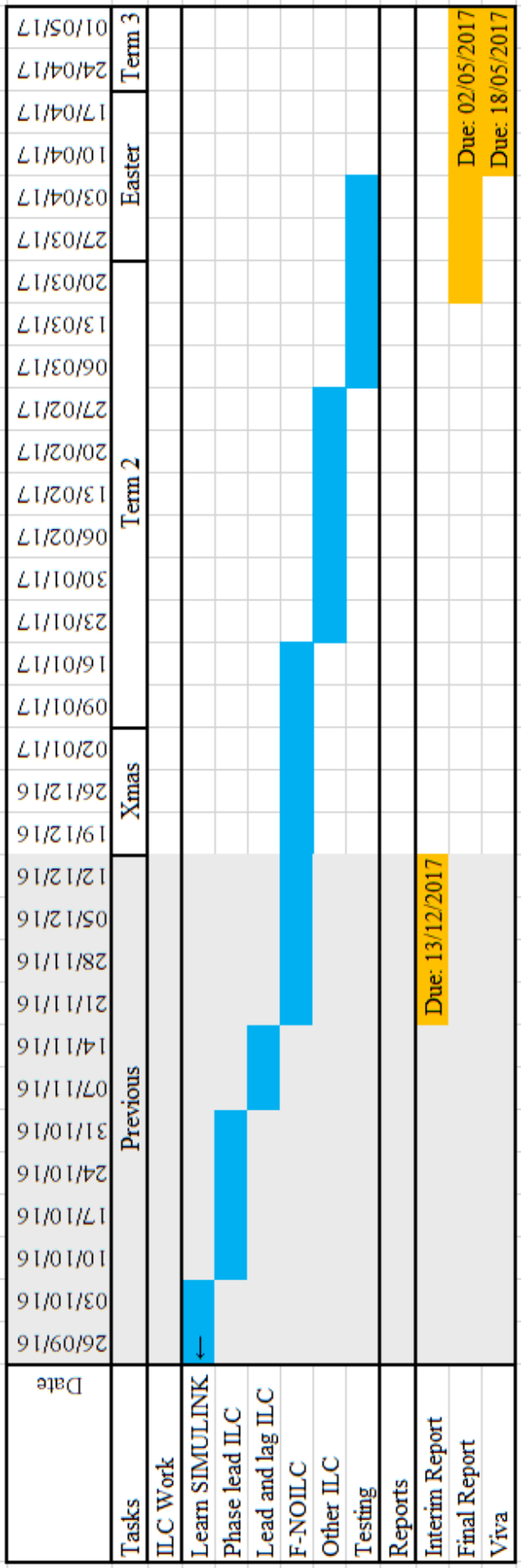
```
            surface(Trials,Time,Yout,'MeshStyle','column')
        end
        %set view to be at angle and reverse y axis direction to make
plot look better
        view(3)
        set(gca,'ydir','reverse')
        %add labels on plot
        title('Output against time for the different trial');
        xlabel('Trial number');
        ylabel('Time(s)');
        zlabel('Amplitude');

        %surface plot of input against time against trial
        figure
        %sets if more than 500 trials to not add lines along trials of
surface plot
        if trials>500
            surface(Trials,Time,Ucurrent,'MeshStyle','none')
        else
            surface(Trials,Time,Ucurrent,'MeshStyle','column')
        end
        %set view to be at angle and reverse y axis direction to make
plot look better
        view(3)
        set(gca,'ydir','reverse')
        %add labels on plot
        title('Input against time for the different trial');
        xlabel('Trial number');
        ylabel('Time(s)');
        zlabel('Amplitude');

        %surface plot of error against time against trial
        figure
        if trials>500
        %sets if more than 500 trials to not add lines along trials of
surface plot
            surface(Trials,Time,Errors,'MeshStyle','none')
        else
            surface(Trials,Time,Errors,'MeshStyle','column')
        end
        %set view to be at angle and reverse y axis direction to make
plot look better
        view(3)
        set(gca,'ydir','reverse')
        %sets to only vary colour of surface between values
        caxis([-0.001,0.001]);
        %add  labels on plot
        title('Error against time for the different trial');
        xlabel('Trial number');
        ylabel('Time(s)');
        zlabel('Amplitude')

        %plot of lookup in
        if LookUp<=(0)
            disp('No trial looked up')
        else
            %find lookup trial and save results
            Lookout = Yout(:,LookUp);
            Looktime = Time(:,LookUp);
            Lookin = Ucurrent(:,LookUp);
            %plot of lookup input
            figure
            plot(Looktime,Lookin)
            %add labels to plot
            title(['Intput from trial ', num2str(LookUp)])
            xlabel('Time(s)');
            ylabel('Amplitude');
            %plot of lookup out and reference
```

```
            figure
            plot(Looktime,Lookout,'b',ref.time,ref.data,'r--')
            %add labels to plot
            title(['Output from trial ', num2str(LookUp)])
            xlabel('Time(s)');
            ylabel('Amplitude');
        end

        %plot of interval figure
        figure
        %find interval trials and save results
        for i=1:x
                int_out(:,i) = Simulation_Y.Data;
                int_time(:,i) = Simulation_Y.Time;
                int_in(:,i) = Simulation_Unow.Data;
        end
        %plot interval values in subplot system
        for y=1:x
            %input on 1st row of figure
            subplot(2,x,y)
            plot(int_time(:,y),int_in(:,y))
            title(['Input of trial ', num2str(interval(y))])
            xlabel('Time(s)');
            ylabel('Amplitude');
            %output (blue) and ref (red dashed) on 2nd row
            subplot(2,x,(y+x))
            plot(int_time(:,y),int_out(:,y),'b',ref.time,ref.data,'r--
')
            title(['Y(t)&Ref of trial ', num2str(interval(y))])
            xlabel('Time(s)');
            ylabel('Amplitude');
        end

        %plot MSE vs trails on semi log(y) graph
        figure
        semilogy(Trials(1,:),mse_err)
        %add labels to plot
        title('MSE vs trials')
        xlabel('Trials');
        ylabel('Relative MSE error');
    end

    %set end, mean and minimum MSE matrix values
    mse_error_global_end(iq,ir)=mse_err(trials);
    mse_error_global_mean(iq,ir)=mean(mse_err);
    mse_error_global_min(iq,ir)=min(mse_err);

    %set values of r and q in matrix
    qvalues(iq,ir)=q;
    rvalues(iq,ir)=r;

end
end

%plot of surface plot for varying q and r
if Vary=='Y'
    %cap mse values at start mse
    mse_error_global_end_capped=min(mse_error_global_end,1);
    mse_error_global_mean_capped=min(mse_error_global_end,1);

    %plot capped end MSE value over trials for each Q and R
    figure

surface(qvalues,rvalues,mse_error_global_end_capped,'FaceColor','interp
','MeshStyle','both','LineStyle',':','Marker','.','MarkerFaceColor','k'
)
    view(3)
```

