Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

A framework for evaluation of iterative learning control

Examensarbete utfört i reglerteknik
vid Tekniska högskolan vid Linköpings universitet
av

Johan Andersson

LiTH-ISY-EX--14/4751--SE

Linköping 2014

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

A framework for evaluation of iterative learning control

Examensarbete utfört i reglerteknik
vid Tekniska högskolan vid Linköpings universitet
av

Johan Andersson

LiTH-ISY-EX--14/4751--SE

Handledare: **Lic. Patrik Axelsson**
ISY, Linköpings universitet
Dr. Henrik Tidefelt
Wolfram Mathcore

Examinator: **Dr. Mikael Norrlöf**
ISY, Linköpings universitet

Linköping, 31 mars 2014

Avdelning, Institution
Division, Department

Avdelningen för reglerteknik
Department of Electrical Engineering
SE-581 83 Linköping

Datum
Date

2014-03-31

Språk
Language

☐ Svenska/Swedish

☒ Engelska/English

☐ ________________

Rapporttyp
Report category

☐ Licentiatavhandling

☒ Examensarbete

☐ C-uppsats

☐ D-uppsats

☐ Övrig rapport

☐ ________________

URL för elektronisk version

http://www.ep.liu.se

ISBN

—

ISRN

LiTH-ISY-EX--14/4751--SE

Serietitel och serienummer
Title of series, numbering

ISSN

—

Titel
Title

Ett ramverk för utvärdering av iterative learning control

A framework for evaluation of iterative learning control

Författare
Author

Johan Andersson

Sammanfattning
Abstract

I många industriella tillämpningar används robotar för tunga och repetetiva uppgifter. För dessa tillämpningar är iterative learning control (ILC) ett sätt att fånga upp och utnyttja repeterbarheten för att förbättra någon form av referenseföljning.

I det här examensarbetet har det tagits fram ett ramverk som ska hjälpa en användare att kunna untyttja ILC. Det visas handgripliga exempel på hur man enkelt kan avända ramverket. Övergången från den betydligt mer vanliga diskreta ILC algoritmen till det kontinuerliga tillvägagångssättet som anänds av ramverket underlättas av teroretiskt underbygga inställningsregler. Den uppnåeliga prestandan demonstreras med hjälp av ramverkets inbyggda plotfunktioner.

Nyckelord
Keywords

framework, iterative learning control, mathematica, systemmodeler

Sammanfattning

I många industriella tillämpningar används robotar för tunga och repetetiva uppgifter. För dessa tillämpningar är iterative learning control (ILC) ett sätt att fånga upp och utnyttja repeterbarheten för att förbättra någon form av referensefölj-ning.

I det här examensarbetet har det tagits fram ett ramverk som ska hjälpa en användare att kunna untyttja ILC. Det visas handgripliga exempel på hur man enkelt kan avända ramverket. Övergången från den betydligt mer vanliga diskreta ILC algoritmen till det kontinuerliga tillvägagångssättet som anänds av ramverket underlättas av teroretiskt underbygga inställningsregler. Den uppnåeliga prestandan demonstreras med hjälp av ramverkets inbyggda plotfunktioner.

Abstract

In many industrial applications robots are used for heavy and repetitive tasks. For these applications iterative learning control (ILC) is a way to capture the repetitive nature and use it to improve some kind of reference tracking.

In this master thesis a framework has been developed to help a user getting started with ILC. Some hands-on examples are given on how to easily use the framework. The transition from the far more common discrete time domain to the continuous time domain used by the framework is eased by tuning theory. The achievable performance is demonstrated with the help of the built-in plot functions of the framework.

Acknowledgments

First and foremost I would like to thank everyone directly involved in the work. Dr. Mikael Norrlöf and Lic. Patrik Axelsson for much appreciated comments and points of view on the thesis. I would like to give an extra big thanks to Dr. Henrik Tidefelt for always being available for discussing any question from any part of the whole spectrum of engineering.

I also would like to thank MathCore for the opportunity to write my master thesis and everyone who works there for the very nice treatment. Wish you the best of luck in the future.

Last but not least I would like to thank family and friends for making the world a funnier, better and more interesting place to live in.

Linköping, Mars 2014
Johan Andersson

Contents

1 Introduction **1**
1.1 Background 1
1.2 Motivation 1
1.3 Outline 2

2 *Mathematica* and *SystemModeler* **3**
2.1 *Mathematica* 3
2.2 *SystemModeler* 3
2.3 *Wolfram SystemModeler Link* 4

3 Industrial robots **5**
3.1 Introduction 5
3.2 Degrees of freedom 5
3.3 Kinematic model 6
3.3.1 Forward kinematics 6
3.3.2 Inverse kinematics 6
3.4 Dynamic model 7
3.5 Path planning 7
3.5.1 Cubic polynomial trajectory 8

4 Iterative learning control **9**
4.1 Concept 9
4.2 ILC algorithm 9
4.3 Arimoto postulates 10
4.4 Advantages 11
4.5 Limitations 11
4.6 Serial or parallel ILC structure 11
4.7 Stability 12
4.7.1 Convergence criteria 12
4.7.2 Conventional controller 13
4.8 Different ILC algorithms 13
4.8.1 Arimoto 13
4.8.2 Plant inversion 13

5 Framework design **15**
5.1 Design goal 15
5.2 Interface 15
5.2.1 Options 16
5.3 Dimensions 16
5.4 Work flow 16
5.4.1 Interfaces between different ILC functions 17

6 Framework implementation **19**
6.1 *Mathematica* function 19
6.2 ILCRun 20
6.3 Plot functions 20
6.4 Creating models 21
6.5 Linearize 22
6.6 Frequency analysis 22

7 Results **25**
7.1 Basic example 25
7.1.1 Choosing parameters with **ILCTimeAhead** 28
7.2 An approximation of an industrial robot example 29
7.3 Non-iteration varying disturbance 34

8 Concluding remarks and future work **39**
8.1 Summary 39
8.2 Future work 39

A Appendix **43**
A.1 Periodogram 43
A.2 Welch's method 43

B Framework functions **45**
B.1 Linearize 45
B.2 ILCRun 46
B.3 Make 47
B.4 Robot specific functions 48
B.5 Create 48
B.6 Plot 49
B.7 Frequency plots 49

Bibliography **51**

1 Introduction

This work is made to present a framework for iterative learning control (ILC). To put it in a realistic context it will be tested on models of industrial robots.

1.1 Background

Industrial robots are a common tool in the industry and are often assigned with repetitive tasks that should be carried out with high precision. Since robots is a competitive business there is always the search for advantages between its competitors. This has increased the demand on cheaper and more flexible robots which in turn increases the demands on modeling and controlling. One way to achieve high-performance control for repetitive processes is iterative learning control. Iterative learning control is a control method which uses the information gained by past iterations to alter the input signal to gain a smaller error between the reference and output [Bristow et al., 2006].

This concept goes back industrially to 1967 when the first patent was applied even though the research field did not mature until 1984. That was namely the year when three different research teams Arimoto et. al, Casalino and Bartolino, and Craig presented their independent research [Norrlöf, 2000].

1.2 Motivation

Constructing new robots is a complex and very expensive process. Various configurations must be measured against each other. Nowadays extensive modeling and simulation is thus required. Therefore integrating a framework for evaluation of ILC schemes in the constructing process can be very useful since it will

help to find out which design is compatible with an ILC scheme.

Another area of interest would be an existing robot that already has a conventional feedback controller implemented and wishes to improve the performance by also applying an ILC scheme to the system.

1.3 Outline

First, Chapter 2 will give an introduction to *Mathematica* and *SystemModeler* which are the softwares used. In Chapter 3 industrial robots will be addressed, including trajectory planning. Chapter 4 presents the theoretical background for the ILC algorithms. In Chapter 5 the interface and work flow of the framework will be motivated whereas in Chapter 6 the actual implementation of the functions will be presented. Examples and results will be given in Chapter 7. Finally, concluding remarks and future work will be discussed in Chapter 8.

2

Mathematica and *SystemModeler*

In this chapter a brief explanation of *Mathematica* and *SystemModeler* will be given. These two programs are the main softwares for the ILC framework, *SystemModeler* for the models and *Mathematica* for the functions defining the ILC framework. The connection between these two is *SystemModeler Link*.

2.1 *Mathematica*

Mathematica is a tool for advanced mathematic calculations and it started to be developed by Stephen Wolfram in 1988. At its core it is a symbolic-numerical solver. This means that if it is possible it tries to find an analytical answer but when deemed impossible it resorts to numerical methods. Since it has been continuously developed over the last 25 years it has a lot of potent functions for calculating as well as presenting data.

Since *Mathematica* is the tool for the implementation of the ILC framework the more predefined functions there exists the easier the implementation becomes. These predefined functions helps in almost all areas from output responses to plotting.

2.2 *SystemModeler*

SystemModeler is a tool for modeling and simulation developed by the MathCore. *SystemModeler* is based on the modelica language which uses a component based approach. A component based approach means that by placing components and then connecting them it will create a number of underlying equations. This means that in the end when all components are connected there will be a large

system of equations which then is solved yielding a simulation data object. The simulation data object contains every signal, state and other information regarding the simulation. An advantage with this components based approach is that the user does not need to explicitly write down equations which can speed up the development.

All signals and states saved in the simulation data object can be accessed by the ILC framework.

2.3 *Wolfram SystemModeler Link*

Wolfram SystemModeler Link (WSMlink) is the link that makes it possible to connect *Mathematica* and *SystemModeler*. It is a package that is called from the *Mathematica* interface that loads functions that can utilize *SystemModeler* models from *Mathematica*. This is what makes it possible to simulate a *SystemModeler* model using *Mathematica*. The data is then saved in *Mathematica* which enables the ILC framework to use all the available tools given by *Mathematica* for computing inputs, data processing and plotting. An illustration of the relationship between *Mathematica* and *SystemModeler* can be seen in Figure 2.1. A more thorough presentation of the work flow will be given in Chapter 5

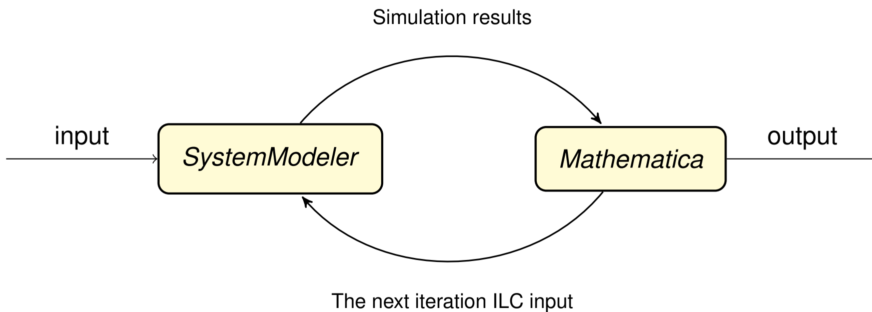


Figure 2.1: *The relation between* Mathematica *and* SystemModeler*.*

3

Industrial robots

The ILC algorithm is not limited to robots but can be used on any repetitive system. However, industrial robots are often used for manufacturing and other repetitive high precision tasks. This makes industrial robots a natural field for ILC research and applications. Therefore some background on industrial robots will be presented.

