University of Calgary

PRISM Repository **https://prism.ucalgary.ca**

The Vault Open Theses and Dissertations

2023-05-08

Digital Twins for Distributed Control Systems in IEC 61499

Lesage, Jonathan Lee Vivian

Lesage, J. L. V. (2023). Digital twins for distributed control systems in IEC 61499 (Master's thesis, University of Calgary, Calgary, Canada). Retrieved from https://prism.ucalgary.ca. http://hdl.handle.net/1880/116208

Downloaded from PRISM Repository, University of Calgary

UNIVERSITY OF CALGARY

Digital Twins for Distributed Control Systems in IEC 61499

by

Jonathan Lee Vivian Lesage

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN MECHANICAL ENGINEERING

CALGARY, ALBERTA

MAY, 2023

©Jonathan Lee Vivian Lesage 2023

Abstract

Digital twins are at the centre of smart manufacturing and production in Industry 4.0. Literature surrounding digital twins demonstrates them to be key in optimizing production systems, monitoring system performance, detecting and predicting faults, intervening in production operations if needed, and even designing new manufacturing systems. They achieve these feats by applying a high-fidelity model of a device built from the understood physics in combination with all available and applicable sensor data, and by establishing a bidirectional communication pathway that allows the digital twin to receive data and send signals. Whatever the goal of a specific digital twin, it typically achieves it by using its model to simulate the physical system in some form.

While the features described here are common amongst nearly all digital twins, the concept becomes less clear once the literature is reviewed more thoroughly. Not only do the features of a digital twin vary with different authors, but the method of constructing them varies even further. This issue is noted in research on the topic and acts to bar the application of digital twins in a widespread manner.

This thesis aims to resolve this issue for production systems managed through distributed control systems. An architecture for digital twins that makes use of a high level programming language, in this case MATLAB, in combination with IEC61499 function blocks is proposed. The architecture produces a twin capable of the key functions necessary for it to be useful in a production system, such as model construction and system monitoring and simulation, and produces a

bi-directional communication pathway which allows it to interact with the device and the remaining control system. The digital twin architecture's functionality is demonstrated and evaluated through a series of tests.

Digital twins are a key element to establishing a distributed intelligent sensing and control system, that is a distributed control system which can self manage and operate intelligently and autonomously. This thesis explains how this may be achieved in the IEC61499 standard, and how the digital twin architecture produced here supports this objective.

Preface

This thesis is original, unpublished, independent work by the author, J. Lesage.

Acknowledgements

I would like to thank, first and foremost, my supervisor Dr. Robert Brennan who's knowledge, mentor-ship, and support has made this work possible.

I would also like to thank Alberta Innovates, APEGA, the University of Calgary, and the Department of Mechanical & Manufacturing for their financial support.

Finally, I want to thank my fiancé Tamara for her love and support as I have worked through this degree.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent years have seen an increasing digital presence in manufacturing and production across industries. This has been referred to as the fourth industrial revolution, or Industry 4.0, and aims to make use of concepts like the Internet of Things, Machine Learning, and Artificial Intelligence in industry. It is the melding of the digital and physical spaces that is expected to bring about a new era of productivity and capability.

Digital twins make up the backbone of this effort, and are widely regarded as a central enabling technology for Industry 4.0. At the very highest level, a digital twin is simply a digital representation of a physical asset. However, that definition fails to narrow down how and why such technology is useful or even of interest. In fact, with a definition at such a high level, digital twins are frequently attributed with software that is nothing more than a set of dynamic equations based on the best estimates of the physical properties of an asset.

More careful examination of the concept reveals that digital twins are models constructed by combining our understanding of physics in combination with sensor data. In this manner, digital twins becomes far more capable models than those created in the past. Simply put, digital twins are the most useful possible models we can construct.

While there is much work on the topic, there are a few challenges presenting more widespread adoption of digital twins in industry. The first of these, and perhaps the most problematic, is that there is not a clear definition for a digital twin nor is there agreement as to what features it ought to have. Second, it is not clear from the literature how a digital

twin ought to be built. As such, engineers seeking to apply this concept in industry are challenged to redefine the concept and use it in their own applications.

Additionally, there is minimal literature available on how digital twins can interact with distributed control systems. Distributed control is a control strategy which sees that the control of a production line, or series of devices, is given to multiple industrial controllers each responsible for one or a few of the many devices of the production line. Furthermore, the various controllers can interact with one another through the distributed control system.

The work presented in this thesis attempts to address these shortcomings. This work aims to create a general template which can be used to rapidly construct a digital twin in distributed control application. This includes how to construct the digital twin so it may interface and interact with a distributed control system, as well as how to construct the digital twin in software. Furthermore, it aims to better define the digital twin so that it is clear what a digital twin is supposed to do within the context of industrial production and control.

With that stated, this these aims to make the following contributions:

- A definition for a digital twin
- An itemized list of the key functions a digital twin should be capable of for industrial production
- A general architecture which may be replicated as necessary for constructing a digital twin of a physical asset managed through a distributed control system
- A proposed approach for constructing digital twins through grey box modelling, with an accompanying evaluation of the approach. The novelty here lays in the application of grey box modelling towards digital twins, and existing grey box modelling algorithms are employed
- An evaluation of the International Electrotechincal Commission 61499 standard in the task of constructing a digital twin and interfacing it with a DCS

The remainder of this thesis is organized as follows. First, a review of the relevant literature is presented to establish the motivation and relevance of this work. Next, the enabling technologies necessary for this work are explained. The methodology applied in this work is the reviewed, which explains the experimental apparatus, how the digital twin architecture is achieved, and the methodology for testing the proposed architecture. Finally, the results from the test plan are presented, and the digital twin architecture is evaluated.

Chapter 2

Literature Review

This review is organized into four sections. First, the history and definition of digital twins is examined, so as to establish this key concept. Second, the application of digital twins to smart manufacturing in current literature is explored. Next, the work conducted by researchers in designing and implementing digital twins in various manufacturing systems is examined. Finally, the current work conducted on digital twins and their interactions with distributed control systems is discussed.

After the current state of the literature is established, the research direction is explored. The results of the literature review reveal a need to investigate digital twins and their implementation with distributed control systems further. Conceivably, the application of digital twins in these systems will permit the creation of distributed intelligent sensing and control systems.

2.1 The History and Definition of the Digital Twin

The concept of the digital twin originates from NASA's Apollo Program in the 1960's [1]. The agency would keep a physical twin of a spacecraft on the ground, which they would use to simulate the response and state of the in-flight vehicle. This would be crucial in the case of an emergency or deviation from the flight plan, as they could simulate the response before making critical decisions. The digital twin is the continuation of this.

NASA provided one of the earliest definitions of the digital twin, defining it as "an integrated multiphysics, multiscale, probabilistic, simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin" [2]. NASA clearly views the digital twin as a detailed model of one of their vehicles, and intend to model the life and status of the vehicle using it. This would have a strong application in maintenance and determining flight worthiness.

The concept of the digital twin may be expanded upon further based on NASA's definition. In addition to the various physical models which analyse the current state of the system, sensors constantly augment the digital twin with the most up to date information [3]. This permits an accurate assessment of the current state of the physical twin, while also providing the data required to predict the future state of the physical twin. This allows the digital twin to simulate operating changes. Digital twins can also include visualisation of their physical counterpart, making use of sophisticated graphics to better display the current state of the physical twin to an observer [3]. Additionally, the digital twin is capable of interacting with the physical twin, thus establishing a two way connection between the twins [3].

While NASA's definition of the digital twin focusses on monitoring existing systems, their application is not so limited. Qi and Tao identified three general areas of application for digital twins [4]. These include product monitoring, much like what NASA originally envisioned, as well as product design and smart manufacturing. The authors explain the ability to process massive amounts of data and to communicate with the physical twin expands the digital twins capabilities greatly. Techniques and capabilities from Big Data permit digital twins to make decisions in a variety of fields.

Despite some common threads in the literature, there still exists a great deal of disagreement on exactly what a digital twin is. Kritzinger et al. note in their own literature review of the topic that three different concepts, all called a digital twin, have appeared in research [5]. These are the digital model, which is little more than a computer model of the physical twin, the digital shadow, which takes data from the sensors but does not interact with the physical twin, and the digital twin itself, which both collects data and interacts with the physical twin.

Madni, Madni, and Lucero note a similar difference in the definition of digital twins [6]. They propose four general types based on the capability of the digital twin. These include the pre-digital twin, which is no different than the digital model noted by Kritzinger et al. [5], as well as the digital twin, adaptive digital twin, and intelligent digital twin. Their definition of the digital twin is much like that explained in this review, while the adaptive digital twin incorporates real time updates and an adaptive user interface, and the intelligent digital twin incorporates machine learning and artificial intelligence (AI).

2.2 Application of Digital Twins in Industry and Manufacturing

Industry has noticed the potential power of the digital twin and intends to develop and apply it to their existing processes and products. TESLA aims to create a digital twin for every car they build, while General Electric intends to use the digital twin to monitor the performance and health of their products, and SIEMENS plans to use it to improve efficiency in manufacturing [2]. The applications for a digital twin are immense.

Given the nature of this research, this review will focus on the application of digital twins to smart manufacturing. Researchers have already identified the importance of the digital twin to the next wave of industry, commonly referred to as *Industry 4.0* in Europe. Rosen et al. are one such group of researchers, and argue that future cyber physical production systems (CPPS) must intelligently make decisions to improve the efficiency and capacity of production [7]. They believe digital twin will play an important role in this task and in ultimately optimizing the plant.

In their central paper, Tao and Zhang proposed creating a digital twin of an entire workshop, and provided the framework for the twin [8]. Such a twin would include workers in addition to the various machines in the workshop. They argue that the use of a digital twin for realizing smart production and management will be an inevitable trend.

The importance of the digital twin in manufacturing is evident in the current state of research on its application. He et al. applied the digital twin in the context of controlling an industrial process [9]. Through the application of a digital twin, they demonstrated they could detect different failures in the system, and re-optimize the controller to continue operation if possible. This approach permits the continuation of production in the event of an anomaly.

Liu et al. used a digital twin during the design phases of a flow-type smart manufacturing system [10]. By creating a digital twin of early an prototype manufacturing system, they identified and simulated the characteristics which would lead to optimal production. This permits the next stages of prototyping to more quickly converge towards an optimal design, maximizing production efficiency once deployed.

Meier et al. applied a digital twin to a manual assembly station [11]. At the station, a worker assembles a pneumatic cylinder. The twin creates a laser grid to assist the worker in organizing parts before completing final assembly. The twin itself also adjusts the projected laser grid based on the type of cylinder the worker is assembling. Rather than automating the process, the digital twin here assists production line workers.

Liu et al. investigated the challenges of workshop scheduling for optimal production, and made use of the digital twin workshop. They proposed an algorithm for determining an optimal workshop schedule, which would iterate based on the results of the previously proposed schedule [12]. By using a digital twin, they simulated whether their proposed schedule could work, and if not, determine what was unreasonable and create an new iteration. The use of a digital twin workshop allowed them to rapidly converge to an optimal result before deploying the schedule in the physical workshop.

Zhang et al. applied the digital twin concept to their opti-state control method [13]. Their control method intended to keep a manufacturing system in an optimal state, ultimately minimizing production costs and by extension maximizing profit. By applying a digital twin with their control method, they readily assessed the current state of the system and re-optimized when required.

Vachalek et al. created a digital twin of a small pneumatic cylinder production line using

Siemens Plant Simulation software [14]. With a fully constructed digital twin, they ran multiple simulations to optimize the production time of the product. The cost function for their optimization routine considered the type of cylinder, quantity of product, operation costs, and number of workstations required. As a result of their effort, they reduced production time by 5.2%.

Qamsane et al. proposed a five step process for creating an industrial digital twin, and then applied it in a case study for maintenance purposes [15]. Their digital twin monitored the vibrations in bearings of a four pump manufacturing system. The bearings were a constant source of failure and frequently required maintenance. By using a digital twin, the onset of failure in the bearing was detected immediately, allowing plant managers to conduct the necessary maintenance, rather than following a maintenance schedule or encountering an unexpected failure prematurely. As a result, the downtime of the production facility was minimized.

Zhao et al. used a digital twin to improve the performance of a micro-punching system [16]. Their digital twin collected both sensor and image data from the manufacturing system, and processed it to optimize the control path. This optimization increased the accuracy of the micro-punching system from $\pm 2\mu m$ to $\pm 1\mu m$, and increased the rate of punching from 25 to 65 holes a minute.

In addition to these real applications, multiple researchers are investigating other potential applications of digital twins in industry. Jeon and Schuesslbauer proposed the use of digital twins for flexible manufacturing systems [17]. Their proposal involved six robotic manufacturing stations, or which parts would need to move through a combination of two of them to be completed. Automated guided vehicles would move between stations and deliver components. The authors proposed using a digital twin to optimize the automated guided vehicles, and dynamically manage them in case of a manufacturing station failure. Such an application would ensure optimal production for this system.

Lin et al. proposed an architecture for large scale CPPS with digital twins [18]. In their implementation multiple modules interact with one another to create a real-time simulation of the CPPS. The modules would handle tasks like material deployment and production

planning and execution, as well as possess machine learning. The authors envisioned digital twins as being responsible for visualizing the CPPS and providing the human-machine interface (HMI).

2.3 Creation and Implementation of Digital Twins

The various applications of digital twins in industry demonstrate their immense potential to revolutionize smart manufacturing. However, it still remains unclear how to create or implement one, and no standard yet exists. The International Standards Organization (ISO) is attempting to address this with their upcoming standard 23247 [5], but, due to the unique nature of individual digital twins, there will likely still be much ambiguity in implementation. As such, this is an active topic of research, with many attempting to provide the needed framework for digital twins.