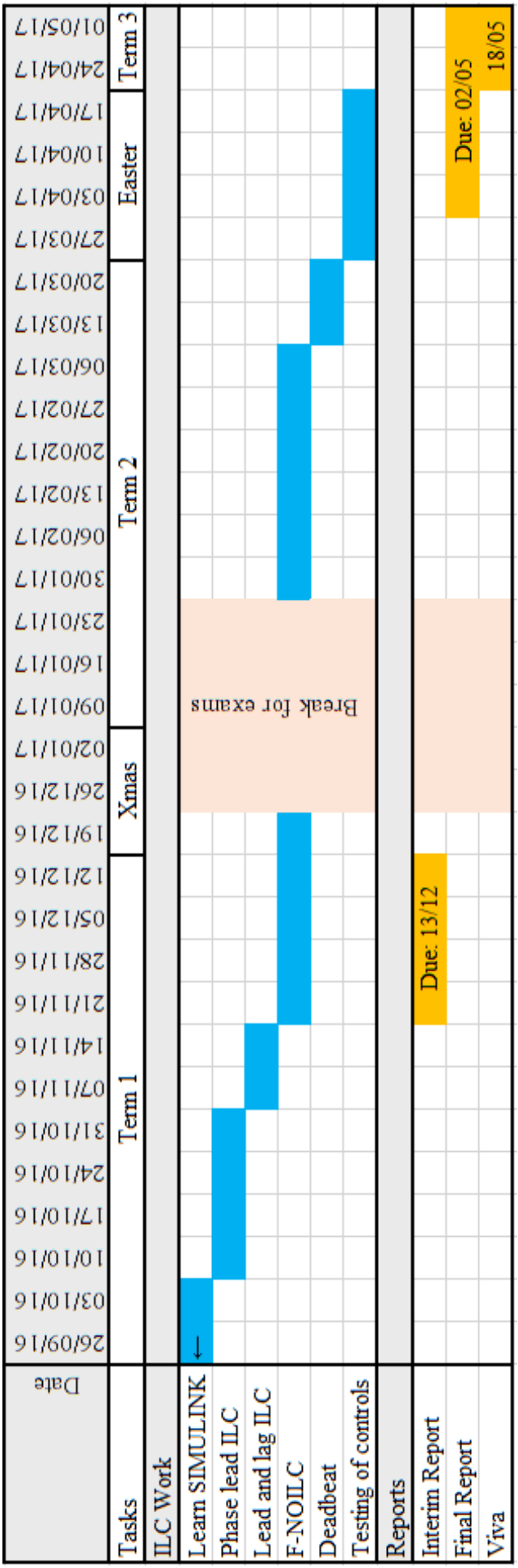
```
    title('Effect of Q and R on Last iteration MSE value capped at 1');
    xlabel('Q');
    ylabel('R');
    zlabel('Last iteration MSE value capped at 1');
    set(gca, 'XScale', 'log', 'YScale', 'log');

    %plot capped mean MSE value over trials for each Q and R
    figure

surface(qvalues,rvalues,mse_error_global_mean_capped,'FaceColor','inter
p','MeshStyle','both','LineStyle',':','Marker','.','MarkerFaceColor','k
')
    view(3)
    title('Effect of Q and R on Mean MSE value capped at 1');
    xlabel('Q');
    ylabel('R');
    zlabel('Mean MSE value capped at 1');
    set(gca, 'XScale', 'log', 'YScale', 'log');

    %plot minimum MSE value over trials for each Q and R
    figure

surface(qvalues,rvalues,mse_error_global_min,'FaceColor','interp','Mesh
Style','both','LineStyle',':','Marker','.','MarkerFaceColor','k')
    view(3)
    title('Effect of Q and R on Minimum MSE value');
    xlabel('Q');
    ylabel('R');
    zlabel('Lowest MSE value');
    set(gca, 'XScale', 'log', 'YScale', 'log');

end
```