3.1 Introduction

Robots have been used for a long time now and have become an integral part of the modern industrialized society. As time has moved forward the complexity and usability of them has increased but the fundamentals still remain the same.

3.2 Degrees of freedom

Degrees of freedom is a way of describing which configurations a robot can have. The configurations regard position and orientation which mean that if a robot wishes to have an arbitrary position it needs at least three degrees of freedom. The robot in Figure 3.1 has three degrees of freedom but it still can not reach an arbitrary position. This comes naturally from the fact that since the robot is fixed in the plane it can can only reach every position in the plane. However, an arbitrary position in a plane only requires two degrees of freedom. The robots last degree of freedom then makes it possible to assume a specified orientation at the desired position. While having high degrees of freedom is useful for more advanced maneuvers it will make the robot more susceptible to singularities [Spong et al., 2006, Chapter 4].

3.3 Kinematic model

The kinematic model describes the motion of the robot without regarding forces. Here we make the division with forward and backward kinematics. The forward kinematics is about computing a so called tool-pose as a function of the joint-variables q. A tool-pose is the position and orientation for the tool. Backward kinematics is the opposite i.e. from a given tool-pose determine the joint positions q. Thus the backward kinematics is the problem to solve when dealing with path-planning robots.

3.3.1 Forward kinematics

The function from the joint angles to the final position and orientation is called the forward kinematics. As an example we have an three linked robot manipulator which can be seen in Figure 3.1. The forward kinematics equations for this robot can be expressed as

$$(x_{tool}, y_{tool}, \theta_{tool}) = f(\theta_1, \theta_2, \theta_3) \tag{3.1}$$

or

$$x_{tool} = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3) \tag{3.2a}$$
$$y_{tool} = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\theta_1 + \theta_2 + \theta_3) \tag{3.2b}$$
$$\theta_{tool} = \theta_1 + \theta_2 + \theta_3. \tag{3.2c}$$

3.3.2 Inverse kinematics

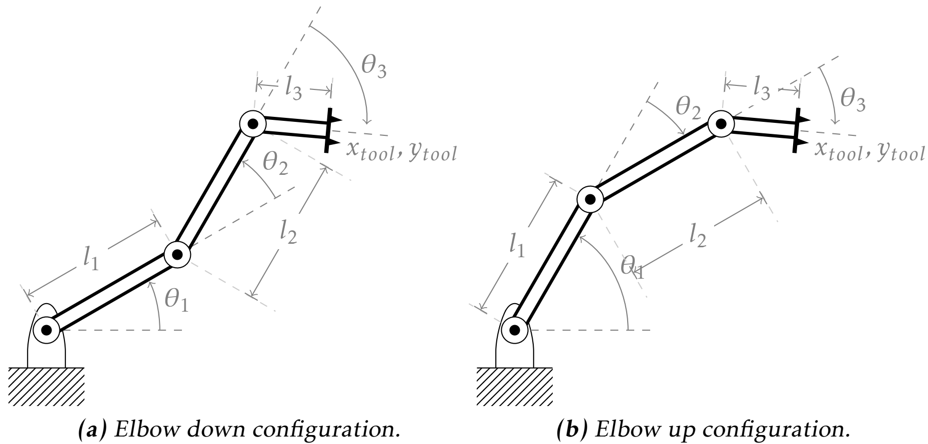
The inverse kinematics planning is very individual for each type of robot. It depends on both its degrees of freedom and physical configuration. It is even more troublesome because since the inverse kinematics is non-linear there is no analytic way to solve every type of robot. Some of these configurations are complex and need advanced algorithms to detect and avoid singularities and correctly choose the right solution.

There are however structures that leads to easier calculations and to illustrate this we take a look at a three-linked, two-dimensional robot. It has three degrees of freedom i.e x-position, y-position and the angle of the last link in that point. For a robot described by (3.2) it is possible to determine a solution by solving the following equation.

$$(\theta_1, \theta_2, \theta_3) = f^{-1}(x_{tool}, y_{tool}, \theta_{tool}) \tag{3.3}$$

which has the following solution [Spong et al., 2006, Chapter 1]

$$\begin{aligned} \theta_2 &= \frac{\arccos(|(x_{tool}, y_{tool}) - l_3(\cos(\theta_{tool}), \sin(\theta_{tool}))|^2 - l_1^2 - l_2^2)}{(2 l_1 l_2)} \\ \theta_1 &= \arctan\left(\frac{y_{tool} - l_3 \sin(\theta_{tool})}{x_{tool} - l_3 \cos(\theta_{tool})}\right) - \arctan\left(\frac{l_2 \sin(\theta_2)}{l_1 + l_2 \cos(\theta_2)}\right) \\ \theta_3 &= \theta_{tool} - \theta_1 - \theta_2. \end{aligned} \tag{3.4}$$



(a) *Elbow down configuration.* ***(b)*** *Elbow up configuration.*

Figure 3.1: *A three linked robot with its two different configurations. The figure is based on an example made by Kjell Magne Fauske [Fauske, 2006].*

As can be seen this is not a linear solution and it is depending on if the joint between link one and two points up or down. The difference between the configurations can be seen in Figure 3.1.

3.4 Dynamic model

The dynamic model of a system describes how it responds to forces and torques. There are two different methods to get the dynamical model; Euler-Lagrange and Newton-Euler. The formulations differ but the result is always a set of identical non linear differential equations. The first method is the Euler-Lagrange which derives the equations by taking the difference between the mechanical systems potential and kinetic energy. The second method is the Newton-Euler which calculates the forces and torques for each link. The final model is obtained by connecting the links i.e. substitute torques and forces for one link with the counterparts for the other links [Spong et al., 2006, Chapter 7].

A quick note here is that when using *SystemModeler* none of these equation needs to be explicitly calculated since it will be solved automatically due to the nature of the component based modeling. That is modeling based on equations as described in Chapter 2.

3.5 Path planning

There exists many ways to create a path for an industrial robot. This can be very complex since collisions must be avoided [Spong et al., 2006, Chapter 5]. However many collision free trajectories can be made by connecting smaller segments. Therefore point-to-point planning will be presented.

3.5.1 Cubic polynomial trajectory

A cubic polynomial trajectory is a smoothed reference that specifies a start and end position and velocities. These constraints mean that a satisfying polynomial trajectory needs to have at least four coefficients, i.e.

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{3.5}$$

Taking the derivative of (3.5) and using the initial and final values at the time t_0 and t_f gives the four equations

$$q(t_0) = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 \tag{3.6a}$$

$$v(t_0) = a_1 + 2a_2 t_0 + 3a_3 t_0^2 \tag{3.6b}$$

$$q(t_f) = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \tag{3.6c}$$

$$v(t_f) = a_1 + 2a_2 t_f + 3a_3 t_f^2 \tag{3.6d}$$

which can be presented as

$$\underbrace{\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} q_0 \\ v_1 \\ q_f \\ v_f \end{bmatrix}}_{b}. \tag{3.7}$$

The coefficient matrix has a non-zero determinant $(t_f - t_0)^4$ for all choices of $t_f > t_0$ and thus the equation is easily solvable as $x = A^{-1}b$ [Spong et al., 2006, Chapter 1].

4 Iterative learning control

In this chapter the background for iterative learning control will be discussed.

4.1 Concept

"Repetition is the mother of all learning" is a classic quote regarding how we as human absorb information and learn new things. For example when seasoning food you taste the food continuously while cooking and planning for the next meal how you should improve it. This is essential the human cooking version of an ILC algorithm. Here the taste deviation from the desired taste form the error. The feedback controller is the chefs perception of how to make the food taste as wanted. Every time the chef makes the same dish again can be seen as a new iteration. It is then very natural to think that the chef has learn the optimal seasoning after a number of iterations. This concept is then transferred to the world of systems and controllers and expressed with mathematics.

4.2 ILC algorithm

Here the ILC update algorithm will be expressed in both words and mathematically. We start with a verbal definition.

4.1 Definition (ILC). For an input-output system that do the same task over many iterations an ILC is an algorithm which uses information from its previous iterations to provide an input for the next iteration that minimizes some norm of the error between the actual output and the desired output.

This definition can be interpreted mathematically in many ways. The following

is a system with a widely used linear and discrete ILC update algorithm [Bristow et al., 2006]

$$e_j[k] = r[k] - y_j[k] \tag{4.1a}$$

$$y_j[k] = G_c[q]u_j[k] \tag{4.1b}$$

$$u_{j+1}[k] = Q[q](u_j[k] + L[q]e_j[k]) \tag{4.1c}$$

where (4.1c) is the actual update algorithm. An explanation for the symbols used can be found in Table 4.1.

q	The shift operator.
$r[k]$	The reference at time k.
$u_j[k]$	The input at iteration j and time k.
$y_j[k]$	The output at iteration j and time k.
$e_j[k]$	The error at iteration j and time k.
$G_c[q]$	The closed loop system.
$L[q]$	The learning filter
$Q[q]$	The stabilizing filter.

Table 4.1: *The symbols used in the ILC algorithm.*

4.3 Arimoto postulates

Now that the ILC update algorithm has been presented the natural question is when it can be used. Arimoto was one of the earliest pioneers in the ILC research field and he formulated six postulates to know where an ILC scheme is applicable [Spong et al., 1992]:

-P1 Every trial (pass, cycle, batch, iteration, repetition) ends in a fixed time of duration T > 0.

-P2 A desired output $r[k]$ is given a priori over $k \in [0, T]$.

-P3 Repetition of the initial setting is satisfied, that is, the initial state $x_j[0]$ of the objective system can be set the same at the beginning of each iteration: $x_j[0] = x_0$, for $j = 1, 2, \cdots$.

-P4 Invariance of the system dynamics is ensured throughout these repeated iterations.

-P5 Every output $y_j[k]$ can be measured and therefore the tracking error signal, $e_j[k] = r[k] - y_j[k]$, can be utilized in the construction of the next input $u_{j+1}[k]$.