Wu et al. examined this challenge at that conceptual stage [19]. They put forward a five step framework intended to capture the major functional aspects of the digital twin, and guide engineers through the process. They went on to apply this in a case study of a smart electric vehicle.

Leng et al. attempted to tackle the challenge of creating a digital twin of a smart workshop. They proposed a detailed framework for this digital twin, which includes a network of digital twins of various CPPS's within the workshop [20]. Their goal behind creating this is to allow engineers to monitor, control, and eventually optimize the performance of a smart workshop. They went on to deploy a prototype using this framework within Java.

Zhuang et al. recently published a paper proposing a five-step framework for making a digital twin workshop, with the aim of predicting equipment failures [21]. To do this, the incorporated Markov Chains into their digital twin, and gathered statistical data surrounding the performance of workshop equipment. This feature permitted them to predict failure and act early on, thus allowing for preventative maintenance and minimizing down time of machinery.

Quamsane et al. created a practical method for engineers to follow in the design of

digital twins, which they used for their bearing failure detection [15]. Their process involves determining if a digital twin is an effective solution, evaluating the resources and requirements for the twin, designing, deploying, and testing it. Their approach provides a route for engineers to evaluate the applicability of digital twin solutions.

In addition to the challenge of creating digital twins for manufacturing, there exist a multitude of research into the technicalities of implementing them. Uhlemann et al. attempted to address a practical method for implementing a digital twin in small and medium sized enterprises [22]. They proposed the use of machine vision and sensor based tracking as an economical, practical, and effective method for gathering the data necessary to generate the digital twin of a workspace. They aim to make it possible for smaller organizations to apply Industry 4.0 concepts, and thus spread the use of digital twins.

Biesinger et al. examined the challenge of automatically generating a digital twin of a CPPS [23]. They determined one of the key challenges to be data transfer to and from the physical twin, which they proposed solving using a technology called the manufacturing service bus. This provides high data transfer speeds with minimal delay. By applying this technology, they succeeded in rapidly gathering data from an industrial programmable logic controller (PLC), providing the basis to begin automatic generation.

Talkhestani et al. proposed a methodology to automatically synchronize a CPPS with a digital twin [24]. They refer to this as the anchor point method, which permits synchronization based on PLC code analysis. With this synchronized digital twin, they rapidly tested new configurations of the CPPS without having to undertake the traditional methods of reconfiguration. In their case study, they estimated this approach could reduce the time required for the reconfiguration process by 58%.

Borangiu et al. proposed an architecture for a smart manufacturing system using three layers of digital twins [25]. At the lowest layer, the digital twins are responsible for collecting and transmitting data to the physical system. The second layer of digital twins processes the data on performance, maintenance, and detecting anomalies. In the final layer, intelligence is achieved, and these digital twins possess machine learning and AI algorithms to optimize and reconfigure the CPPS as required. This layered approach permits the system to effectively

collect and process data, as well as make decisions on it, without exceeding computing limitations. The researchers went on to implement this in a robotic grabber system.

Xia, Lu, and Zhang attempted to provide a more complete structure for the digital twin workshop initially proposed by Tao [26]. Their implementation included a digital physical workshop, a digital twin engine, and the virtual workshop. The digital physical workshop is the actual workshop where data is collected and devices and personnel are interacted with, while the digital twin engine is responsible for processing and analysing data as well as interacting with the workshop. The virtual workshop uses this data to simulate, optimize, and visualize the physical workshop. Similar to the work of Boarngiu et al., this implementation represents a layered approach.

Preuveneers, Joosen, and Ile-Zudo addressed a critical issue regarding CPPS which contain multiple digital twins, of which many do [27]. They note that as functionality between various digital twins overlaps, there is a potential for data to conflict and result in system failures. To address this, they proposed using feature toggles and software circuit breakers, thus containing a failure should it occur and permit the rest of the system to continue operating. The end result is a more robust CPPS.

Yu-ming, Bing, and San-peng addressed another key issue regarding data from digital twins during implementation [28]. When a single digital twin collects data, it typically does so multiple sources, and as such the data is heterogeneous. Recognizing this, the authors applied AI techniques to sort, process, and interpret the heterogeneous data. With this capability, they created a digital twin of a robotic sorting arm which could simulate and optimize the physical twin. In line with the definitions presented by other authors, their digital twin communicated with the manufacturing execution system to implement optimal performance.

Ait-Alla et al. went to one of the more fundamental questions surrounding sensor requirements for a digital twin [29]. They used a sample CPPS, which consisted of a series of moving carts transferring components between manufacturing stations, for their experiment. By increasing the number of sensors in their system, they would slow communication between the digital and physical twins. However, they more rapidly detected malfunctions

in the system as a result, and could more quickly redirect resources when required. The net effect was an overall increase in productivity in configurations with more sensors.

The concept of visualizing a digital twin has been another topic of investigation, with Zhu et al. demonstrating the use of augmented reality for this [30]. They used Microsoft's HoloLens to visualize the data from the digital twin of a CNC milling machine in real time. The use of this environment permits users to view the operational status of the CNC mill and control it as required, thus creating an HMI. They intend to use this application to further develop intelligent control processes through augmented reality in the future.

2.4 Digital Twins and Distributed Control Systems

The current state of research on digital twins greatly supports its application in smart manufacturing and production processes. Researchers have already applied the process in optimization, failure detection, and assisting workers in their labours. However, the literature also reveals there remains a great deal of technical issues regarding the design and implementation of digital twins in industry. As such, various authors have attempted to address these using a variety of novel techniques. Given the potential for digital twins in smart manufacturing systems, they could be applied to a distributed control system (DCS). A DCS manages the interactions of multiple devices in a broader manufacturing system. Rather than controlling a single device, a DCS will manage the interactions of multiple devices in a production line or process. As such, they are crucial to managing large scale processes and factories which cannot be managed by a single PLC or controller.

A digital twin in a DCS could provide intelligence, robust operation, and complete automation. The digital twin could optimize the production process the DCS is responsible for, permit operators to detect impending failures and notify them to minimize downtime, or even reconfigure the manufacturing process to maximize production in the event of a partial failure. These are critical aspects for minimizing cost and maximizing profitability.

This concept is not lost on researchers, and the applicability of digital twins to DCS's has been identified, in the energy sector directly. Abramkin and Dushin argue that digital

twins would be useful for natural gas production plants [31], where over a billion data points must be collected and processed each year. Such facilities are commonly managed with DCSs. Kovalyov and Nebra identified several other applications for digital twins in energy, including peak power demand management, virtual power plant management, and commercial consumer dispatching [32]. They argue as a digital twin can take economic considerations into account, it could better manage the DCS's used for these systems.

Interestingly though, the application of digital twins to DCSs in literature is limited. Jazdi et al. proposed using an intelligent digital twin to automate a DCS of a modular production system with 32 sensors and 90 actuators [33]. They are yet to obtain results though. Azangoo, Taherkordi, and Blech have proposed using the universal modelling language to create a digital twin to interact with a DCS [34]. Similar to other layered approaches, they have used multiple digital twins to create estimations, conduct consistency checks, and detect faults of a simple conveyor system.

These two works, while applicable to DCSs, do not discuss much regarding how the the digital twin interacts with the rest of the physical system. In the work of Jazdi et al. the topic is not discussed, while in the work of Azangoo, Taherkordi, and Blech, the digital twins are limited in their ability to interact with the rest of the CPPS. For the digital twin to be truly useful, it must be able to interact, reconfigure, and control the physical counterpart as required. These papers lack much explanation on how, or if, this is achieved.

For interactions between digital twins and CPPSs using a DCS to be successful, it is conceivable that the digital twin should interact within the same program that the DCS operates in. This requires that the digital twin interact using an applicable standard. The International Electrotechincal Commision (IEC) 61499 standard is one such candidate. IEC 61499 uses function blocks to define interactions between the various elements of CPPSs , creating the control required for a complex network. These blocks typically represent low level functionality.

Within the literature, IEC 61499 has already been identified as a strong candidate for interfacing with digital twins. Landolfi et al. explain that the event based nature of IEC 61499 simplifies integration with digital twins, making it an ideal choice [35]. They went

on to propose the design of a marketplace of CPPSs with digital twins built within the standard to make digital twin technology more accessible. They argued that given the broad applicability of such technology their proposed marketplace platform would be economically viable.

2.5 Work of this Thesis

The literature makes two concepts abundantly clear. The first is that digital twins are a key element of smart manufacturing systems. Researchers have already established the ability of digital twin's to detect faults [34], provide maintenance predictions [15], dynamically reconfigure control systems [13], optimize production systems [10] [16] [14], and assist workers in factories [11]. All of these are critical elements in maximizing the efficiency of production and manufacturing, as well as automating various processes.

The second is that there remains a great amount of work in determining how to design and implement digital twins for smart manufacturing systems. It is clear that no one standard method exists for creating digital twins, and as such various authors proposed different methods with which this technology should be designed and implemented. Table 2.1 demonstrates this by summarizing literature from this review in which authors either developed a digital twin, or developed an architecture for one. This table presents whether the work of these papers considered three key feature of digital twins, those being a physics based model, data based model, and bi-directional communication pathway, in their implementation. Many do consider all three, but many others do not. There still exists disagreement on what constitutes a digital twin, and as such how to create one.

From the first concept, it becomes clear that a digital twin may have great applicability in a DCS. It is not far-fetched that a digital twin could allow a DCS to automatically detect faults in the production line and minimize troubleshooting, or provide operators with real time maintenance information to minimize downtime and maximize production, and optimize existing processes. Additionally, the ability of digital twins to apply intelligence could allow them to dynamically manage and configure production systems implemented

using DCSs. Together, these could lead to a fully automated factory systems which require minimal human supervision and interaction.

Author	**Physics Based Model**	**Data Driven Model**	**Bi-Directional Communication Pathway**
Rosen et al [7]	?	?	?
Tao and Zhang [8]	✗	✓	✓
He et al. [9]	✓	✓	✗
Liu et al. [10]	✓	✓	✓
Meier et al. [11]	✓	✓	✗
Liu et al. [12]	✓	✓	✓
Zhang et al. [13]	✓	✓	✓
Vachalek et al. [4]	✓	✓	✓
Qamsane et al. [15]	✓	✓	✗
Zhao et al. [16]	✓	✓	✓
Jeon and Schuesslbauer [17]	?	?	✓
Lin et al. [18]	✓	✓	✓
Wu et al. [19]	✗	✓	✓
Leng et al. [20]	✓	✓	✓
Zhuang et al. [21]	✓	✓	✓
Uhlemann et a.l [22]	✗	✓	✗
Biesinger et al. [23]	?	✓	?
Talkestani et al. [24]	?	✓	?
Borangiu et al. [25]	?	✓	✓
Xia, Lu, Zhang [26]	✓	✓	✓
Preuveneers, Joosen, and Ile-Zudo [27]	?	✓	✓
Yu-ming, Binch, and San-peng [28]	✓	✓	✓
Ait-Alla et al. [29]	?	?	✓
Zhu, Liu, and Xu [30]	✓	✓	✓

Table 2.1: A summary of the literature in which authors implemented a digital twin or proposed an architecture for it based on three common features. Note that a "?" denotes if it was uncertain from the paper if the authors considered that feature.

This idea can be succinctly explained as follows. A digital twin has the potential to turn a DCS into a distributed intelligent sensing and control system (DISCS). Digital twins can provide the sensing and intelligence requirements of these systems, and achieve efficiency not accessible via existing approaches.

However, the second concept from the literature identifies a major blockade in creating a DISCS using digital twins. As no single standard method exists for creating or implementing

digital twins, it is not obvious how a digital twin ought to interact with a DCS to create a DISCS. Furthermore, proprietary or custom ways of implementing these digital twins will prevent mass adoption of the technology in industry. There is a clear need for a seamless and effective way for digital twins to communicate and interact with a DCS.

Thus, this thesis aims to address this challenge. Specifically, it aims to provide a framework to create a digital twin for a device controlled in a DCS in the IEC61499 standard. IEC61499 function blocks will make an effective platform for interfacing the digital twin with the rest of the DCS, while also providing key functionality needed for the function of the digital twin. This will provide a bridge between the DCSs and digital twins, and act as the basis for achieving a DISCS in future work.

Chapter 3

Enabling Technologies

The purpose of this chapter is to examine the key technologies that enable the work conducted in this thesis. The two main elements of this are the digital twin concept itself and the IEC61499 standard. First, the concept of distributed control is given some attention beyond the literature review, as it is central to the objective of this work. Second, the concept of a digital twin is honed in for the purpose of distributed control systems. Finally, the IEC61499 standard is explored in a short overview.

3.1 Distributed Control

Distributed control is central to the research conducted here. While a brief overview of it was given in the literature review, a more detailed explanation is given here. This section will examine distributed control in more depth, and examine some of the benefits of its application.

Distributed control is not a control strategy in the sense of classical control, i.e., driving a measurement to match a reference signal. Nor is it focussed on the automation of one or a few devices as a typical PLC would focus on. Rather, distributed control is a strategy which employs individual controllers to manage a few devices or processes, and then links those various controllers through a network so they communicate with one another. It is through this network that data, like sensor readings and control signals, may be shared and

a complex production line may be automated.

This concept is illustrated in Figure 3.1, which outlines a basic DCS consisting of several devices. Individual devices are referred to as nodes, and these nodes are what communicate with a controller. A series of nodes that are collectively managed by a single controller are referred to as a zone, and with this in place it should be noted that there is a single controller per zone. It is through the DCS network that various zone controllers interact with one another and the entire control line can be automated.