Appendix G.

Gantt chart from interim report



Gantt chart of actual timeline



Appendix H.

Appendix contains Project Brief:

Project Brief Part III

Project Title:
Design, Simulation and Analysis of Iterative Learning Control for a Multi-Axis Gantry Robot
Name:
Arjun Patel (ap5g14)
Supervisor:
Professor Eric Rogers
Robotic systems which need to carry out a repeatable action can often have problems always performing the task perfectly when a control algorithm is not implemented. Iterative learning control utilises a reference and the previous operation to perform the task better.
During this project, I will investigate different iterative learning controls and simulate them on a model of the gantry robot in Simulink. Simulink is a program integrated into MATLAB and will allow me to try different controllers without loading them onto the physical robot. The model will provide a realistic plant on which the control algorithms will be tested and the outputs can be graphically visualised so the best control can be seen. If the iterative learning control algorithm meets the bounds of the gantry robot, then it can be tested on the hardware. The physical robot and the model will have some discrepancies and inconsistencies due to the model being slightly dated and not perfectly simulating the hardware.
The goals in this project are as follows:-

1. Research different iterative learning control algorithms
2. Design algorithms on Simulink
3. Analyse and adjust controller on the model of the gantry model
4. Find suitable iterative learning control for the hardware gantry robot
5. If controller meets requirements, test it on the gantry robot

Appendix I.

Folder with submission includes the Matlab code for all of the controls.