-P6 The system dynamics are invertible, that is, for a given desired output $r[k]$ with a piecewise continuous derivative, there exists a unique input $u_\infty[k]$ that drives the system to produce the output $r[k]$.

These postulates are very natural since they make sure each iteration will be comparable and has a solution. There is of course no point in using a learning algorithm if there is great variance between each iteration or if no solution exists.

4.4 Advantages

To validate why the ILC algorithm has some merit let us ask ourselves the questions. Why would anyone implement an ILC algorithm and not just only use a well tuned PID? What can be gained? Let us examine one of the most important impacts gained by the postulates which is that the time interval is fixed and the signal thus can be saved for each iteration. This implies that we now are not limited only to causal filtering which is an advantage for the ILC algorithm compared to a conventional feedback controller.

4.5 Limitations

It is easy to think that any stable system can be saved by iterating enough times. While this is mostly true if the conventional controller is too bad the ILC algorithm will try to compensate with very large inputs. This can lead to difficulties when controlling a system if the input has a limit, which is usually the case for all physical systems.

The use of an ILC scheme comes with some inherent limitations when it comes to such things as noisy environments where it is mandatory to use a designed ILC update law which will limit how close the error can get to zero.

4.6 Serial or parallel ILC structure

There are two versions of ILC structures which are categorized depending on where the update signal shall be applied. The difference is that the parallel ILC structure applies the update directly on the input signal to the system whereas the serial ILC structure applies it to the reference [Bristow et al., 2006]. This is illustrated in the block schedules in Figures 4.1 and 4.2. In these figures the letter c denotes the added input given from the ILC algorithm.

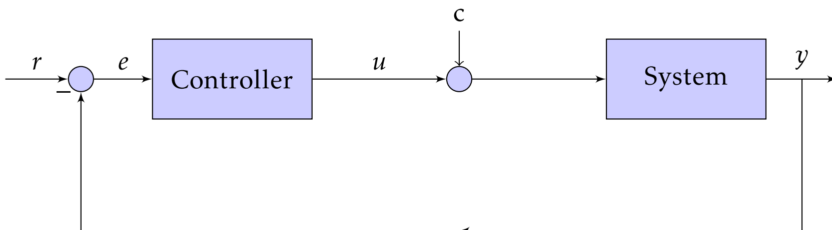


Figure 4.1: *Parallel ILC structure.*

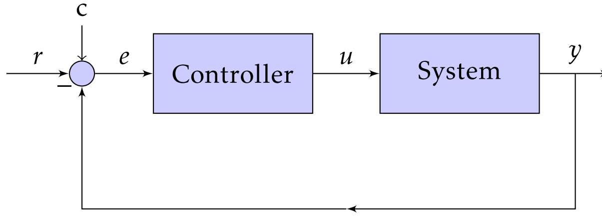


Figure 4.2: *Serial ILC structure.*

4.7 Stability

Stability for the ILC algorithm refers to that the iteration process is stable.

4.7.1 Convergence criteria

We begin with a traditional linear description of an ILC system,

$$e_j = r - y_j \tag{4.2a}$$

$$y_j = G_c u_j \tag{4.2b}$$

$$u_{j+1} = Q(u_j + Le_j). \tag{4.2c}$$

The most straight-forward way of analyzing is to see that the absolute value of the error decreases for every iteration which is easily done when $Q = 1$. The error can be written as

$$\begin{aligned} e_{j+1} &= r - G_c u_{j+1} = r - G_c(u_j + Le_j) \\ &= e_j - G_c L e_j = (1 - G_c L)e_j \end{aligned} \tag{4.3}$$

which is stable if G_c and L satisfy

$$\|1 - G_c L\| < 1. \tag{4.4}$$

If $Q \neq 1$ there is no explicit way of expressing e_{j+1} as a transform of e_j so we use that $u_{j+1} = Q(u_j + Le_j)$ is a contraction if $\|Q(1 - G_c L)\| < 1$ The reason to check if the ILC algorithm is a contraction is that it will then be upper bounded [Kreyszig, 1989, Chapter 5]. To verify this we express the ILC update with two signals $u_{j+1} = T(u_j) = Q(u_j + Le_j)$ and $v_{j+1} = T(v_j) = Q(v_j + Lf_j)$ and takes the difference between them. If $\|T(u_j) - T(v_j)\| < \mu\|u - v\|$ for $\|\mu\| < 1$, $\mu \in \mathbb{R}$ then the ILC update T is a contraction. The difference can be written as

$$\begin{aligned} \|T(u_j) - T(v_j)\| &= \|Q(u_j + Le_j) - Q(v_j + Lf_j)\| \\ &= \|Q(u_j - v_j + L(e_j - f_j))\|, \end{aligned} \tag{4.5}$$

substituting (4.2b) into (4.2a) gives

$$e_j = r - G_c u_j, f_j = r - G_c v_j \tag{4.6}$$

which substituted into (4.5) gives

$$\begin{aligned} \|T[u_j] - T[v_j]\| &= \|Q(u_j - v_j + L(r - G_c u_j - r + G_c v_j))\| \\ &= \|Q(u_j - v_j - LG_c(u_j - v_j))\| \\ &= \|Q(1 - LG_c)(u_j - v_j)\| \end{aligned} \tag{4.7}$$

which since $\|Q(1 - LG_c)\| = \sup_{\|u_j - v_j\| \neq 0} \frac{\|Q(1-LG_c)(u_j - v_j)\|}{\|u_j - v_j\|}$ [Glad and Ljung, 2011, Chapter 1] finally gives

$$\|T[u_j] - T[v_j]\| \leq \|Q(1 - LG_c)\| \|u_j - v_j\|. \tag{4.8}$$

Equation (4.8) shows that $T(u)$ is a contraction if $\|\mu\| = \|Q(1 - LG_c)\| < 1$ which means the ILC algorithm is upper bounded for $Q \neq 1$ and thus stable.

4.7.2 Conventional controller

A pure ILC algorithm is essentially an open-loop system that will converge to an optimal input. However the system will always be susceptible to noise and model disturbances. If the model disturbance is non-varying between each iteration the ILC algorithm will be able to compensate due to the anticipatory nature of the ILC algorithm. However, for non-repeating disturbances and noise, the lack of feedback can lead to serious performance issues. Therefore a conventional controller in conjunction with the ILC algorithm is often desirable for guaranteed stability [Bristow et al., 2006].

4.8 Different ILC algorithms

Specifying Q and L can be done in many ways with different benefits. Here follows a few ways that can be accessed by the framework developed in this work.

4.8.1 Arimoto

One of the more straight forward ILC designs corresponds in many ways to a PID-controller for the iteration domain. The most classical version is

$$u_{j+1}[k] = u_j[k] + Ke_j[k+1] \tag{4.9}$$

but it is easily expandable to include the P and I parts [Moore, 1999, Chapter 4]. One of the most interesting things about this basic update algorithm is the fact that it uses $e_j[k+1]$. This is a good example of how non-casual filtering can be used which is one of the main advantages of ILC algorithms.

4.8.2 Plant inversion

In many ways the ILC update algorithm can be interpreted as some form of feed forward controller. This make it interesting to look at an update algorithm like plant inversion since a common filter when making feed forward controller are based on some kind of inverse of the plant. The plant inversion ILC algorithm is

expressed as followed:

$$u_{j+1}[k] = u_j[k] + G_c^{-1}[q]e_j[k]. \tag{4.10}$$

Assuming that G_c is an exact model of the system the ILC algorithm will converge in one iteration with an error $e_\infty = 0$ [Bristow et al., 2006]. This can be seen by substituting $L = G_c^{-1}$ into equation (4.3) which yields

$$e_{j+1} = (1 - G_c L)e_j = (1 - G_c G_c^{-1})e_j = 0. \tag{4.11}$$

As always when working with system inversions a more realistic approach is however to make a pseudo-inversion. How close the error will tend to zero is hence depending on how good of an approximation G_c^{-1} is.

5

Framework design

In this chapter the structure of the framework will be discussed. The main parts will be the interface for functions and the work flow. Implementation will be addressed in Chapter 6.

5.1 Design goal

The framework is designed to make it easy to start and analyze the ILC algorithm on a system. This is achieved by using intuitive standard values for startup and options that the more advanced user can utilize. For example the framework will accept all custom made filters as long as they have correct dimensions. To make it easier though, the framework contain functionality to quickly generate filters that work. Being able to use custom filters, references, plots and connected systems while having a framework that can create easy startups is the essence of the framework.

5.2 Interface

Lets begin by looking at a function defined by the framework.

ILCErrNormH2[*errVec***, {***startTime***,** *endTime***}]**.

which will output $\|e\|_2$. All functions of the framework begins with an **ILC**. Everything between the square brackets are the arguments and the braces indicates that the arguments should be in a list.

The arguments are always easy to find out by typing "?" followed by the function

name e.g. **?ILCErrNormH2** will return **ILCErrNormH2[errVec, {startTime, endTime}]**.

5.2.1 Options

Many of the functions have optional arguments that will be useful for different cases. To show this we look at

ILCMakeGainTimeAheadFilter[*Gain*, *TimeAhead***]**

which will produce the two filters Q and L. The option for this function is accessible by calling **Option[ILCMakeGainTimeAheadFilter]** which will return the available options in a list

{“Q_CutOffFreq” -> Automatic, “LowPassButterOrder” -> 5}.

In this case the options are used to specify Q as a low-pass Butterworth-filter with a given cutoff frequency. The options always have a default value which are defined in the function. In this case the “Q_CutOffFreq” -> Automatic means that $Q = 1$ regerdless of the “LowPassButterOrder”.

Lets say we desire a Butterworth filter of the fourth order with a cutoff frequency of 1000 Hz, then the function call will become

ILCMakeGainTimeAheadFilter[*Gain*, *TimeAhead*, “Q_CutOffFreq” -> 1000, “LowPassButterOrder” -> 4].

5.3 Dimensions

One of the most important aspects of the framework is that the dimension of the connected systems transfer matrix is $n \times n$, i.e there are as many inputs as outputs. This is because the ILC algorithm used by the framework is essentially a SISO-algorithm. The framework does however work with square MIMO-filters thus interactions between different SISO-system can be addressed.

5.4 Work flow

The work flow for the framework is presented in Figure 5.1. The following list will address each of the boxes in the diagram and give some explanation and reasoning behind this particular breakdown.

- **System and Controller** - The system refers to the plant or model of the plant which must be created in *SystemModeler*. Likewise the controller must also be created with *SystemModeler*. These two will naturally define the closed loop-system. They are also the only two entities that the framework will not create.