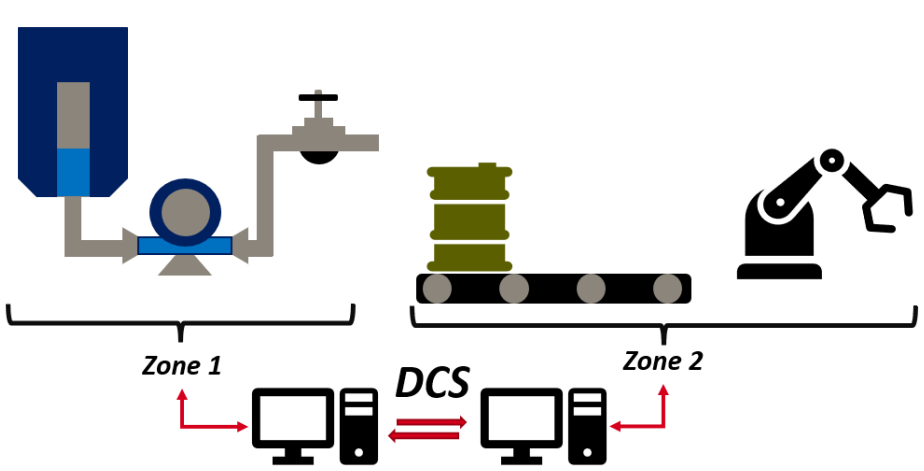

Figure 3.1: A basic DCS network example. In this example, two zones are each given their own controller, and through the DCS they communicate and automate the whole line.

DCSs are commonly found throughout industrial control applications. The energy sector is one such example, where large networks are in place to manage crude oil extraction, refinement, and distribution. Automotive production is another industry that sees DCS application, where complex manufacturing operations take place in discrete steps.

There are two primary advantages to employing such a control strategy. The first of these is scalability. Unlike a centralized control strategy which employs a single controller, a DCS can have nodes and zones added onto it without necessarily having to alter the existing control system. This leads to the ability to rapidly and easily increase the size of a production network. Were a central controller in place, increasing the size of the production operation would entail completely recreating the already in place controller, as well as connecting it to all the various sensors and actuators.

The second major advantage is robustness. In a central control setup, if the controller fails it leads to the entire production facility failing. In the case of a DCS, if a single controller fails it only leads to its respective zone shutting down. This may not lead to the production line failing in its entirety, and as such there is some degree of robustness.

Earlier in the literature review, the concept of a DISCS was introduced. It is appropriate to define this concept here, as it will assist in the following discussion on digital twins. This thesis defines this concept below.

Definition 1 *A distributed intelligent sensing and control system is a distributed control system which can adapt, self-manage, optimize, and operate in the face of uncertainty and disturbances through the application of artificial intelligence and machine learning*

This concept is introduced here as it will have some bearing later on.

This establishes the basics of distributed control. Depending on the DCS platform in place, details like the communication protocols, device interfaces, and programming languages will vary. Thus, these details are left outside of the overview presented here.

3.2 Digital Twins

Digital twins were examined in great detail in the literature review, however, they were not examined in great detail within the context of distributed control systems. This section intends to define a digital twin within the context of a device managed through a DCS.

We begin with a definition of a digital twin. The literature review made it clear that the concept is not well defined once the details are examined in depth. The literature disagrees on what features digital twins should have, with some omitting the bi-directional communication pathway, and some only having a physics or data based model but not both. Futhermore, Features like bi-directional communication, simulating, optimizing, and visualizing systems are not inherent to models or digital representations. Rather, these capabilities require software programs which can collect, store, and process data, visualize performance metrics and results, and provide simulation capabilities.

Based on this assessment, this thesis defines a digital twin as follows:

Definition 2 *A digital twin is a software, or collection of software, which is centred around a high fidelity model of a physical system or process built from the best understanding of the physics available in combination with all available and applicable sensor data. Additionally, a digital twin possesses a bidirectional communication pathway with its physical counterpart which enables the two to interact.*

While the digital twin is fundamentally centred on the high fidelity model described above, that model only becomes useful once it is applied in some meaningful capacity. Consequently, accompanying software must be built around the model and aimed towards applying it in a useful capacity. Thus, a digital twin is not only a model nor is it only software, but rather a powerful tool built from the two. With that concept established, it is important to note that the digital twin architecture of this thesis aims to both provide a method of constructing the model as well as applying it.

Considering this discussion, a digital twin should provide the following functionality to devices in a DCS:

- Construct the high fidelity model at the centre of the digital twin. While the high fidelity model is indeed the heart of the digital twin, constructing it becomes another challenge. A digital twin program should ultimately be capable of using the physics of the system in question and any available sensor data to construct a model, and continuously update that model if needed, which can then be used in all other capacities as required.

- Simulate individual subsystems and overall production system. The ability to simulate the system with changed parameters allows system operators or AI to determine optimal operating points, and determine if changes may be effective given certain circumstances. As digital twins obtain sensor data to augment their internal model, they can create more accurate real time simulations compared to traditional system models, which lack this feedback capability.

- Detect changes and anomalies in the production system, and troubleshoot as accurately as possible. A digital twin can achieve this by comparing current and past performance, or by monitoring key fault indicators, like vibrations. This feature is necessary to minimize downtime in the case an anomaly should occur, as well as for fault recovery. By determining the location and type of failure, the digital twin can provide operators or AI with key data required to either continue production at reduced outputs or safely shut down. This may extend to providing predictive maintenance based on measurements.

- Deliver critical information to operators via an effective graphical user interface. This may take the form of an advanced 3D rendering, text on a screen, or graphs relaying key performance metrics. Information regarding simulation, anomalies, and maintenance should be delivered in a suitable manner to human operators.

These functions are key to realizing the digital twin in its full capacity as the literature outlines it. The work in this thesis will aim to produce a digital twin that can achieve all of these functions, and by extension create a digital twin architecture capable of realizing the complete potential of the technology in a DCS.

The definition and functions provided here adequately define a digital twin and what it should do in the context of a single device or process in a DCS. However, in a DCS there are many devices and processes involved. By extension, in creating a digital twin of a production facility there are many digital twins of individual devices and processes which have to come together to simulate the entire process. Such a setup is necessary if a DISCS is to be achieved. Thus, a framework is needed for connecting these twins.

Figure 3.2 provides an overview of this framework. Such an architecture will use a layered approach, with multiple digital twins. These are defined as node, zone, and overall digital twins. Node digital twins monitor and simulate their individual devices, optimizing locally while taking into account inputs from zone and overall twins, and sending key anomaly and failure data to the other twins. Zone digital twins encompass the entirety of a zone, and as such monitor and optimize their individual zones as required, and are constructed with

the use of the node digital twins. Finally, overall digital twins handle the highest level of operations, including interactions between various zones. It does not simulate the lower level operations, but focuses solely on higher level interactions. The overall digital twin is constructed using multiple zone twins, and as such encompasses all devices in a production line. Due to the layered approach, any behaviour determined by the zone and overall twins is implemented at the device level via the node twins.

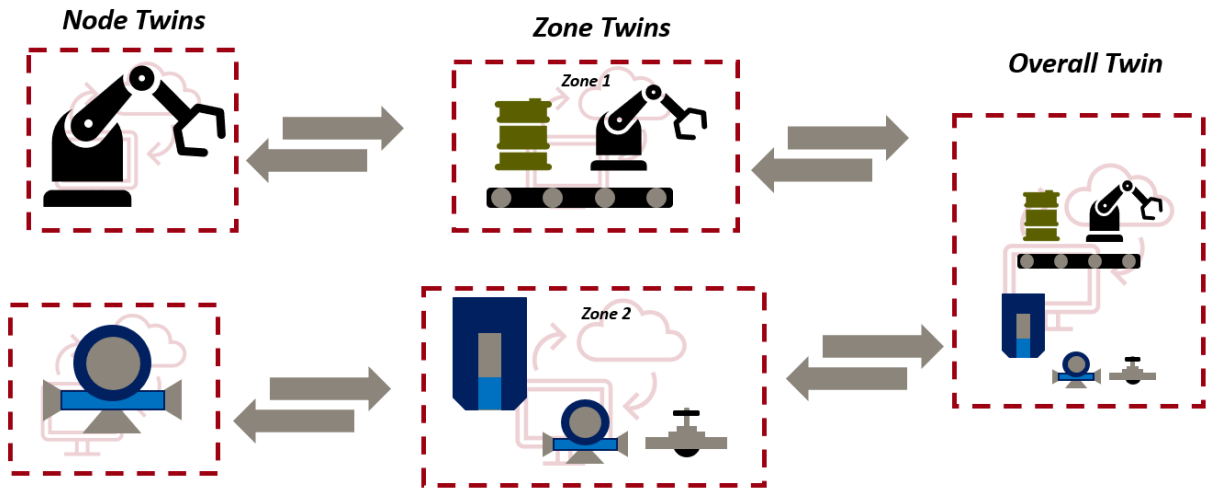

Figure 3.2: A high level overview of the digital twin architecture required to achieve a DISCS

The work of this thesis focusses on the node twin exclusively. This is the lowest block of the conceptual architecture, and necessary to define and test in such a manner that higher level digital twins may be established. As stated previously, the digital twin concept of interest in this thesis will be constructed in IEC61499. Thus, the discussion moves away from the digital twin concept and towards the standard.

3.3 Overview of IEC61499

This section is dedicated to a brief overview of the IEC61499 standard. While the standard can be given a far more dedicated examination, this is left to other well established resources such as that by Vyatkin [36]. For the sake of this overview four main topics will be examined. These include the basics premise of a function block and its event based nature, basic function blocks, service function interface blocks, and composite function blocks. Fi-

nally, a justification for the application of this standard is given along with a brief overview of the software used to implement IEC61499 in this thesis, 4diac.

3.3.1 Fundamentals of Function Blocks

This section is intended to give a brief introduction to the basics of function blocks. While there are three main types of function blocks in IEC61499, they all have common features. Furthermore, IEC61499 relies exclusively on function blocks to achieve its automation goals, and thus it is necessary to examine this. As a precursor to this discussion, a function block is displayed in Figure 3.3.

In Figure 3.3 four main items are outlined. These are *Input Events*, *Output Events*, *Input Data*, and *Output Data*. Each of these items are common throughout all types of function blocks. For the purpose of this discussion, these items are reduced from four to two, namely events and data.

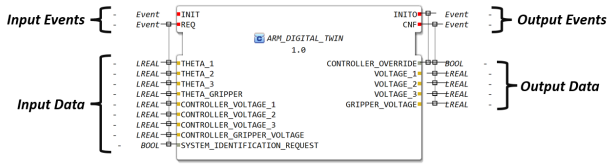

Figure 3.3: An IEC61499 function block, with the major aspects outlined

Events are of particular importance as IEC61499 function blocks are event based in their nature. An event, simply put, is a trigger for action. When an event is received by a function block, the function block will undergo a previously determined set of actions. These actions can be algorithms unique to the function block, and may include multiple algorithms dependent on other conditions. Additionally, these actions may simply be the triggering of additional events. These latter events would be classified as output events, while the previously described are input events. These events will be discussed further while examining basic function blocks.

Data is exactly what is sounds like: information. IEC61499 possesses many data types ranging from integers, to long real numbers, to strings, and to boolean true-false values. The data is used, in some form, by the algorithm previously set up in the function block itself.

Input data may come from sensors, networks, or other function blocks. Output data may be used by other function blocks, or sent to a device or network for use elsewhere.

As a final note, all function blocks have events, but not all have data. There are some function blocks which are employed to send manage events exclusively. As an example, an E_SPLIT_3 block is shown in Figure 3.4. This block takes an input event, and produces two output events. A key rule in IEC61499 is that an events may only originate from a single location and deploy to a single location. As such, event blocks like that in Figure 3.4 are crucial in managing automation programs built in IEC61499.

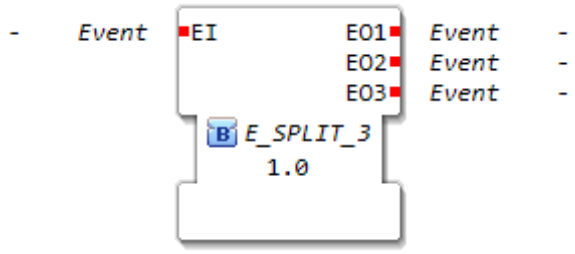

Figure 3.4: An event only based function block

3.3.2 Basic Function Blocks

Basic function blocks (BFBs) are characterized by the use of an *execution control chart* (ECC). An ECC for a 3211 signal function block is given in Figure 3.5 to aid in this discussion. ECCs all possess a *start* or default state, in which no action takes place. This default state is connected via arrows to other states, which are enacted by the arrival of input events. It is the arrival of events that cause a change in the state of the function block. The function block may change state by default, and this can allow the function block to cycle through states without further input of events. Most commonly, this setup returns the function block to the default state as to allow new actions to take place.

The ECC can also be given conditional statements. Rather than simply moving between states upon receiving an input event, an ECC can have a condition which is also required to move. This condition may be linked to a data input, and include a true or false statement, or link to a value statement.

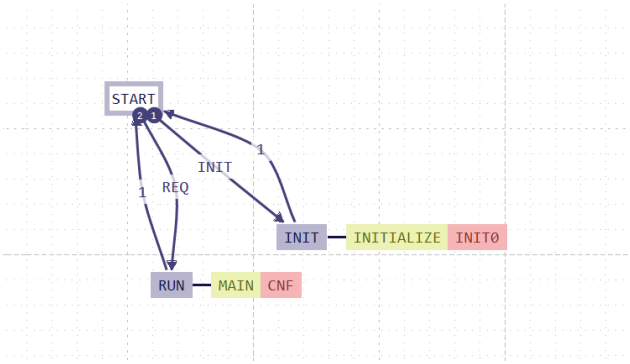

Figure 3.5: An execution control chart, which has two states each enabled by two different events

States typically contain an action. This action includes either the execution of an algorithm, the setting off of an event, or both. These actions ultimately provide the functionality of the function block, regardless of whether or not an algorithm takes place. As a final note on actions, a single state may contain multiple actions, and thereby provide even more flexibility to the function block.