- **Connection ILC** - Say one has created a model with a good enough controller, then it is time to redraw the connections to create the ILC scheme. Here it is possible to either draw the connection in *SystemModeler* or use the framework function **ILCCreateModel**. The reason behind create and not connect is that *SystemModeler* does the connecting by creating a new model and not by altering the original one. All functions belonging in this category can be found in Appendix B.5.

- **Designing filter and references** - Before the ILC algorithm can run, the filters and references needs to be specified. All functions belonging in this category begin with **ILCMake** to easily distinguish them and they can be found in Appendix B.3 and B.4.

- **System analysis** - System analysis includes the calculation of the singular values of the system. All functions belonging in this category can be found in Appendix B.1.

- **Run the ILC** - **ILCRun** is the actual ILC function since it is the only function that actually runs the ILC algorithm. The return value is consisting of all the signals and settings. All functions belonging in this category can be found in Appendix B.2.

- **Plot signals and frequencies** - After the algorithm has been run, different plotting functions exists to make a quick overview. All functions belonging in this category can be found in Appendix B.6 and B.7.

5.4.1 Interfaces between different ILC functions

Closely related to the work flow is the interface between different functions. How the different outputs of some functions interacts with the inputs of others. Since some functions are very limited in what they do compared to others they require a different level of flexibility. This leads to that it is hard to make a specific rule about how the inputs and outputs of a general function will be. For example take the following two functions:

- **ILCPlot** - these functions are designed to quickly get some plots of the ILC iterations and in order to do this as simple as possible it takes all the outputs from **ILCRun** and retrieves the ones that are needed.

- **ILCFreqAnalysis** - on the other hand the functionality for frequency analysis in *Mathematica* is very unsupported so these functions are made more general in order to be useful elsewhere as well.

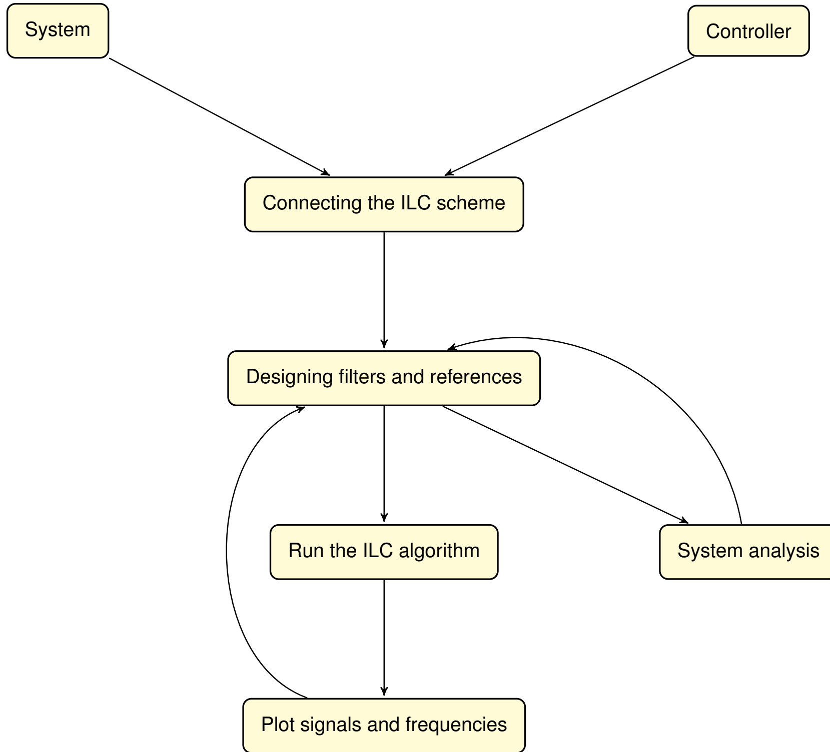


Figure 5.1: *The work flow when using the framework. A description of the steps is given in Section 5.4 and all the functions with their respectively inputs and outputs in the framework can be found in Appendix B.*

6

Framework implementation

In this chapter the implementation of the ILC functions will be presented and motivated. First an example will be given in code to show how *Mathematica* code looks like. The following examples will be given in pseudo code to save space and highlight the important part of the functions.

6.1 *Mathematica* function

Since all functions are written in *Mathematica* it means that they will follow a certain structure. To gain some basic understanding lets begin by showing the implementation of the function **ILCMakeGainAheadFilter** in Algorithm 1.

Algorithm 1: ILCMakeGainTimeAheadFilter

```
1 ILCMakeGainTimeAheadFilter[Gain_, TimeAhead_, OptionsPattern[]] :=
  Module[{Q, L, s, cutOffFreq, butterOrder},
2 butterOrder = OptionValue["LowPassButterOrder"];
3 cutOffFreq = OptionValue["Q_CutOffFreq"];
4 If[cutOffFreq =!= Automatic,
5 Q = ButterworthFilterModel[{butterOrder, cutOffFreq}],
6 Q = TransferFunctionModel[1,s]
7 ];
8 L = TransferFunctionModel[Gain*Exp[TimeAhead*s], s];
9 {Q,L}]
```

The function itself is a solution to bridge between the discrete and continuous domain and is further explained in Section 7.1.1. What follows now are functions

described by pseudo code to give a grasp of what makes them interesting from an implementation point of view.

6.2 ILCRun

The most central functionality of the framework is of course the functions which performs the update-algorithm which in the frameworks case is the **ILCRun** function. The pseudo code is described in Algorithm 2.

Algorithm 2: ILCRun

```
Input: [model, (startTime, endTime), reference, (Q,L)]
Output: All saved simulation data and the calculated results
1 while Convergence criteria not met do
2     Simulate: simData = WSMSimulate[model, startTime, endTime,
          WSMInputFunction -> Join[refInputFunction,iLCCorrInputFunction]] /* The
          option WSMInputFunction is required to use the given
          reference and the update of the ILC-update                */
3     Error: error = e = simData[[referenceName]] - simData[[outputName]]
          /* extracts the desired signals which is used to get
          e = r − y.                                                 */
4     Update: OutputResponse[L,e] and OutputResponse[Q,u_k + Le]
      /* takes the calculated e and uses the given Q- and
          L-filter to update the input signal.                       */
5     Save data - Store signals in vectors for each iteration.
6 return Save data
```

The output of **ILCRun** is given as a list with two element where the first one contains the trajectories from the simulation and the second contains general info such as simulation time and the number of iterations. The output is from now on referred to as {trajs, iLCInfo}. {trajs, iLCInfo} is namely the input for all **ILCPLot** functions which will recur in Section 6.3 and Chapter 7.

6.3 Plot functions

All plot functions follow the same pattern. They take the output from **ILCRun** extract the desired signals and plots accordingly to the chosen plot function. There is however a difference in what will be shown depending if the ILC structure is serial or parallel. If the structure is serial the reference, the output and the reference corrected by the ILC update will be plotted in the same window, whereas if it is parallel the ILC correction signal will be printed in a different window. The choice is made with the "OptionValue" - > mode where mode can be either "Serial" or "Parallel". The pseudo code can be seen in Algorithm 3.

Algorithm 3: ILCPlotAll, ILCPlotCompact, ILCPlotAnimate

Input: [trajs, iLCInfo]
Output: Signal and norm plots
1 **if** *mode == Serial* **then**
2 ILCPlotSerial - /* Plotting the reference, output and the ILC corrected reference in one plot */
3 **else**
4 ILCPlotParallel - /* Plotting the reference and output in one plot and the ILC corrected input in a second plot. */
5 Extract all the signals from trajs
6 Extract number of iterations and the simulation time from iLCInfo
7 Compute the norms
8 Compute the plots and place them in a grid
9 **return** *Plot*

6.4 Creating models

Creating models in *Mathematica* is possible in two ways. Either by WSMConnectComponents or by WSMLink‘Library‘EvaluateModelicaInput. It would be ideal to use WSMConnectComponents in the framework but since it at the moment do not support systems with vectors as inputs it is unfortunately not viable. Therefore **ILCCreateModel** uses WSMLink‘Library‘EvaluateModelicaInput which evaluates a string of text. To generate a model this way the first step was to create a base model in *SystemModeler* and copy the text input into *Mathematica*. Then the function **ILCCreateModel** replaces all instances of the systems and controllers name in the text string to create a new model. The pseudo code is given in Algorithm 4.

Algorithm 4: ILCCreateModel

Input: [System, Controller, Dimension]
Output: Connected system
1 **if** *Mode == Serial* **then**
2 ILCCreateSerial
3 **else**
4 ILCCreateParallel
/* Depending on Mode ILCCreate chooses which of two different strings of text it should use. */
5 Replace all instances of the base models *System, Controller* and *Dimension* with the inputs.
6 Call **WSMLink‘Library‘EvaluateModelicaInput**[string] with the replaced string.
7 **return** *Model*

6.5 Linearize

Linearize is based on WSMLinearze which is a *Mathematica* function for linearizing a *SystemModeler* model. When linearizing around zero it works fine to use it in its original execution. However when linearizing around non-zero inputs and states it will need explicit setting of each input and state. Since this, in many cases, is not feasible for large systems **ILCLinearize** attempts to simplify this process. **ILCLinearize** will for a given input simulate the system for a given amount of time, then it will extract all states and inputs via WSMModelData and use that as an input for WSMSimulate. An important note here is that it is up to the user to make sure that the system becomes stationary in the given time frame. The pseudo code is given in Algorithm 5.

Algorithm 5: ILCLinearize

Input: [model, inputs, outputs, inputvalues, {starttime, endtime}]
Output: Linearized model
1 Simulating with the input values until the system becomes stationary
2 Extract the simData object
3 WSMLinearize with the extracted inputs and states
4 **return** Linearized model

6.6 Frequency analysis

There are three implemented functions for frequency analysis in the framework. The most straightforward is the **ILCFreqAnalysisFourier** shown in Algorithm 6. **ILCFreqAnalysisFourier** takes the discrete time fourier transform of a signal and plots the periodogram as in Appendix A.1.

Algorithm 6: ILCFreqAnalysisFourier

Input: [signal, tSample , timeRange]
Output: A periodogram plot
1 Sampling the continuous function signal
2 Compute the discrete fourier transform with the *Mathematica* function Fourier
3 Plot the squared absolute value of the transform
4 **return** *Plot*

The **ILCFreqAnalysisWelch** is based on Welch's method of averaging periodograms by dividing a signal into smaller segments that may overlap as is shown in Appendix A.2. Then the periodogram is calculated for each segment and averaged. This is shown step by step in Algorithm 7.

Algorithm 7: ILCFreqAnalysisWelch

Input: [signal, tSample, timeRange, window, tWind, numberOfSegments, overlap]
Output: A periodogram plot

1 Divide the signal into smaller segments
2 Sample the segments of continuous function signal
3 Sample the window function
4 Compute the discrete fourier transform with the *Mathematica* function Fourier for each segment
5 Compute the squared absolute value of the transform for each segment
6 Average every periodogram into one and plot it
7 **return** Plot

7

Results

Here we shall present some results from examples that focus on highlighting the functionality and performance of the framework.

7.1 Basic example

Lets show how one can use the framework with a basic example. We have already in *SystemModeler* created a model of a motor with an arm and a flexible joint shown in Figure 7.1. A PID controller has been created as well and can be seen in Figure 7.2.

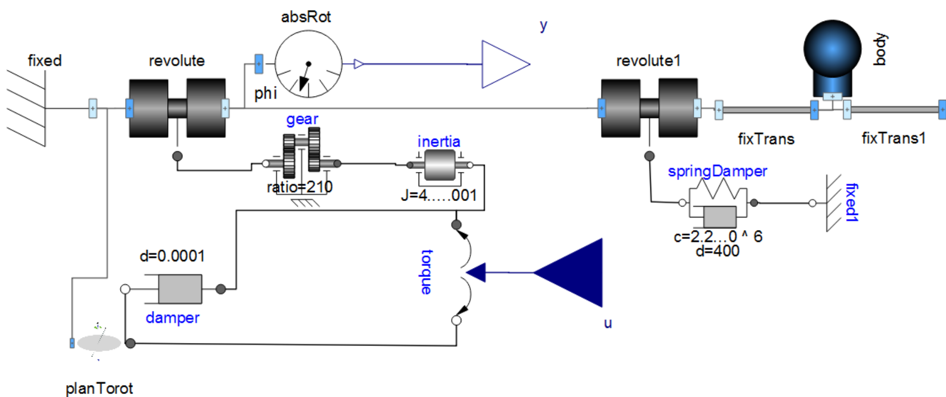


Figure 7.1: *A model of a motor and an arm with a flexible joint.*

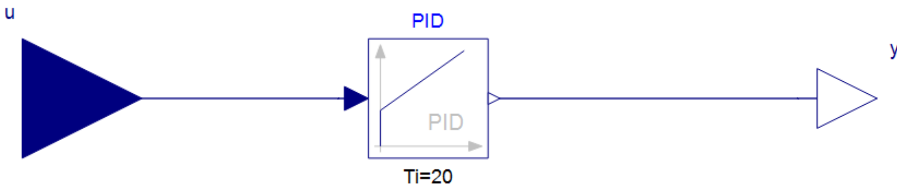


Figure 7.2: One dimensional PID-controller.

The first step is to create the connections since the ILC scheme needs an extra input for the update signal. To do this we call **ILCCreateModel[**“OneAxisSer”**,** “systemOneAxis”**,** “controllerOneAxis”**,** 1**]** which will create the serial model “OneAxisSer” by using the system “systemOneAxis” and the controller “controllerOneAxis” and the fact that the dimension of the system is 1. The result is shown in Figure 7.3.

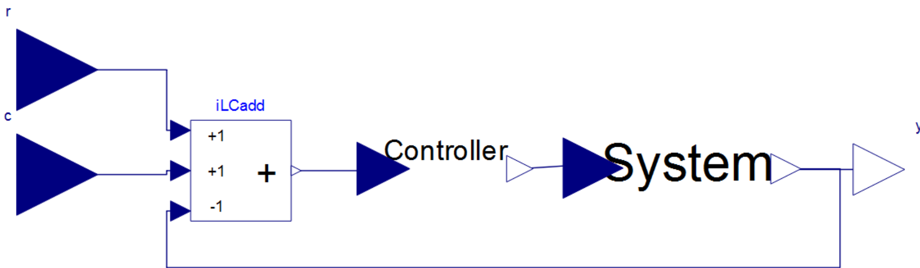


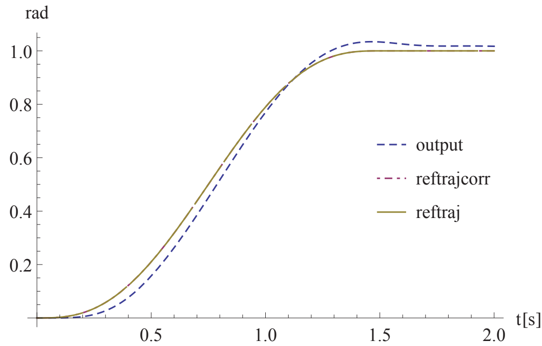
Figure 7.3: The serial connected system.

Now that we have a system we need a trajectory and filter for the ILC update algorithm. The following code exemplifies how to create a filter and reference and also how to run the ILC algorithm.

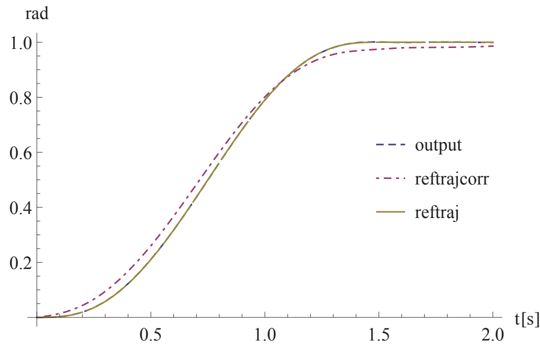
startTime = 0; endTime = 2; iterations = 10;
gain = 1; timeAhead = 0.12;
startPos = 0; endPos = 1; startVel = 0; endVel = 0; startAcc = 0; endAcc = 0; filter = **ILCMakeGainTimeAheadDesign[**gain**,** timeAhead**]**;
quintRef = **ILCMakeQuinticRef[{**startPos,endPos**}, {**startVel**,** endVel**}, {**startAcc,endAcc**}, {**startTime**,** endTime - 0.5**}]**;
{trajs, iLCInfo} = **ILCRun[**“OneAxisSer”**, {**startTime**,** endTime**},** quintRef**,** filter**,** "ConvergenceTest” -> **ILCFixedIter[**iterations**]];**

Now we call **ILCPlotAll[**trajs**,** iLCInfo**]** which makes one plot for each iteration and a list plot for the L_∞-norm and L_2-norm.

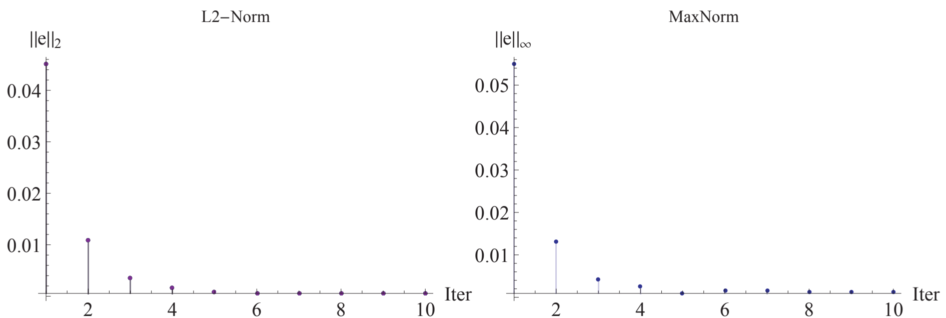
The plot of first iteration is shown in Figure 7.4a and shows the output, reference and the reference corrected by the ILC update algorithm. However, at the very first iteration the ILC algorithm has not yet given any impact and thus it is zero.



***(a)** The first iteration is without any influence of the ILC algorithm.*



***(b)** The same system after 10 iterations of the ILC algorithm.*



***(c)** The L_2-norm of the error for each iteration.*

***(d)** The L_∞-norm of the error for each iteration.*

***Figure 7.4:** The figure shows the tracking for the one dimensional rotating mass.*

However the impact can be clearly seen by the last iteration in Figure 7.4b. The delay and overshoot seen in the first iteration is completely gone and it is very hard to distinguish the output from the reference which indicates "perfect tracking". The L_2- and L_∞-norm are saved for each iteration and shown in separate list plots. The L_2-norm and the L_∞-norm are shown in Figures 7.4c and 7.4d.

7.1.1 Choosing parameters with ILCTimeAhead

The results in this chapter is shown when using the **ILCTimeAhead** function with some parameters. Therefore some motivation for the choice of its parameters will be given. For **ILCTimeAhead** the ILC update is defined as

$$u_{k+1}(t) = u_k(t) + Ke_k(t + \tau) \tag{7.1}$$

and can be viewed in two different ways. The first is to see it as an approximation of (4.9) with a very small τ which would correspond to a next sample. The second is to see it as a rough approximation of the inverse to a system $G(s)$ i.e.

$$Ke^{\tau s} \approx G^{-1}(s). \tag{7.2}$$

This basically approximates the system with a gain and an inverse time delay. This makes sense intuitive since ILC algorithms can be seen as some kind of a feed forward filter. **ILCTimeAhead**:s ability to minimize the norm of the error is closely connected to the accuracy of the plant inversion as can be related to Section 4.8.2. In Figure 7.5 a step response of the system in Figure 7.3 is shown.

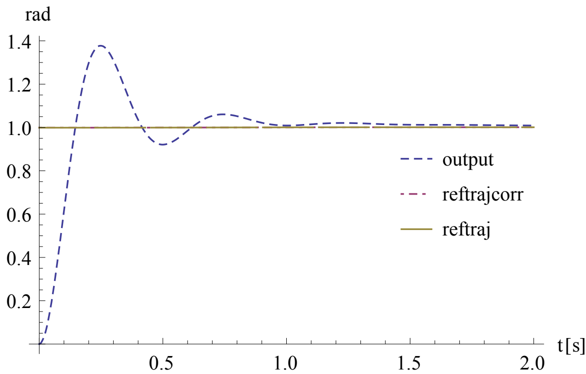


Figure 7.5: *A step response for the motor and arm example.*

From the step response it is possible to see that the rise time is somewhere between 1.1 and 1.4 seconds. This would indicate that the time ahead filter would have $\tau \approx 1.1 - 1.4$. Since the static gain of the system is $|G(0)| = 1$ the inverse of the static gain will be $K = \frac{1}{|G(0)|} = 1/1 = 1$. For $K = 1$, different τ:s was tested by iterating the system 30 times and saving the error norm of the last iteration. The result is shown in Figure 7.6. Here it is possible to distinguish that the performance with respect to minimizing the error is best when $\tau = 0.12$.