As a final note of discussion on BFBs, these blocks effectively act as their own objects, similar to those in object oriented programming. The blocks can contain their own algorithms, or methods, internal variables with initial values, or properties, and can be replicated in the same program many times, i.e. the same concept of classes and objects. This is a powerful property for IEC61499 basic function blocks, and permits a level of functionality beyond what simplistic programs are capable of. However, it should be noted that function blocks do not permit inheritance as in object oriented programming.

3.3.3 Service Interface Function Blocks

Service interface function blocks (SIFBs) make up the second type of function block. They possess many of the same characteristics as other function blocks, but possess a more rudimentary level of functionality. These blocks are responsible for enabling communication across networks, receiving data from sensors, and sending signals to actuators. In this manner, they are critical for interfacing with systems outside of the IEC61499 program.

This research makes use of these SFIBs in two main capacities. First, they are respon-

sible for communication between the IEC61499 programs and the experimental apparatus. Second, they are responsible for communication in the digital twin between its MATLAB and function block components. A more detailed explanation on this will be given later.

3.3.4 Composite Function Blocks

Composite function blocks (CFBs) are the final major type of function block. These blocks are characterized by an internal function block network, an example of one is shown in Figure 3.6. CFBs are important because they allow for the creation of complex function block networks that combine the functionality of many BFBs and SFIBs.

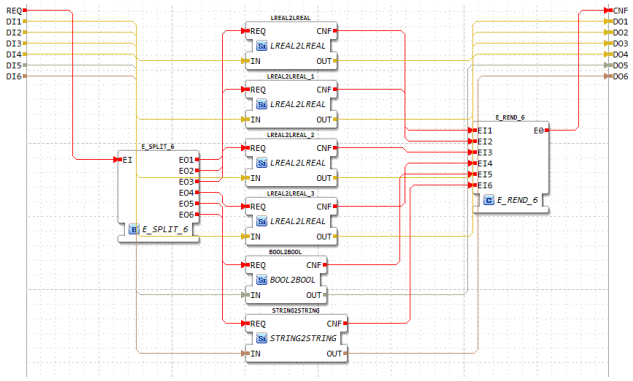

Figure 3.6: A function block network consisting of many basic function blocks

CFBs provide a powerful feature for IEC61499 programs in that they may make use of complicated, and very effective, BFBs and CFBs without having to entirely recreate them at a base level. This allows for rapid reapplication of previously created function blocks, and a far simpler method to make complicated control programs. Furthermore, this feature allows for simple deployment of complicated function block networks into new programs. It is perhaps this feature, most of all, that makes the standard suitable for use with digital twins.

3.3.5 Justification for the Use of IEC61499

It seems appropriate, at this stage, to justify the use of IEC61499 in this work on distributed control systems. Many other platforms for distributed control exist, and it could

be argued that these should be used. However, IEC61499 possesses several features which makes it very effective for interfacing with and building digital twins.

The greatest single reason to use IEC61499 is the event based nature of its function blocks. This is noted in other work, and again reiterated here. By having the event based sequencing setup, a digital twin can interact in discrete manner with a physical device through the control system. Additionally, given the ability for IEC61499 ECCs to have conditional statements on actions, the manner in which a digital twin interacts can easily be controlled without conflicting with the rest of the control system.

The nature of function blocks is another reason to use IEC61499. As previously noted, BFBs are effectively objects belonging to a class. This gives the blocks a high level of functionality that is typical of object oriented programming, and by extension allows for the deployment of complex algorithms where values must be held and used later on. This gives a level of flexibility which is extremely useful for digital twins.

CFBs further reinforce the use of IEC61499 for digital twins. Very complex function networks can be built up for a digital twin to work with. Furthermore, these function block networks can then be easily deployed in their CFB form. It makes the deployment of complex programs, which digital twins necessitate, rather simple on an industrial scale. This is something noted by Vyatkin [36], and becomes an excellent reason to apply IEC61499.

It should be noted that applying the IEC61499 standard allows for a simple mechanism to communicate with the physical device and the rest of the plant. Rather than having an entirely different suite of sensors and software, function blocks allow for the digital twin to leverage the already in place sensors and send signals to actuators on a device. In this manner, IEC61499 provides a ready manner to build the bidirectional communication pathway which helps characterize digital twins.

As a final argument for applying this standard, rather than other automation approaches, is that it is dedicated to distributed control and this is built into its architecture. The standard provides the framework necessary for linking multiple controllers together, as outlined in Figure 3.7, and this may be leveraged in making the higher level digital twins noted earlier.

These elements, taken together, make IEC61499 a good choice for interfacing digital twin

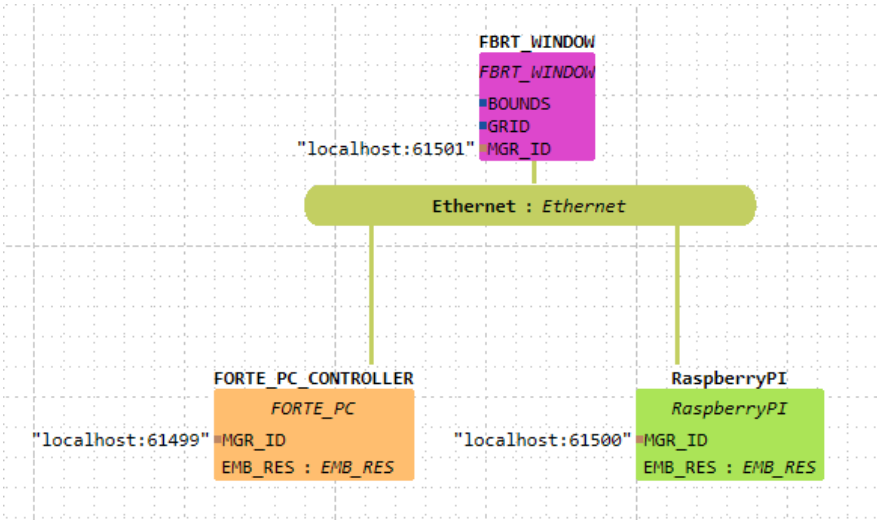

Figure 3.7: An example of a 61499 distributed network consisting of three devices

software with distributed control systems. Furthermore, is makes the use of function blocks suitable for aiding in the construction of a digital twin.

3.3.6 4diac

As a final element of the discussion on IEC61499, the software used in this thesis is introduced. Eclipse's 4diac IDE is employed throughout this thesis and acts as the industrial controller when needed. The software permits the deployment of IEC61499 compliant controllers making it invaluable here. Furthermore, it permits the development and use of custom function blocks and this feature will see frequent use. A detailed explanation of the software is omitted here, but more details can be found on the 4diac website [37].

Chapter 4

Methodology

This chapter provides the primary methodology used in developing the digital twin architecture. Thus, the experimental apparatus, namely the robotic manipulator emulation is first explored. Afterwards, the digital twin architecture is examined, along with a brief display of its functionality.

4.1 Experimental Apparatus

The experimental device used in this work is a three link planer robotic manipulator. This is pictured in Figure 4.1. The manipulator was chosen as it effectively consists of three control loops. As each link of the manipulator possesses a servo motor which controls and measures its position relative to the other links, this manipulator becomes a suitable platform on which to develop and test a digital twin in a DCS. Within this thesis, it is referred to as the *emulation*. To emulate and to simulate are synonymous in many contexts, and the term emulation is adopted here to differentiate the experimental apparatus from the simulation capabilities of the digital twin architecture. Thus, when the when emulation or emulate are applied, they are specifically referring to this emulated manipulator.

In this work, the manipulator is given the task of moving coloured boxes to a designated location based on the colour of the box. This entails path planning, manipulator control, and disturbance mitigation, all of which is explained in detail within this section. Given

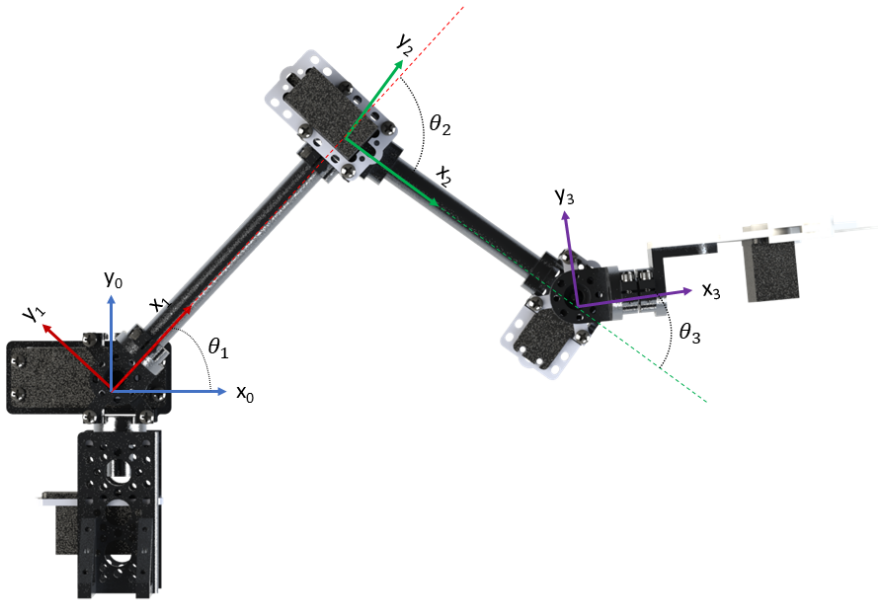

Figure 4.1: The robotic manipulator used in this work. Note that the various local coordinate systems of each link are defined relative to that of the previous link

the widespread details to be covered, this section is separated into discussions surrounding the physics of the manipulator, pertinent details surrounding the boxes the manipulator is required to move, the control strategies employed, and a set of validation tests to demonstrate the emulated robotic manipulator has been properly established.

4.1.1 Detailed Overview of the Robotic Manipulator

The manipulator consists of three links, each of which has a servo motor attached with an accompanying angular position sensor, and a gripper at the end which makes use of a single motor and accompanying position sensor. There is much to describe in the manipulator and these components and the dynamics are a natural starting point. As a precursor for this discussion, due to the size of the equations they are left out of this section and instead provided in Appendix A. The dynamics of the manipulator can largely be described in Equation 4.1.

$$\tau = M(\theta)\ddot{\theta} + B(\theta)\dot{\theta}_i\dot{\theta}_j + C(\theta)\dot{\theta}_i^2 + G(\theta) + D\dot{\theta} \tag{4.1}$$

Where:

- τ is a three by one vector which contains the applied torque by each motor on each link
- θ is a three by one vector which contains the angular positions of the links relative to the previous link, except in the case of the first link which is relative to the global coordinate system. Note that $\dot{\theta}$ and $\ddot{\theta}$ are the angular velocities and accelerations, once again relative to the previous links
- $M(\theta)$ is a three by three mass matrix, which contains terms pertinent to the angular accelerations' influence on the dynamics
- $B(\theta)$ is a three by three coriolis matrix, which contains terms pertinent to the coriolis forces influencing the dynamics
- $C(\theta)$ is a three by three centrifugal matrix, which contains terms pertinent to the centrifugal forces influencing the dynamics
- $G(\theta)$ is a three by one vector which contains the gravitational forces acting on the links
- D is a three by three diagonal matrix, in which each term of the diagonal contains a constant viscous damping constant
- $\dot{\theta}_i\dot{\theta}_j$ is a three by one vector containing the products of different angular velocities (i.e. $\dot{\theta}_1\dot{\theta}_2$ etc.)
- $\dot{\theta}_i^2$ is a three by one vector containing the individual angular velocities squared (i.e. $\dot{\theta}_1^2$)

The detailed expressions of these terms and equations are included in Appendix A. These terms require a number of physical parameters, which will be defined later on.

Before this, however, the dynamics of the servo motors must be defined. There is no ability for a control system to directly apply a torque through a motor, rather the controller can apply a voltage to the motor and thus produce a torque. This distinction is important,

as it offers more sources of uncertainty and subsequent sources of error. These dynamics are linear, and are defined in Equation 4.2.

$$\tau = K_t V_{in} \tag{4.2}$$

Where:

- V_{in} is the input voltage to the motor given by the controller
- K_t is a voltage torque constant which relates the input voltage to the torque generated by the motor. For the sake of simplicity, this includes the dynamics of the gears between the connection of the DC motor and the actual link.

This representation is not uncommon in the use of modelling DC motors which the servo possesses. In the case here, any inductance the DC motor may have is ignored and considered negligible. There is no consideration here regarding any back EMF or the angular velocity of the motor affected by its inertia. In the case here, the back EMF is considered negligible and the inertia of the motor and gears is considered negligible relative to the inertia of the links. Finally, each servo is given a saturation limit of $\pm 8V$, which limits their ability to generate torque.

Next, the gripper's dynamics must be defined. In this research, a simplified model is taken so that the action of the gripper opening and closing may be taken into account. This is described by Equation 4.3.

$$\ddot{\theta}_{gripper} = \frac{V_{gripper} * K_{t_{gripper}} - D_{gripper} * \dot{\theta}_{gripper}}{I_{gripper}} \tag{4.3}$$

Where:

- $\theta_{gripper}$ is the angular position of the line passing through the forward tip of the outer arm of the gripper and this arm's rotation point
- $V_{gripper}$ is the applied voltage to the gripper servo by the controller

- $D_{gripper}$ is a damping constant accounting for viscous friction in the gripper
- $I_{gripper}$ is the moment of inertia of the gripper's link

This representation effectively reduces the gripper into a pendulum rotating about the axis parallel to the direction of gravity, and at the rotation point a motor is present. It does not take into account the more complex dynamics that the gripper has in its multi link mechanism. However, it does achieve the goal of accounting for the action of the gripper opening and closing.

This discussion largely summarizes the physics of the manipulator used here. There is still discussion to be had regarding the sensors, however, for now the relevant physical properties are listed in Table 4.1.