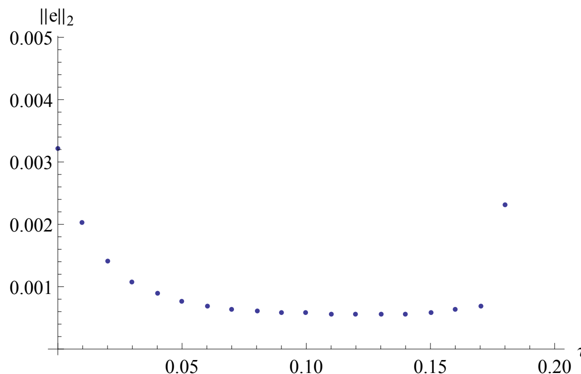


Figure 7.6: *For different τ:s the error norm after 30 iterations.*

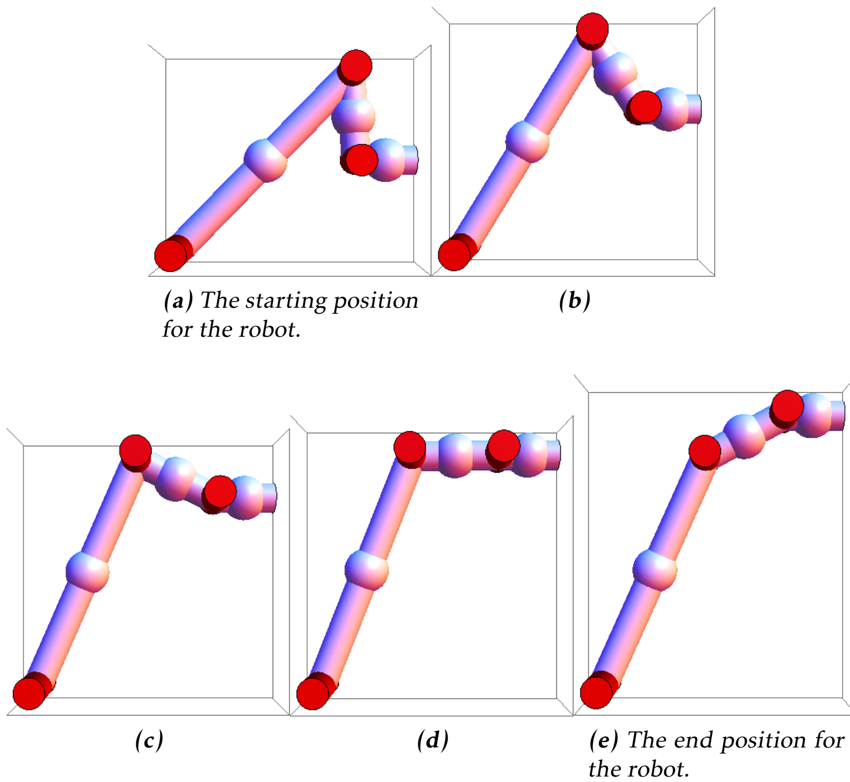
7.2 An approximation of an industrial robot example

In this section we will focus on a two-dimensional approximation of a three-dimensional robot similar to the ABB IRB 7600 [Wernholt and Östring, 2003]. The approximation is made by restricting the robots movement to the vertical plane and has been made due to computational complexity.

The robot has been given a trajectory with **ILCMakeRefFromSurface** to make the tool follow in a straight line along the y-axis while being fixed in x-axis. The tool orientation should be fixed to illustrate that the tool follows a vertical surface. For illustration, the trajectory from start to end position of the robot is shown in Figure 7.7.

The robot is controlled with a PID controller without great success. The result for each axis when following the reference can be seen in Figures 7.8a, 7.9a and 7.10a. In an attempt to minimize the error between the reference and the output a SISO-filter was prepared for each of the axis as **ILCTimeAhead[*1*, *0.15*]** for axis 1, **ILCTimeAhead[*1*, *0.25*]** for axis 2 and **ILCTimeAhead[*1*, *0.15*]** for axis 3. The ILC algorithm was then applied for ten iterations with the result shown in Figures 7.8b, 7.9b and 7.10b. As can be seen the tracking is more accurate after ten iterations of the ILC algorithm. This is also reflected in the norm plots in Figures 7.8c, 7.8d, 7.9c, 7.9d, 7.10c and 7.10d.

Another interesting thing to notice is the simulation time. The time to build the model is roughly 60 seconds and the simulation takes roughly 11 seconds. What makes this separation interesting is that the model does not need to rebuild unless a parameter is changed or another alteration is done to the *SystemModeler* model. This makes it possible to make 5 iterations of ILC faster than it takes to change one parameter in the controller and rebuild it.



(a) The starting position for the robot.

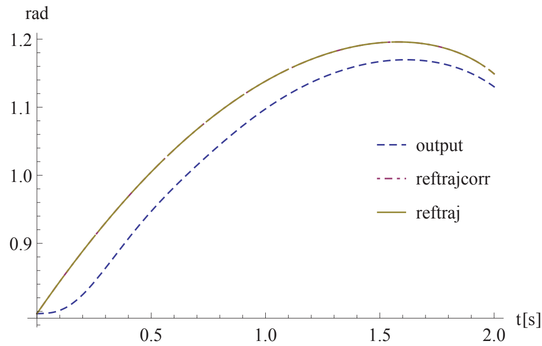
(b)

(c)

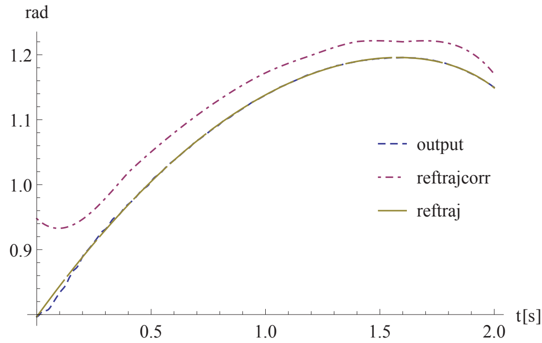
(d)

(e) The end position for the robot.

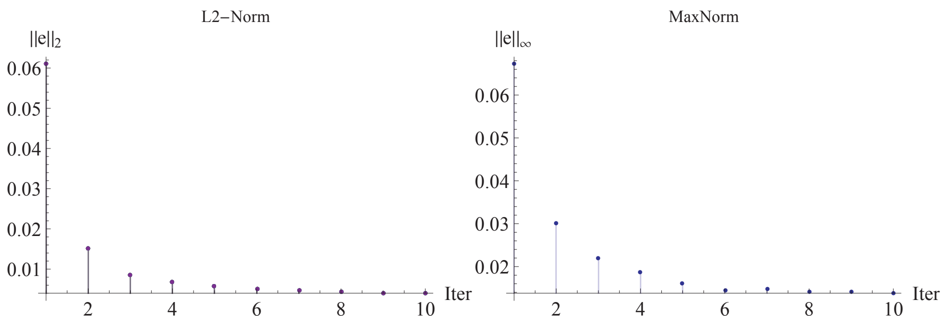
Figure 7.7: *Here is a visualization of the two-dimensional approximation of the ABB IRB 7600. The robot goes with the tool in a straight line from the start to the end position. The pictures are taken as snapshots from* ***ILCAnimation****.*



***(a)** Axis 1 without the ILC update.*



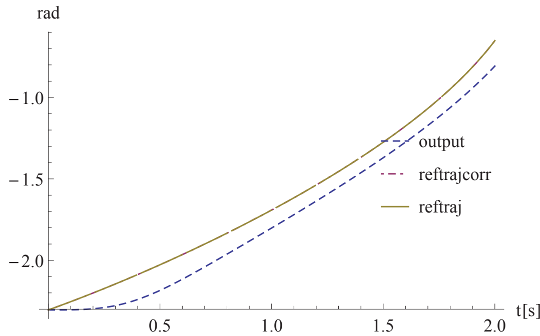
***(b)** Axis 1 after 10 iterations.*



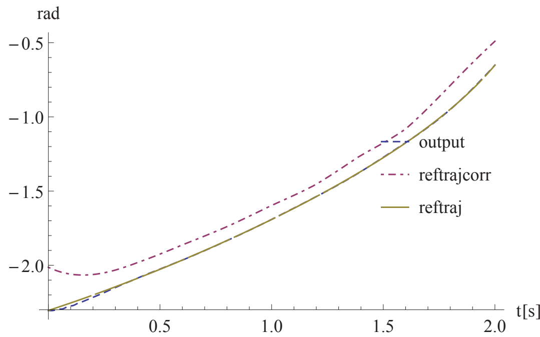
***(c)** The L_2-norm of the error for each iteration.*

***(d)** The L_∞-norm of the error for each iteration.*

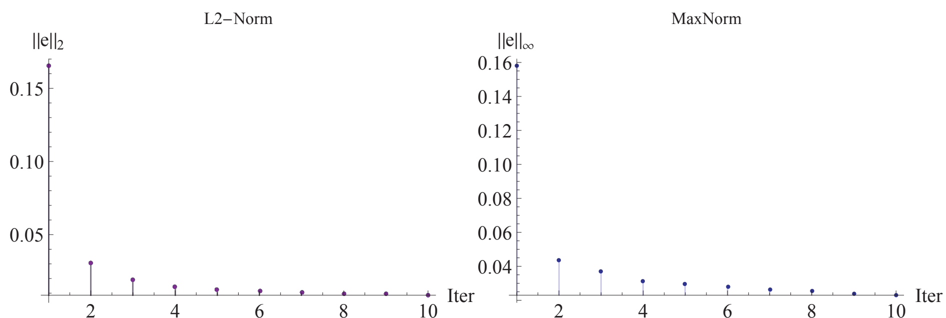
***Figure 7.8:** The figure shows the tracking for axis 1 without any disturbances present.*



(a) *Axis 2 without the ILC update.*



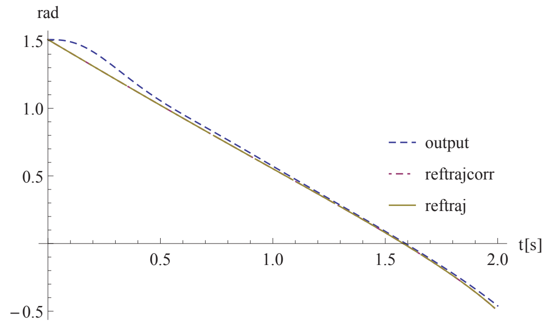
(b) *Axis 2 after 10 iterations.*



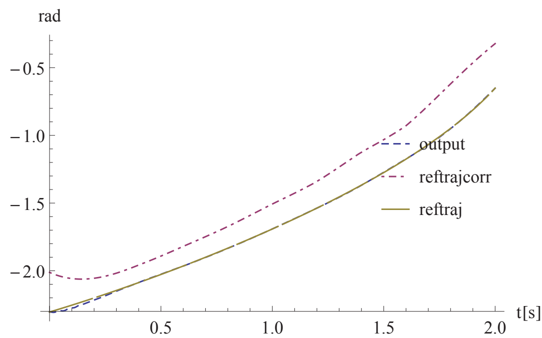
(c) *The L_2-norm of the error for each iteration.*