Property Name	**Symbol in Equations**	**Value**
Link 1 Moment of Inertia	I_{z1}	$5.3325e-4kgm^2$
Link 2 Moment of Inertia	I_{z2}	$3.7766e-4kgm^2$
Link 3 Moment of Inertia	I_{z3}	$2.4296e-4kgm^2$
Gripper Inerta	$I_{gripper}$	$0.1kgm^2$
Link 1 Mass	m_1	$0.11804kg$
Link 2 Mass	m_2	$0.11604kg$
Link 3 Mass	m_3	$0.12994kg$
Link Length 1	l_1	$0.1578m$
Link Length 2	l_2	$0.1325m$
Link Length 3	l_3	$0.1608m$
Link 1 Centre of Mass	cg_1	$0.120m$
Link 2 Centre of Mass	cg_2	$0.100m$
Link 3 Centre of Mass	cg_3	$0.140m$
Voltage Constant 1	K_{t1}	$0.2533\frac{Nm}{V}$
Voltage Constant 1	K_{t2}	$0.1573\frac{Nm}{V}$
Voltage Constant 1	K_{t3}	$0.0.1573\frac{Nm}{V}$
Damping Constant 1	D_1	$0.05\frac{Nm}{rad/s}$
Damping Constant 2	D_2	$0.05\frac{Nm}{rad/s}$
Damping Constant 3	D_3	$0.05\frac{Nm}{rad/s}$

Table 4.1: The various physical properties of the robotic manipulator.

Finally, there comes the details surrounding the sensors the effects of digital to analog conversion in the manipulator. Sensor use is necessary for both control and digital twin construction. However, this comes with limitations in accuracy, precision, and sampling. Furthermore, it adds the challenge of analog to digital and digital to analog conversion. As

such, it it absolutely necessary to clarify the assumptions used in the emulation surrounding the sensors.

In the research here, two simplifying assumptions are used. First, each arm possesses an absolute encoder with infinite resolution. This eliminates accuracy issues surrounding resolution of the encoder, and eliminates issues with calibrating the sensors about a new location. Second, it is assumed that the resolution provided by the digital to analog conversion, and vice versa, is so fine that the effects of converting the digital value to an analog voltage in the motor may be neglected. Lastly, outside inherent sensor accuracy and digital analog conversion are not the only issues that occur in measurement, others include noise, sensor offset, etc. However, no assumptions are made regarding noise or offset, as these are left as independent variables for testing.

4.1.2 Details Surrounding Box Pick-Up and Drop-Off

The emulated manipulator is given the task of moving boxes of different colours from a common pickup location to a colour specific drop off location. Defining this tasks requires defining the mass of the boxes being picked up as well as the pickup and drop off locations for the coloured boxes.

First, the mass of the boxes are defined in Equation 4.4. This representation includes a random number generated in MATLAB between 0 and 1, and in this manner the mass of the box has a degree of uncertainty. This reflects real world conditions.

$$m_{box} = 200g - 10g + 20g \times RAND \tag{4.4}$$

Next, the pickup and drop off locations are defined. These are outlined in Table 4.2, which gives the coordinates of the various locations of interest relative to the same origin from which the first link of the manipulator originates.

Box Colour	X Coordinate (m)	Y Coordinate (m)
Pickup	0.2	−0.2
Blue	0.1	0
Red	0.15	0.15
Green	0.15	0.28

Table 4.2: The various locations for box pickup and generation in Cartesian coordinates relative to the manipulator

4.1.3 Control Strategy

With the details of the boxes established, a control strategy may be defined. First, the positions to which the manipulator must move can be defined based on the previously defined box locations. However, as there are two coordinates in the Cartesian coordinate system used to define the box position, and there are three links on this manipulator, there are, by extension, and infinite number of joint angles which may be used to reach each of the box locations.

To resolve this, the third joint angle is defined such that the third link is parallel with the x-axis of the global coordinate system. This last relationship allows the definition of a system of three equations as seen in Equations 4.5-4.7.

$$\theta_1 + \theta_2 + \theta_3 = 0 \tag{4.5}$$

$$l_1 sin(\theta_1) + l_1 sin(\theta_2) + l_1 sin(\theta_3) = y_{box} \tag{4.6}$$

$$l_1 cos(\theta_1) + l_1 cos(\theta_2) + l_1 cos(\theta_3) = x_{box} \tag{4.7}$$

This system of equations was solved using a non-linear numerical solver. This leads to a set of angles to which the controller must move to in order to reach the desired location. These resulting angles are given in Table 4.3.

As is typically seen in the case of robotics control, moving between positions is achieved by following a designated path between two points. In this thesis, a cubic function is used to define this path, which is given in Equation 4.8. Note that the time t, is defined relative to a control time. That is, when the arm begins motion, the control time is zero, and when

Box Colour	θ_1 **(rad)**	θ_2 **(rad)**	θ_3 **(rad)**
Pickup	5.6133	−1.5931	−4.0201
Blue	2.1855	−3.5259	1.3404
Red	2.5907	−2.2251	−0.3656
Green	1.8516	−0.5321	−1.3915

Table 4.3: The solutions to equations 4.5-4.7.

arm completes the motion, the control time is equal to a control target time, t_{target}.

$$\theta_i(t) = a_{i3}t^3 + a_{i2}t^2 + a_{i1}t + a_{i0} \tag{4.8}$$

Where:

- $\theta_i(t)$ is the desired angular position of the i-th as a function of time
- t is the elapsed time since the trajectory started
- a_{in} is a coefficient needed to define the path

This naturally leads to the question of how are the various coefficients are defined. This requires some thought about the initial and final positions. Namely, four conditions can be used. First, at the start of the motion, that is $t = 0$, the angular position is the starting position. Second, at the start of the motion the velocity of the link is zero. Third, at the end of the motion, that is $t = t_{target}$, the angular position is the target position. Finally, at the end of the motion the velocity of the link is zero. With that established, the various coefficients can be defined in Equations 4.9-4.12.

$$a_{i3} = -2\frac{\theta_{i_{target}} - \theta_{i_{intitial}}}{t_{target}^3} \tag{4.9}$$

$$a_{i2} = 3\frac{\theta_{i_{target}} - \theta_{i_{intitial}}}{t_{target}^2} \tag{4.10}$$

$$a_{i1} = 0 \tag{4.11}$$

$$a_{i0} = \theta_{i_{initial}} \tag{4.12}$$

With the trajectory established, a control strategy becomes the next logical item to define. In this research, computed torque control is applied to move the manipulator along the planned path. The control strategy may be described in Equation 4.13.

$$\tau(t) = M(\theta(t))\ddot{\theta}_d(t) + B(\theta(t))\dot{\theta}_i(t)\dot{\theta}_j(t) + C(\theta)\dot{\theta}_i(t)^2 + G(\theta(t)) + K_p e(t) + K_i \int e(t) + K_d \dot{e(t)} \tag{4.13}$$

Where:

- $\ddot{\theta}_d(t)$ is the desired angular acceleration of the link at the given point in time along the path
- θ is the actual angular position taken from measurements
- $e(t)$ is the position error at a given point in time along the path
- K_p, K_i, and K_d are proportional, integral, and derivative controller gains respectively

In this equation the error is defined, at a given point in time along the path, as the difference between the desired angular position as computed in Equation 4.8 and the actual measured position. This is summarized in Equation 4.14.

$$e(t) = \theta_d(t) - \theta(t) \tag{4.14}$$

The controller gains are left for definition later on.

In this manner, the controller is effectively a feedback and feedforward combination controller. The understood physics of the manipulator are used to estimate the required torque, and a small feedback loop is used augment that signal as necessary and eliminate the error. As a final note, while this calculates the torques, the controller actually sends voltages to the manipulator using the relationship defined in Equation 4.2.

While these equations do explain the bulk of the control strategy employed in this work, they are all defined in the continuous time domain. In reality, the controller is a sample data system, taking position measurements from the sensors at discrete intervals and updating its

control signal at discrete intervals. This leads to the challenge of how to implement Equation 4.13 in the sample data domain. This is achieved in Equation 4.15.

$$\tau_i = M(\theta_i)\ddot{\theta}_{d_i} + B(\theta_i)\dot{\theta}_{j_i}\dot{\theta}_{k_i} + C(\theta_i)\dot{\theta}^2_{j_i} + G(\theta_i) + K_p e_i + K_i \int e_i + K_d \dot{e}_i \tag{4.15}$$

Where i is the i-th time step.

This implementation, in combination with the fact that only positions may be read from the sensors on the manipulator, means that all the derivatives and integrals must be calculated using numerical techniques. In the case of the derivative, a second order implementation is employed and shown in Equation 4.17. Similarly, in the case of the integral a first order implementation is employed, and shown in Equation. These implementations are shown with the focus on the error derivative and error integral.

$$\int e_i = \int e_{i-1} + e_i \Delta t \tag{4.16}$$

$$\dot{e}_i = \frac{3e_i - 4e_{i-1} + e_{i-2}}{2\Delta t} \tag{4.17}$$

Where:

- e_i is the error at the i-th time step
- Δt is the time step defined in the controller

Equation 4.17 employs the same numerical derivative that is used in calculating the angular velocities.

IEC61499 Control Implementation

It is pertinent now to discuss the manner in which this control strategy is implemented in the IEC61499 controller. Some features will be displayed here which are pertinent to the interface with the digital twin, and will not be explained in great detail at the moment.

First and foremost, the controller interacts with the manipulator through a server SIFB, shown in Figure 4.2. This works through the use of Marc Jakobi's tcpip4diac package [38]

which permits the transmission of data to and from MATLAB through a TCP/IP protocol. This function block receives six sensor readings, and sends five signals to the MATLAB emulation. The sensor readings include position measurements for the three links and the gripper, a true or false value for the presence of a box in the queue, and lastly a reading from the colour sensor. The signals that are sent through this block include four voltages, one for each link and one for the gripper, and finally a true or false value for a box being in the gripper. This final signal is not technically needed in a real device, and it would find no use. However, it makes the implementation of the emulation in MATLAB simpler, as it provides a mechanism to tell the emulation if a box has been picked up by the gripper, and does not force the emulation into a situation which violates its physics or realism.

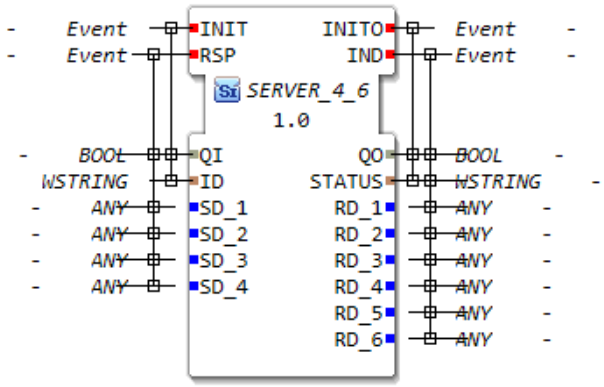

Figure 4.2: The server function block responsible for communication between the MATLAB emulation and the industrial controller

Actual control is based in two function blocks. The first of these is the PATH_PLANNER basic function block, pictured in Figure 4.3. This block conducts two main tasks. First, upon the necessity of a change in the destination of the manipulator, the block will calculate all the parameters required to define the path as outlined in Equations 4.9 through 4.12 and thus define the path, as well as reset the controller and make it prepared for the new path. In addition to this, it will take the required sensor readings needed. Second, it sends the required target location data, desired acceleration, and final destination to the second control block, the computed torque controller block.

This second block, referred to as COMPUTED_TORQUE_CONTROL_3, is responsible for the control of the manipulator. It is shown in Figure 4.4. It takes the data from the

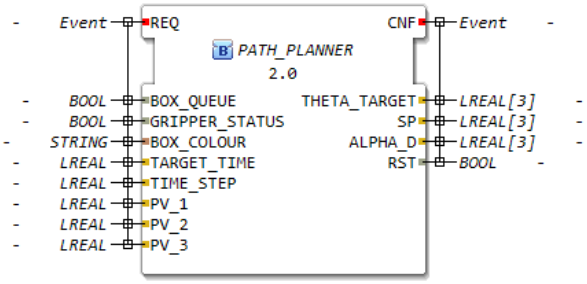

Figure 4.3: The path planning function block

PATH_PLANNER, along with the sensor data from the SIFB, and uses Equation 4.13 to calculate the required control signal. In addition to this, it has data inputs to allow for new controller gains as well as new model parameters. The intention of these inputs is to make use of the digital twin model.

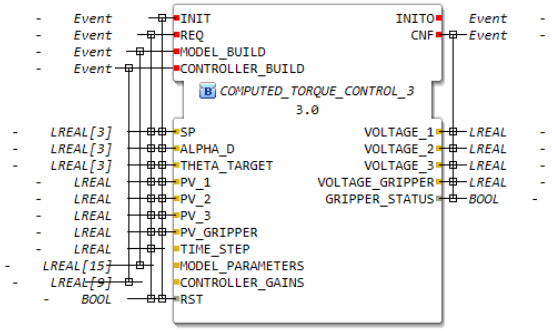

Figure 4.4: The computed torque control function block

4.1.4 MATLAB Implementation

With the manipulator's dynamics and control methodology defined in detail, the method in which the emulation is constructed in MATLAB may be explained. This requires describing how the manipulator is represented, how the boxes are generated, and how these come together to create a real time emulation. As a precursor to this discussion, all of the code used in MATLAB is included in Appendix B.

The manipulator is implemented as an object. This object possesses all of the physical properties of the manipulator, the equations of motion which are implemented as a system of first order ordinary differential equations, and an internal fourth order Runge-Kutta method

which, when called upon, emulates the motion of the arm. The object oriented approach permits all data from resulting simulations to be saved and used elsewhere.

Similarly, the boxes are implemented as objects. These boxes are generated as required, with a randomized colour and mass as defined earlier. Again, this implementation permits storing all pertinent information for later use.

With these two elements established, how they interact may be established. A custom MATLAB application holds the details of the simulation, presents the results in a pseudo real time manner through a graphical user interface (GUI), and manages the execution of emulation details. The GUI is presented in Figure 4.5 as a start to this discussion.

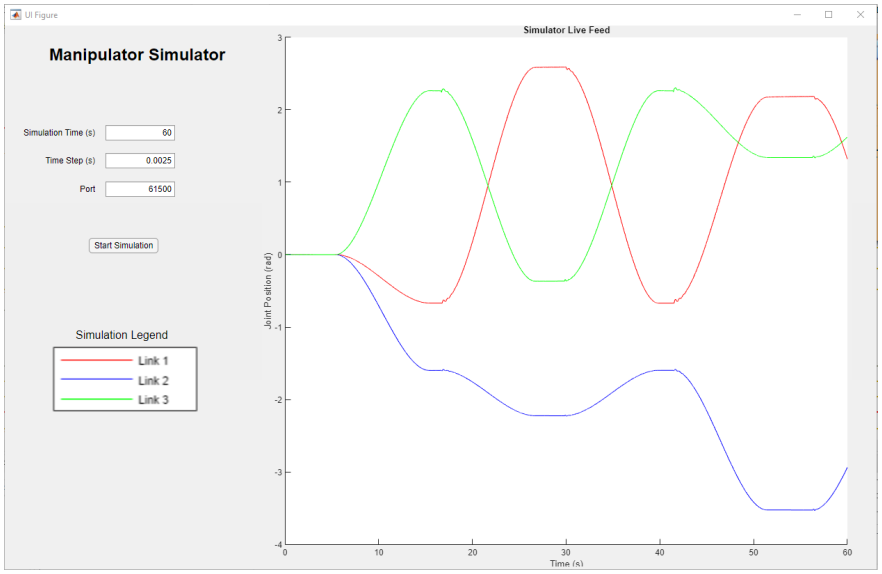

Figure 4.5: The manipulator emulation implemented in MATLAB with its included GUI

The application responsible for the emulation operates in a manner presented in the flow chart in Figure 4.6. At the beginning of an emulation run, the manipulator is initialized, that is freshly defined in the workspace, along with all other pertinent variables required. This leads the to beginning of the emulation itself. Throughout the execution of a time step, a few actions take place. First, it is checked if a box has arrived. If the simulation time aligns with the box generation time, then a box is generated, and if not then this step is skipped. Also note, that a box is only generated if another box is not in the queue.

Next, it is checked if the emulation time aligns with a controller time step. As the controller acts in discrete time intervals, it is only called upon if the emulation time can be

divided by the controller time step into a whole number. For example, if the emulation time

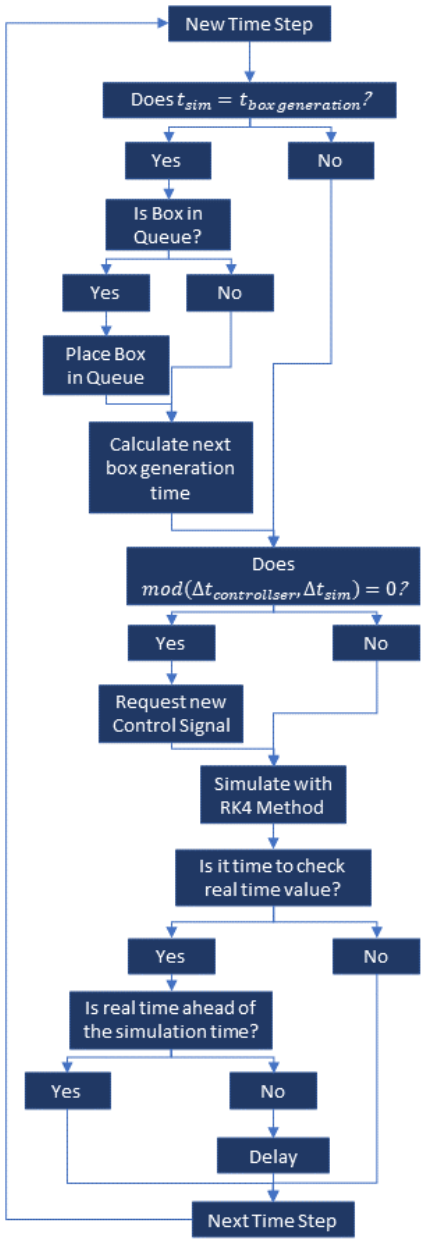

Figure 4.6: A flow chart describing the main emulation

is 15 seconds, and the controller time step is 0.010 seconds, then the controller is called upon for a new signal. However, if the emulation time is 3.0025 seconds, then the controller is not called upon. In this manner, the sample data nature of the controller is implemented and respected. The controller moves the manipulator from its present location to the location of the box queue should a box be present there and no box be in the gripper, and should a box

be in the gripper then the controller moves the manipulator to its destined location.

It is through this mechanism that timing is managed. By controlling when the controller is called upon, i.e. only every 0.010 seconds in emulation time, the discrete nature of the controller is respected. This contrasts with a real system in which a timer in the controller would manage when a signal is sent to the manipulator.

Recall from earlier the controller exists in the IEC61499 standard through 4diac. The MATLAB application calls upon 4diac through a TCP/IP protocol defined in Marc Jakobi's tcpip4diac package [38].

Moving from these checks, the actual emulation takes place. This involves determining the load force based on the box mass should there be one in the gripper, and then calling upon the RK4 method within the manipulator object. This largely completes the emulation process, with the previous loop being repeated until the simulation time is completed.

All that is left to explain is the manner in which real time is implemented. MATLAB has functions to measure the amount of time that a single loop takes to complete, as well as functions capable of pausing the emulation to allow real time to catch up. However, MATLAB is not precise enough to do this pause every loop. To work around this limitation, the amount of time it takes for 25 loops to complete is recorded. If this is less than the real time, i.e. 25 times the time step, then the emulation is paused to allow real time to catch up. Otherwise, the emulation continues. In this manner, a pseudo real time emulation is achieved which allows a user to interact with the manipulator emulation as if it were a real device.

4.1.5 Validation Tests

This section intends to establish the validity of the emulation by examining the physics. Four tests are presented here, which compare the results of the emulation with the anticipated behaviour based on physics. As a precursor to this section, it is noted that all simulations employ a time step size of $2.5ms$.

Arm Drop Test

The first of the four tests is the arm drop test. The other tests are all fundamentally variations of this one. The test begins by keeping all links of the manipulator at a $0rad$ angle in their respective coordinate systems. In this manner, the manipulator is in a horizontal position. The manipulator is then allowed to drop without any application of voltage to the servo motors. In such a scenario, one would anticipate the manipulator to fall and eventually settle such that is was aligned vertically and facing towards the negative y direction in global Cartesian coordinates. The resulting simulation is presented in Figure 4.7.

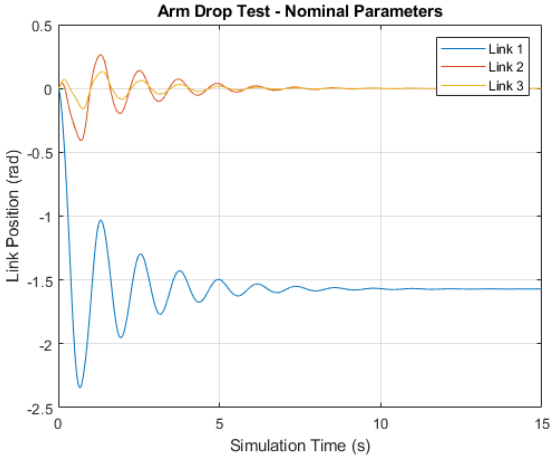

Figure 4.7: The results of the first verification test on the manipulator simulation

The results here align with the expected result. The first link has aligned to $-1.57rad$, that is -90 deg to the global coordinate system, and the second and third link have aligned to that same orientation, that is $0rad$ to the first link. This supports the notion that the physics have been properly implemented in the simulation.

Increased Mass Test

The second of the four tests is effectively the same as the first, except the mass of all three links are increased by 50% while the other parameters are held at their true values. This effectively increases the amount of potential energy the manipulator links have relative to the equilibrium position they will reach. As such, the amplitude of the motion and the settling time can be expected to increase. The results of this test are presented in Figure

4.8.

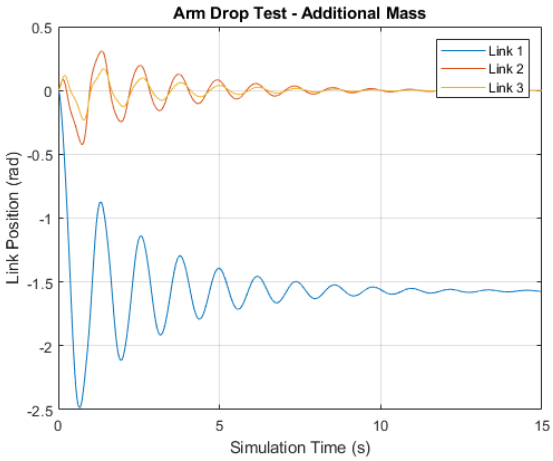

Figure 4.8: The results of the second verification test on the manipulator simulation

The results once again support the notion that the manipulator simulation is properly implemented. The first link reaches a minimum position of $-2.4867rad$ in this simulation compared to a minimum of $-2.3422rad$ in the first verification test, an increase of 6.2%. Additionally, the oscillation of all links continues to the $15s$ mark in this test compared to the first where they all settle around $10s$. Finally, there is no behaviour here that suggests that the physics were improperly implemented, such as a sudden explosion. These results all suggest that the simulation is implemented correctly.

Increased Inertia Test

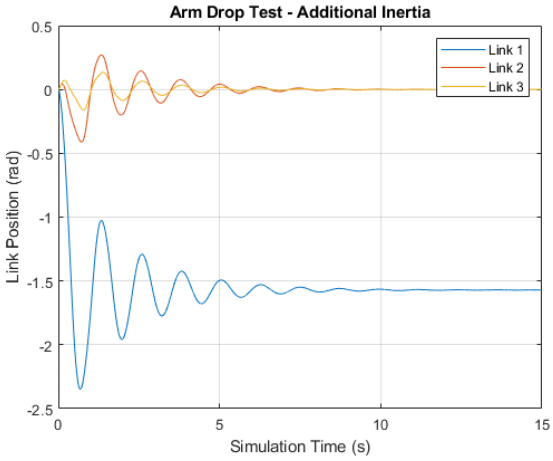

Figure 4.9: The results of the third verification test on the manipulator simulation

In this third test the various moments of inertia are all increased by a factor of two and the other parameters are left at their nominal parameters. This will not add any additional potential energy to the system, but should slow the response of the system as more torque is required to accelerate the various links. The result are shown in Figure 4.9.

These results are harder to discern from the first test compared to the increased mass test. However, close inspection reveals that the minimum position of the first link is reached at $0.6825s$ in this test compared to $0.67s$ in the nominal parameter test. This represents an increase of 1.9%. Thus the response of the system is indeed slower, as was anticipated. Additionally, there is no strange or unexpected behaviour occurring from changing these parameters. All of these results further support that the physics are properly implemented.

Upwards Vertical Test

In the final verification test, the initial conditions are altered from the first test. Rather than starting in a horizontal position, the arm is aligned vertically upwards along the positive y direction. That is to say the arm is in an unstable equilibrium. However, in the absence of any forces, which is only achieved due to the simulated nature of the arm, one may anticipate that the arm retains this position. The results of the simulation are given in Figure 4.10.

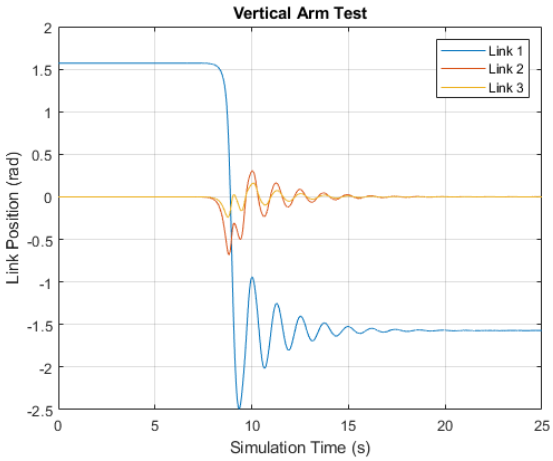

Figure 4.10: The results of the fourth verification test on the manipulator simulation

The initial results of the simulation are in line with the expected result. Without any forces acting on the arm it is able to maintain its position even though that position is

an unstable equilibrium. However, around nine seconds the arm falls down and eventually reaches its stable equilibrium point. While this was not anticipated, it is explainable. In a numerical simulation, like the one of the manipulator, small perturbations in calculations occur due to the nature of representing numbers in computers. This eventually will lead to the arm not maintaining its original position exactly and the arm falling. This is observed here and explains the result. However, the fact the manipulator maintained its unstable equilibrium without the presence of exterior forces for the first nine seconds again suggests that the equations of motions have been properly implemented in the code.

4.2 Digital Twin Architecture

This section aims to establish the digital twin architecture and explain its functionality. This will be linked to the main functions of a digital twin presented in Section 3.2. This discussion is broken into four sections. A general overview of the digital twin architecture is given, followed by a detailed explanation of each module.

4.2.1 General Overview of the Digital Twin

All modules presented in this digital twin architecture have a common feature. This being that they use function blocks in parallel with MATLAB to achieve their complete functionality. This concept is visualized in Figure 4.11.