(d) *The L_∞-norm of the error for each iteration.*

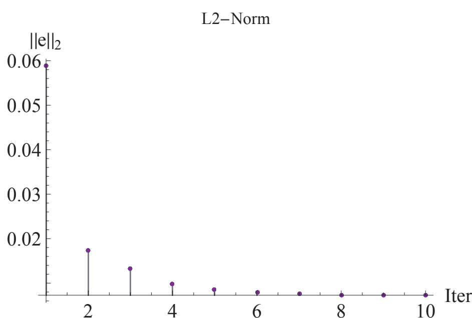
Figure 7.9: *The figure shows the tracking for axis 2 without any disturbances.*



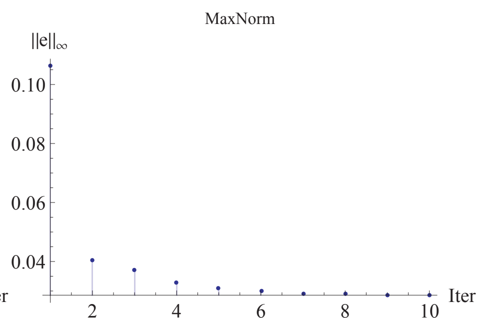
(a) *Axis 3 without the ILC update.*



(b) *Axis 3 after 10 iterations.*



(c) *The L_2-norm of the error for each iteration.*

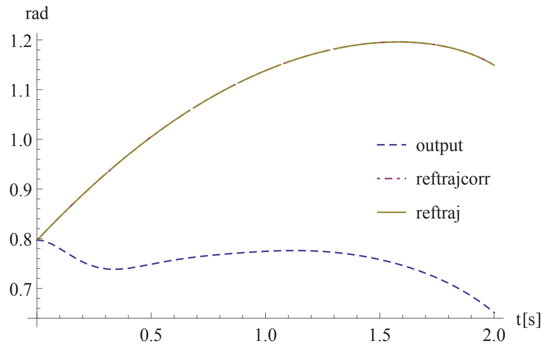


(d) *The L_∞-norm of the error for each iteration.*

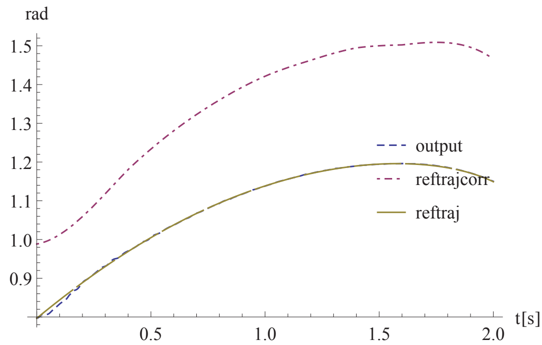
Figure 7.10: *The figure shows the tracking for axis 3 without any disturbances.*

7.3 Non-iteration varying disturbance

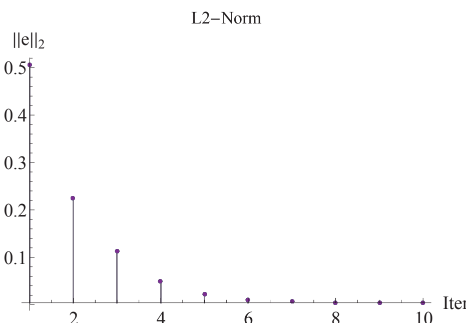
In Section 4.7.2 we claimed that the ILC algorithm is very adept at handling non-iteration varying disturbances. To show this we will once again use the same approximation of the ABB IRB 7600. The difference is now that we also apply gravity as a non-iteration varying disturbance. The result can be seen in Figures 7.11, 7.12 and 7.13. As especially can be seen in Figure 7.11 the difference between the system before and after the ILC algorithm is substantial. The tracking before the ILC algorithm is very poor but since the disturbance has the same dynamics each iteration the ILC algorithm can handle it well. After ten iterations the tracking is roughly the same as after ten iterations without any disturbance. This shows the strength of the ILC algorithm when dealing with non-iteration varying disturbances.



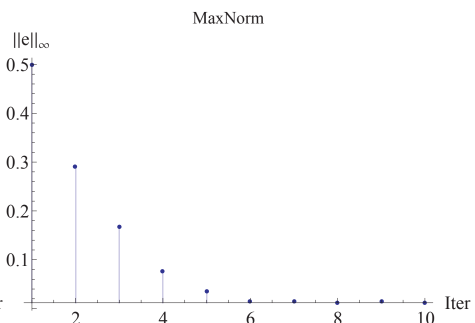
(a) *Axis 1 without the ILC update.*



(b) *Axis 1 after 10 iterations.*

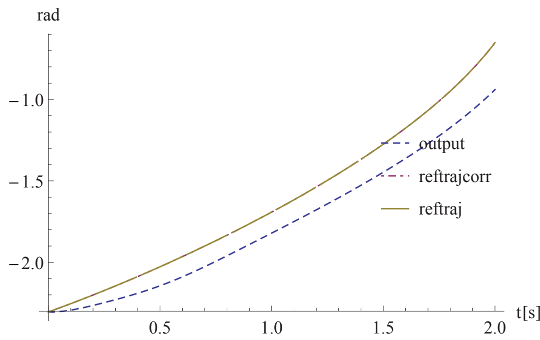


(c) *The L_2-norm of the error for each iteration.*

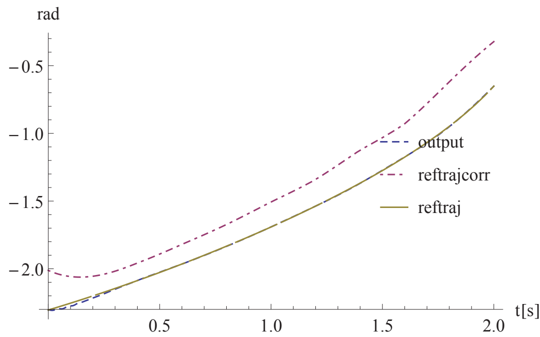


(d) *The L_∞-norm of the error for each iteration.*

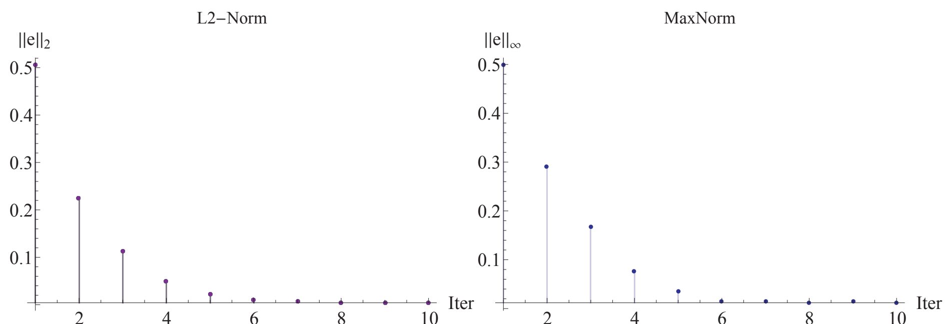
Figure 7.11: *The figure shows the tracking for axis 1 with disturbances present.*



(a) *Axis 2 without the ILC update.*



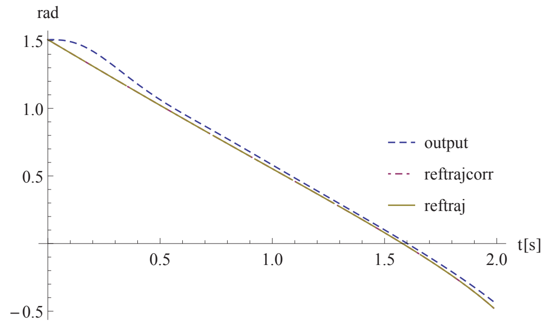
(b) *Axis 2 after 10 iterations.*



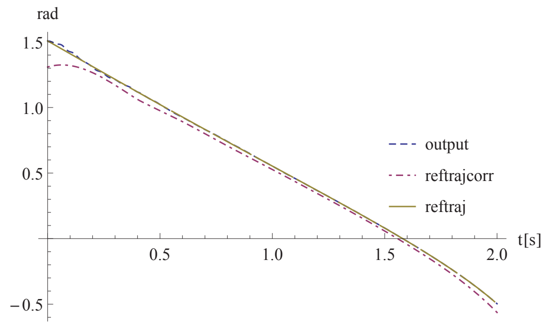
(c) *The L_2-norm of the error for each iteration.*

(d) *The L_∞-norm of the error for each iteration.*

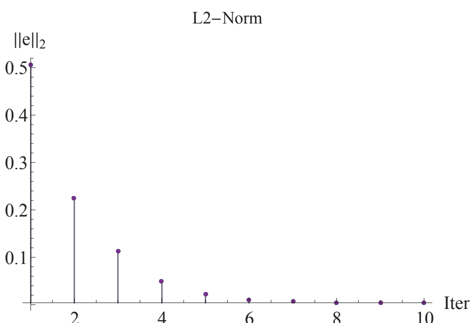
Figure 7.12: *The figure shows the tracking for axis 2 with disturbances present.*



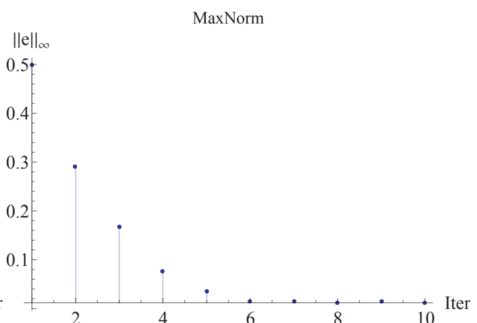
(a) *Axis 3 without the ILC update.*



(b) *Axis 3 after 10 iterations.*



(c) *The L_2-norm of the error for each iteration.*



(d) *The L_∞-norm of the error for each iteration.*

Figure 7.13: *The figure shows the tracking for axis 3 with disturbances present..*

8

Concluding remarks and future work

8.1 Summary

First and foremost it is interesting to see that it is plausible to make a framework using *Mathematica* and *SystemModeler*. This is because it was not entirely certain that it would work. There were mainly two reasons for this. Firstly, because it has not been made before and that the link between *Mathematica* and *SystemModeler* is still very much under development. Secondly, because that the ILC theory is in the discrete time domain versus the continuous time domain of *Mathematica* and *SystemModeler*. However, as can be seen in Chapter 7 it worked out well with respect to the performance.