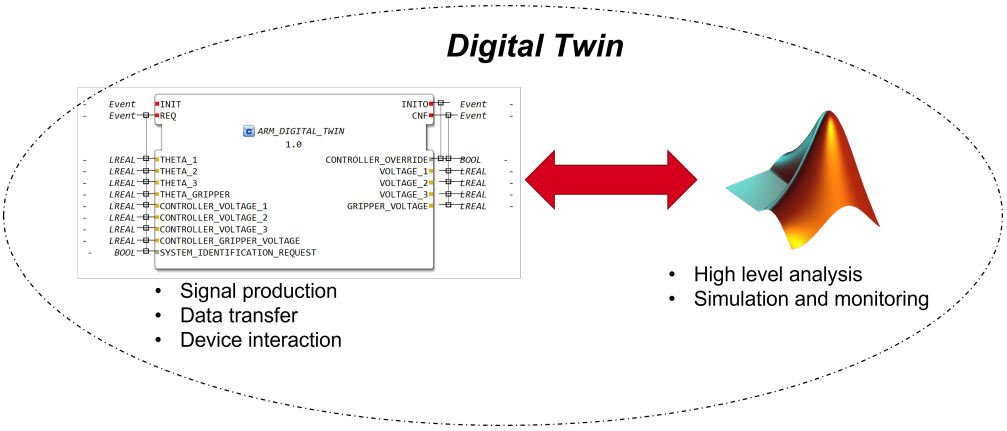

Figure 4.11: A very high level view of the digital twin

MATLAB is employed in this work to complete several main high level tasks which, while possible in IEC61499, are best left to the domain of high level programming languages. These include tasks like system model construction, simulation of motions, and monitoring of the device. The high level of functionality provided by MATLAB means operations like matrix mathematics, large data storage, and data visualization are easily achieved. In this particular work MATLAB is employed because it has algorithms needed to realize the digital twin's functionality. This is not to say that MATLAB is the only language capable of completing the task of building a digital twin, and that should be noted.

There is another key reason for using a high level programming language. Software with such functionality would certainly be located on a computer other than industrial controller itself. It would be very problematic if an error in a highly complex program were to cause an industrial controller to crash. Thus, even if one were to construct all the functions previously described in a function block application, the application would still be located on a computer other than the one used for the controller and a communication protocol between the two would still be required. At that stage, it makes sense to take advantage of a high level programming language as any benefit of completely collapsing the twin to a single computer is lost.

This does not eliminate the need for function blocks in the digital twin. Function blocks provide a ready and simple method for interfacing with the physical device and, for that matter, the rest of the DCS. This greatly simplifies the implementation of a digital twin and is best visualized in Figure 4.12. In an exclusively high level approach, the digital twin needs two separate communication pathways, one to the DCS and the other to the device. This brings in a level of complexity which can more easily lead to faults occurring. In contrast to this, employing function blocks reduces this to a single communication pathway between the higher level side of the digital twin and the function block element. This function block element is then responsible for all device interfacing and working with the rest of the DCS, and greatly simplifies the implementation.

In addition to this, the function block element of the digital twin can exist on the industrial controller responsible for regular control of the device. Again, this simplifies interfacing

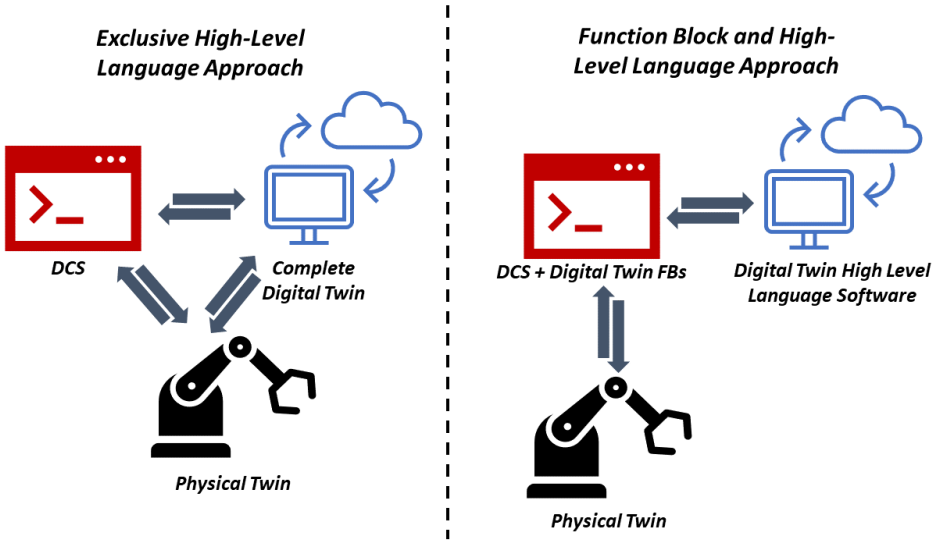

Figure 4.12: Two possible ways of constructing the digital twin. Note in the approach adopted in this thesis, only two communication pathways are required.

with the rest of the DCS. More importantly though, it provides a level of robustness crucial for safety critical tasks. One of the functions listed for a digital twin in a DCS was its ability to detect hardware failure. This may be followed by an emergency shut down procedure. Such a procedure must be left to the function blocks, as it eliminates potential failures in communication between the physical device and high level programming element of the twin, as well as possible failures occurring from bugs in the more complex code. As a final note on this particular element, the function block element of the twin also allows robust production of signals required for various operations, like system identification, which will be demonstrated later on.

All elements of the digital twin make use of this approach in some manner, though they vary in their application of the function blocks. It is of important note, however, that all function block elements of each module are collapsed into a single CFB. This is shown in Figure 4.13. In this manner, the full capacity of employing the hybrid approach is realized and a ready method for interfacing with the DCS and physical device is achieved.

A final note must be made on the digital twin architecture presented here. All modules of the digital twin act as separate programs, but link back to the core model which is stored as an object and saved to the computer which all the modules run on. When required, the

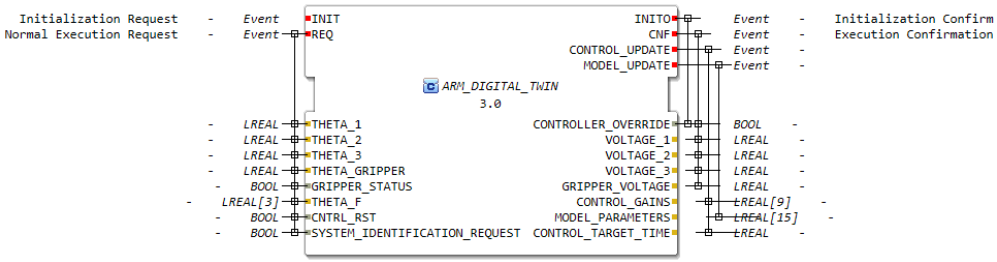

Figure 4.13: The CFB of the digital twin, which includes all the function blocks and their various networks that are included here

digital twin model is called upon by the module in need. In this manner the modules can operate in parallel to each other.

4.2.2 System Identification Module

The system identification module is arguably the most critical of the three built and presented in this thesis. Additionally, it is the most complex, making full use of MATLAB in its task along with function blocks in capacities beyond communication. Its critical nature is due to its role in constructing the high fidelity model which the other modules will use in their tasks. Its success or failure here will define the efficacy of the rest of the digital twin.

To fully explain this module, four discussions will take place. First, the manner in which the high fidelity model is constructed by the module is established. Here, this is grey box modelling. Next, the implementation in IEC61499 is provided, and the discussion is included with the implementation of the module in MATLAB.

Grey Box Modelling

Earlier in this thesis a definition for a digital twin was given. That definition stated that a digital twin was a software built around a high fidelity model constructed from the understood physics and available sensor data. This must be respected in whatever capacity a digital twin is implemented.

In applications of smart manufacturing involving control of a device, the behaviour of interest is most commonly the system's dynamic response to an input. Modelling such a response falls into the realm of system identification, hence the name of the module.

System identification aims to produce a model which effectively replicates a system's dynamic response to an input. Generally speaking, three types of models can be produced with system identification including white box, black box, and grey box models.

White box modelling makes use of the believed physics of a system exclusively. It is a model in the manner that one would typically think of it. Believed properties and physics are combined to estimate the dynamics of the system. The word 'believed' is used here as no physical property can be measured exactly nor physics known perfectly. This leads to the main challenge with white box modelling, and hence why it is not employed beyond early designs. This challenge, in combination with the fact it makes no use of sensor data, makes it unsuitable for a digital twin.

In contrast with white box modelling, black box modelling attempts to replicate the behaviour of the system with no assumptions made about the underlying physics. This strategy makes use of the sensor data obtained from the device in combination with the recorded input signal. This information, followed by a series of mathematical operations, establishes a model which produces an output based on an input. Arguably, this approach encompasses most regression type models which may find use in other digital twins. However, its key fault is it takes no advantage of the believed physics. This may lead to faults in the model, like an instability, which were not caught because the black box modelling algorithm never obtained data needed to establish complete behaviour.

Grey box modelling falls in between the two extremes of black and white box modelling. Grey box models make use of similar algorithms seen in black box models, and combine those with the believed physics which are seen in white box models. In this manner they employ all available and applicable sensor data along with the believed physics of a system. This makes them a good fit for use in digital twins, and hence is why the approach is applied here.

As a final note in this particular discussion, grey box modelling is employed here because it produces a model suitable for the application in question and is also in line with the definition of the digital twin. It is not intended as an approach for every digital twin. Other digital twins may make use of state estimators, stress-strain models, or chemical processing

models in combination with applicable sensor data. The takeaway from this discussion is that however a digital twin is implemented, it must not make use of sensor data or physics exclusively. Such an approach will likely not produce a model with the same capability as a combined approach, and in essence defeats the purpose of a digital twin.

Function Block Implementation

The function block used in the system identification module is, at its highest level, presented in Figure 4.14. This is the surface level of a CFB which includes all the functions required to complete the grey box model construction. At this level, there are inputs for the various position measurements from the manipulator's encoders, and there are outputs for the voltages sent to the manipulator.

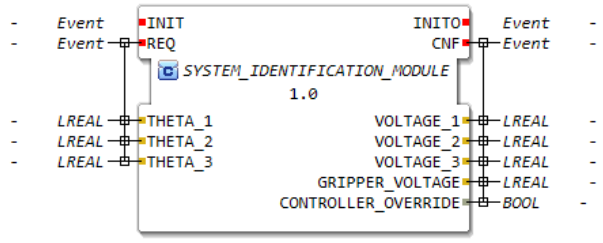

Figure 4.14: The CFB of the system identification module, which interfaces in the digital twin function block.

The network of this CFB is given in Figure 4.15. This level of the module's implementation has the task of receiving a request to begin the identification procedure from the GUI in the MATLAB, which is done through the SERVER_0_1 block, and activate a block called SYSTEM_IDENTIFICATION_BLOCK. This block will be examined in more detail momentarily, but it is responsible for sending out the signal used in the system identification process and collecting the data for the grey box algorithm.

The SYSTEM_IDENTIFICATION_BLOCK is another CFB, with its function block network shown in Figure 4.16, which is responsible for transmitting data to the MATLAB application as well as producing the signal used for system identification. In this work, a modified 3211 signal is applied to the manipulator by the sys_ident_3211sig_v2 block. A 3211 signal sends a $+1$ input signal to a device for three time steps, then a -1 input signal for

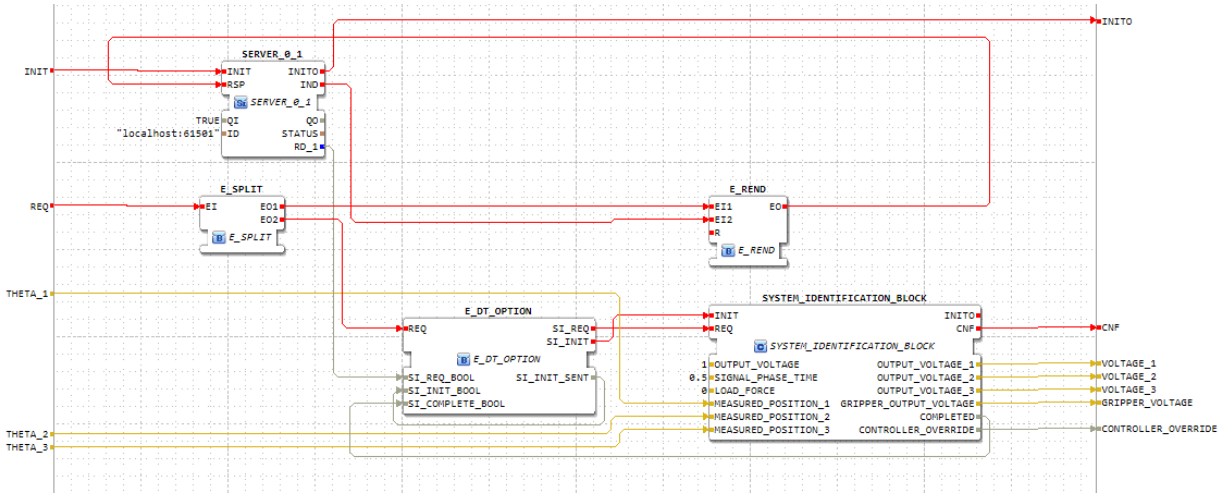

Figure 4.15: The internal network of the system identification module function block.