To truly judge the interface is harder since it requires external testing. Furthermore to complicate things it is also very dependent on the where the framework would be used and the background of the users.

8.2 Future work

Since it is a framework it can of course always get bigger and better with more functionality. However there are some limitations built into the framework since it was a lot of learning by doing during the creation process. To work around these limitations would probably need some major rework. One of those things is the full support for MIMO-systems. Adaptive learning filters that updates between each iteration is also something that would be very interesting to implement. Another interesting thing would be to further investigate the difference between the discrete time and continuous time to see if the continuous time nature of *Mathematica* can contribute with something new, either theoretical or practical

implementation.

Then there are some wishes for the future development of *Mathematica* and *SystemModeler* since much of the frameworks capability is based on how those two can interact. The biggest upgrade would be to incorporate vector valued systems without resorting to the text mode in *SystemModeler*. It would be much appreciated since it would make connecting and creating models much more flexible. Smaller improvements would be a clearer singular value plot function in *Mathematica* and making sure that the output response function in *Mathematica* does not give any small imaginary parts which would disregard the need for chopping the signal manually.

Appendix

A
Appendix

A.1 Periodogram

The most straightforward method suited for the framework is the use of the discrete Fourier transform as defined in *Mathematica* [mat]

$$\hat{f}(m) = \frac{1}{\sqrt{N}} \sum_{k=1}^{N} f(k) e^{2\pi(k-1)(m-1)/N}. \tag{A.1}$$

Since we are interested in how each frequency affects the energy of the signal energy we take $\|\hat{f}(m)\|^2$ to plot the power spectrum [Ljung and Glad, 2009].

A.2 Welch's method

Welch's method is a way of averaging periodograms to increase visibility. By dividing the original signal into smaller overlapping segments, then discretize and transform them individually. This is made by choosing how many segments and how big the overlap should be. Then the resulting length and position for the segments can be calculated as

$$L = nk - (n-1)kp \Longleftrightarrow k = \frac{L}{n-(n-1)p} \tag{A.2}$$

where L is the total length of the signal, n is the number of segments, $p \in [0,1]$ is the overlap and k is the length of a single segment [Gustafsson et al., 2010].

B Framework functions

This appendix contains the inputs and outputs for all the functions included in the ILC framework.

B.1 Linearize

This section contains the functions **ILCCheckConvergence**, **ILCHInfinitySystemNorm** and **ILCLinearize**.

Algorithm 8: ILCCheckConvergence

Input: [Linear model, {Q,L}]
Output: Singular value plots.

Algorithm 9: ILCHInfinitySystemNorm

Input: [Linear model]
Output: H_∞-norm.

Algorithm 10: ILCLinearize

Input: [model, inputs, outputs, inputvalues, {starttime, endtime}]
Output: A linearized model.

B.2 ILCRun

This section contains the functions **ILCRun**, **ILCFixedIter**, **ILCNormConvMaxIter**, **ILCPureOutputResponse**, **ILCPureFunctionAdd**, **ILCErrNormH2**, **ILCErrNormMax** and **ILCSampledNoise**.

Algorithm 11: ILCRun

Input: [model, {starttime, endtime}, refpolynom, {Q, L}]
Output: All the trajectories and information saved about the simulation.

Algorithm 12: ILCFixedIter

Input: [number of iterations]
Output: A function that is an option to **ILCRun** which make it run a fixed number of iterations.

Algorithm 13: ILCNormConvMaxIter

Input: [minIter, normcriteria, max]
Output: A function that is an option to **ILCRun** to make it run until a convergence criteria is met. minIter gives the number of iterations that must run until convergence can be met and maxIter is the maximum allowed iterations.

Algorithm 14: ILCPureOutputResponse

Input: [system, input, {startTime, endTime}]
Output: An output response.

Algorithm 15: ILCPureFunctionAdd

Input: [function1, function2]
Output: A function which is the sum of the function1 and function2.

Algorithm 16: ILCErrNormH2

Input: [errVec, {startTime, endTime}]
Output: The H_2-norm of the error.

Algorithm 17: ILCErrNormMax

Input: [errVec]
Output: The H_∞-norm of the error.

Algorithm 18: ILCSampledNoise

Input: [Distribution, Ts]
Output: A first order interpolating function representing a continuous approximation of sampled noise.

B.3 Make

This section contains the functions **ILCMakeGainTimeAheadFilter**, **ILCMakeInversionFilter**, **ILCMakeQLMatrix**, **ILCMakeCubicRef**, **ILCMakeQuinticRef** and **ILCMakeCyclicRef**.

Algorithm 19: ILCMakeGainTimeAheadFilter

Input: [Gain, TimeAhead]
Output: {Q,L}

Algorithm 20: ILCMakeInversionFilter

Input: [system, lambda-parameter for choosing cutoff frequency]
Output: {Q, L}

Algorithm 21: ILCMakeQLMatrix

Input: [A list of {Q,L} pairs: {{Q,L}...}]
Output: Two $n \times n$ transfer functions that has Q-list and L-list respectively in the diagonal.

Algorithm 22: ILCMakeCubicRef

Input: [{rStart, rEnd}, {vStart, vEnd}, {tStart, tEnd}]
Output: Reference signal as a cubic polynomial.

Algorithm 23: ILCMakeQuinticRef

Input: [{rStart, rEnd}, {vStart, vEnd}, {aStart, aEnd}, {tStart, tEnd}]
Output: Reference signal as a quintic polynomial.

Algorithm 24: ILCMakeCyclicRef

Input: [polyForward, polyBackward, cycles, {startTime, endTime}]
Output: Reference signal which repeats itself for a given number of cycles.

B.4 Robot specific functions

This section contains the functions **ILCForKine**, **ILCInvKine**, **ILCCreateRefFrom-Surface** and **ILCAnimation**.

Algorithm 25: ILCForKine

Input: $[\{L_{link1}, L_{link2}, L_{link2}][\{\theta_1, \theta_2, \theta_3\}]$
Output: $\{x_{Tool}, y_{Tool}, \theta_{Tool}\}$

Algorithm 26: ILCInvKine

Input: $[\{L_{link1}, L_{link2}, L_{link2}][\{x_{Tool}, y_{Tool}, \theta_{Tool}\}]$
Output: $\{\theta_1, \theta_2, \theta_3\}$

Algorithm 27: ILCMakeRefFromSurface

Input: $[\{x_{\text{ref}}(t), y_{\text{ref}}(t), \theta_{\text{ref}}(t)\}, \{L_{\text{link1}}, L_{\text{link2}}, L_{\text{link2}}\}$, {startTime, endTime}]
Output: $\{\theta_1(t), \theta_2(t), \theta_3(t)\}$

B.5 Create

This section contains the function **ILCCreateModel**.

Algorithm 28: ILCCreateModel

Input: [modelName, system, controller, dimension]
Output: Model with the ILC signal added as an input.

B.6 Plot

This section contains the functions **ILCPlotAll**, **ILCPlotCompact** and **ILCPlotAnimate**.

Algorithm 29: ILCPlotAll

Input: [trajectories, simulationInfo]
Output: Plots the references, outputs for every iteration and the H_2- and H_∞-norm of the error.

Algorithm 30: ILCPlotCompact

Input: [trajectories, simulationInfo]
Output: Plots the reference and output for the last iteration and the H_2- and H_∞-norm of the error. Also plots the the error for every iteration in one plot.

Algorithm 31: ILCPlotAnimate

Input: [trajectories, simulationInfo]
Output: Plots the references and outputs in one animating plot to be able to view them over the iterations. Also plots the H_2- and H_∞-norm of the error.

B.7 Frequency plots

This section contains the functions **ILCFreqAnalysisFourier**, **ILCFreqAnalysisWindow** and **ILCFreqAnalysisWelch**.

Algorithm 32: ILCFreqAnalysisFourier

Input: [signal, tSample, timeRange]
Output: A plot of the signals periodogram.

Algorithm 33: ILCFreqAnalysisWindow

Input: [signal, tSample, timeRange, window, tWind]
Output: A plot of the signals periodogram where the signal first has been multiplied with a window function.

Algorithm 34: ILCFreqAnalysisWelch

Input: [signal, tSample, timeRange, window, tWind, numberOfSegments, overlap]
Output: A plot of the signals periodogram where the signal first has been treated with Welch's method.

Bibliography

Wolfram Mathematica 9 documentation center - Fourier. URL http://reference.wolfram.com/mathematica/ref/Fourier.html. Cited on page 43.

Douglas A. Bristow, Marina Tharayil, and Andrew G. Alleyne. A survey of iterative learning control. *IEEE Control systems magazine*, 26(3):96 – 114, 2006. Cited on pages 1, 10, 11, 13, and 14.

Kjell Magne Fauske. Example: Annotated manipulator, 2006. URL http://www.texample.net/tikz/examples/three-link-annotated/. Cited on page 7.

Torkel Glad and Lennart Ljung. *Reglerteori*. Studentlitteratur, Lund, Sweden, second edition, 2011. Cited on page 13.

Fredrik Gustafsson, Lennart Ljung, and Mille Milner. *Signal Processing*. Studentlitteratur, first edition, 2010. Cited on page 43.

Erwin Kreyszig. *Introductory Functional Analysis with Applications*. John Wiley and Sons Inc., 1989. Cited on page 12.

Lennart Ljung and Torkel Glad. *Modellbygge och simulering*. Studentlitteratur, Lund, Sweden, second edition, 2009. Cited on page 43.

Kevin L. Moore. *Iterative Learning Control: An Expository Overview*. Springer, first edition, 1999. Cited on page 13.

Mikael Norrlöf. *Iterative learning control: Analysis Design and Experiments*. PhD thesis, Linköping Studies in Science and Technology. Dissertations No. 653, Linköping University Sweden, 2000. Cited on page 1.

M. W. Spong, F. L. Lewis, and C. T. Abdallah, editors. *Robot Control: Dynamics, Motion Planning and Analysis*. IEEE Control Systems Society, IEEE Press, 1992. Cited on page 10.

Mark W. Spong, Seth Hutchinson, and Mathukumalli Vidyasagar. *Robot Modelin and Control*. John Wiley and Sons Inc., 2006. Cited on pages 5, 6, 7, and 8.

Erik Wernholt and Måns Östring. Modeling and control of a bending backwards industrial robot. Technical Report LiTH-ISY-R-2522, 2003. Linköping University, Sweden. Cited on page 29.

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida `http://www.ep.liu.se/`

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: `http://www.ep.liu.se/`

© Johan Andersson