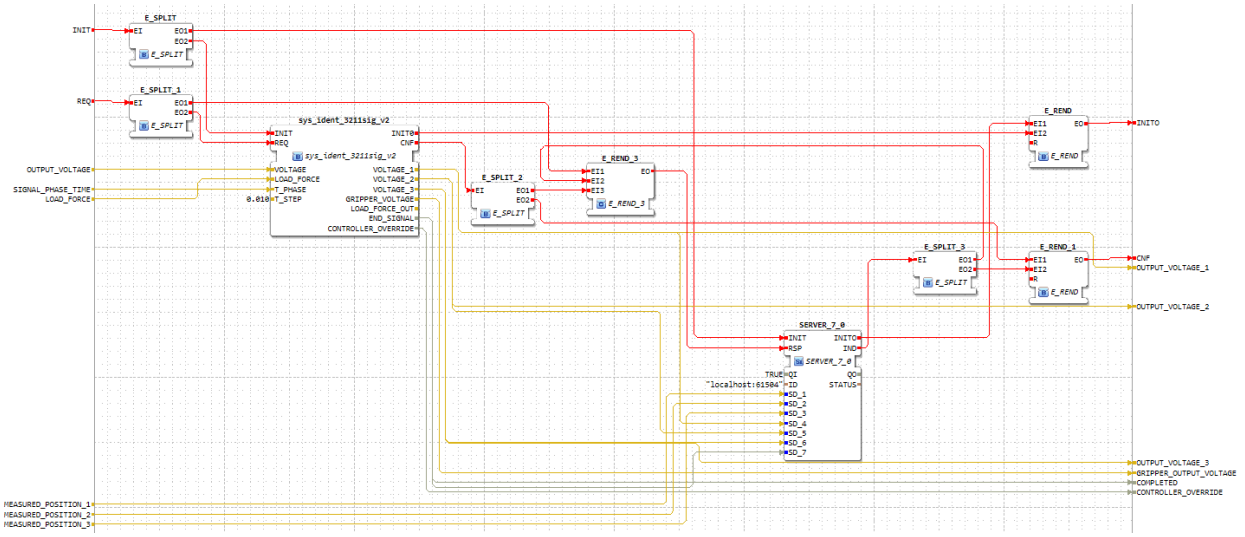

Figure 4.16: The internal network of the system identification identification function block, responsible for sending signal data to the manipulator as well as MATLAB.

two time steps, and then a $+1$ input signal followed by a -1 input signal each for one time step. This signal is used in system identification applications as a substitute for a random signal, where a random signal may cause damage to the system being identified. The 3211 signal function block thus produces a model independent signal which allows the grey box identification algorithm in MATLAB to estimate the parameters. The SERVER_7_0 block transmits both the signal as well as the position measurements to the MATLAB application. Through this channel all necessary information is transmitted to complete the grey box modelling process. Note that the events are oriented such that the signal is sent to the device and not dependent on the successful transmission of data to MATLAB. This ensures

that the manipulator continues to receive the signal, regardless of whether or not a response is received from MATLAB acknowledging the previous data was received. This is possible due to the event based nature of IEC61499.

A final note needs to be made in the discussion of the function block implementation of this module. It may be noted that in the previous few figures there is an output called CONTROLLER_OVERRIDE. This is used in combination with the block called DATA_SELECTOR, shown in Figure 4.17 in combination with the system identification module block and controller. This block is needed here, as it overrides the controller signal and allows for the system identification signal to be sent through to the manipulator. This feature is key in any digital twin where the twin is designed to directly interact with the device in question.

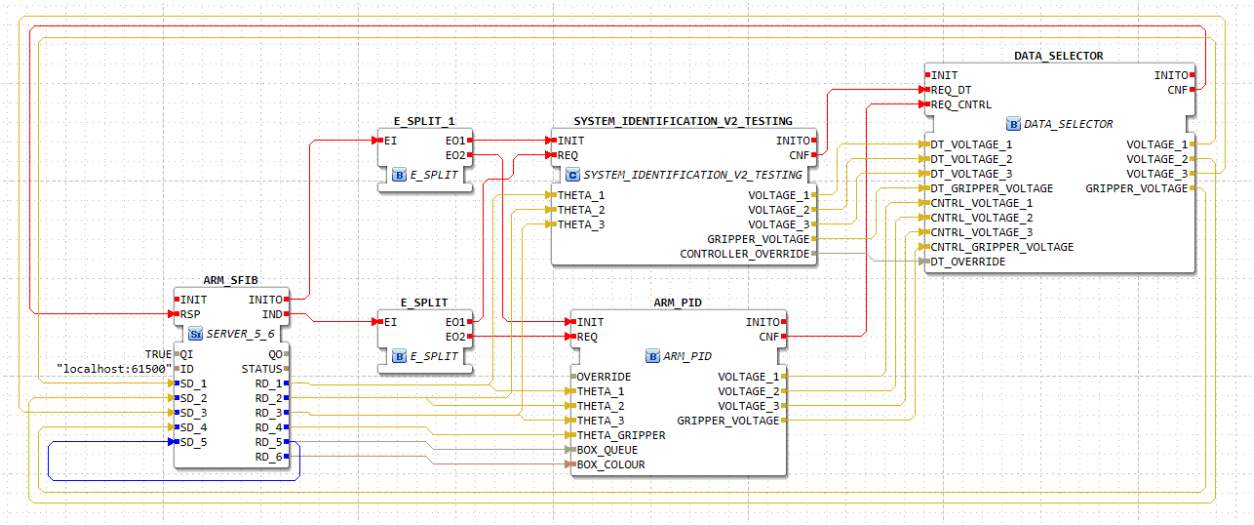

Figure 4.17: The data selector block shown in configuration with the other blocks needed for the system identification module function.

MATLAB Implementation

The MATLAB implementation of the system identification module completes the complex analysis required to estimate the various parameters listed in Table 4.1. At the user level, that is the highest level which an operator would use, is the module's GUI shown with some results in Figure 4.18. At this high level a user simply needs to enact the algorithm by hitting the 'START SYSTEM IDENTIFICATION' button, which informs the function block element of

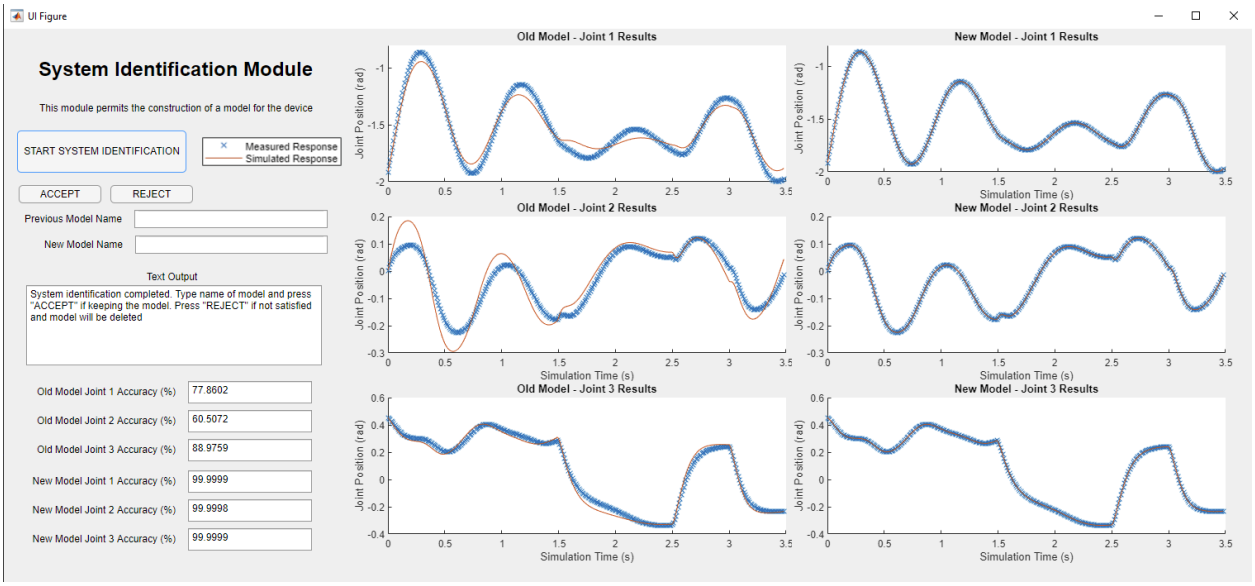

Figure 4.18: The MATLAB GUI of the system identification module, here with some results from testing, which shows the model's accuracy compared to the collected data

the module to begin the system identification signal generation and transmission. This enacts the algorithm, which will be given more attention momentarily, and upon completion of the algorithm the use decides whether or not to save the model based on the results presented. This model then becomes the core of the digital twin and can be accessed through the file name by the other modules in their appropriate tasks.

While the GUI presents a clean form for managing the system identification process, much occurs in the background which is now of focus. Upon activation of the module, it immediately begins storing position and signal data in line with the time step of the controller, which is $0.01s$. This provides all the necessary data required to complete the system identification process.

The second major step is to define the initial state of the manipulator. As noted earlier, the manipulator has three absolute encoders on it and thus only position and time data is available. Additionally, the model which MATLAB uses in estimating the properties of the manipulator is effectively the same as the dynamics of the system, referenced earlier in Equation 4.1. The descriptor 'effectively' is employed deliberately, as testing later on will aim to examine the effects of uncertainty in the dynamics. Thus, the model is a system of first order ordinary differential equations which contains six state variables, three positions

and three velocities.

Defining the initial state thus requires an estimation of the initial velocities based on available position data. In this work, numerical differentiation is employed using a sixth order forward finite difference scheme displayed in Equation 4.18. This is an important note in this work, as it constructs the digital twin with the available sensors, thus taking into account real limitations of existing hardware.

$$\dot{\theta_0} = \frac{-\frac{49}{20}\theta_0 + 6\theta_1 - \frac{15}{2}\theta_2 + \frac{20}{3}\theta_3 - \frac{15}{4}\theta_4 + \frac{6}{5}\theta_5 - \frac{1}{6}\theta_6}{\Delta t} \tag{4.18}$$

The next step is where the estimation actually takes place. The actual grey box estimation algorithm is enacted with all the data collected. In this work this is achieved using MATLAB's internal non-linear grey box estimation algorithm, *nlgreyest*. This algorithm may or may not fail, and should it fail then the algorithm must be stopped and the process restarted.

Upon completion of the algorithm the results are presented in the GUI along with curve fits. At this stage, one may decide to use the produced model for the digital twin and save it, or discard it. This largely concludes the manner in which the MATLAB element of the module functions.

4.2.3 System Monitoring Module

The system monitoring module, in contrast to the system identification module, is a passive system which only collects data from the physical device via its function block interface. It does not send data back to the manipulator. Similar to the previous discussion, this will be separated into two separate examinations, one into the function block implementation and the other into the MATLAB implementation.

Function Block Implementation

Like the system identification module, the system monitoring module function block is also a CFB, its highest level is presented in Figure 4.19. It has several data inputs including the various sensor readings from the encoders of the manipulator, the gripper status, whether

the controller has reset, the target destinations from the path planning function block, and an override which allows the system identification module to override and pause the system monitoring module.

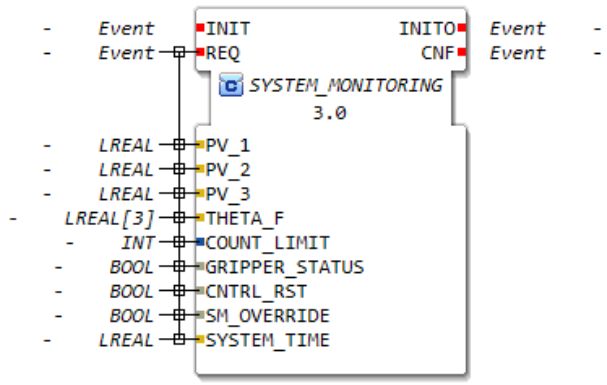

Figure 4.19: The highest level of the function block for the system monitoring

Two data inputs that are somewhat less self explanatory include the SYSTEM_TIME and the COUNT_LIMIT variables. As the system monitoring task may be paused, such as by the system identification module, the module has no real mechanism to know how long the manipulator has been running. The only recourse is for the monitoring module to obtain the running time from the remaining DCS, and hence is why the system time variable exists.

The COUNT_LIMIT variable exists due to timing challenges. The amount of time needed to transfer data from the function blocks of the digital twin to its MATLAB counterpart is not zero. As such it happens that all the key data cannot typically be transferred every single time step, especially in the case of this simulation which completes other operations using the transmitted data. The count limit variable permits one to adjust how many time steps pass before new data is transferred over. As a consequence of this, the system monitoring module checks its simulated values against the actual values obtained from the encoders in discrete intervals, rather than continuously.

The internal function block network of the system monitoring block is shown in Figure 4.20. There are a few blocks here which are of note. The first it the EVENT_COUNTER block, which is a custom block and only allows for the CNT_CMP event to be sent once the number of REQ events received reaches the count limit, and thus provides the mechanism needed to control how frequently data is transmitted to MATLAB. There is also an

EVENT_SWITCH block, which permits the override variable to prevent data from being transmitted should it be requested. Of final note, there is a SERVER_8_0 which is responsible for sending data to the MATLAB element of the module. The remaining blocks are mainly for bookkeeping so to speak, ensuring that the event timing is maintained and the data is transmitted in a digestible form for the tcpip4diac package.

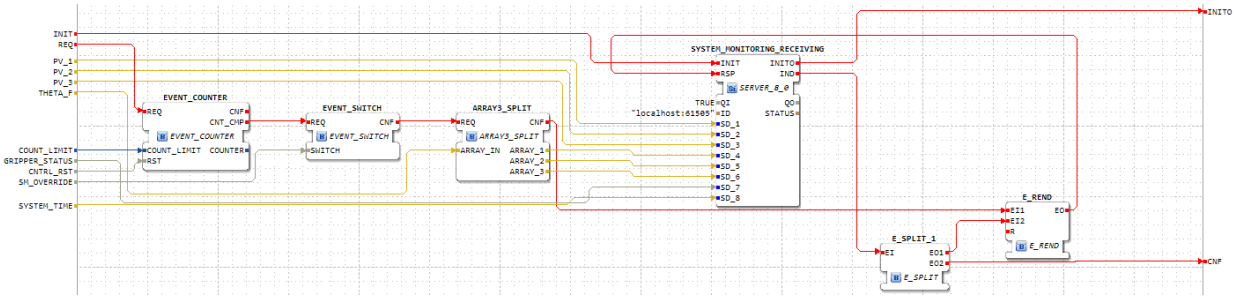

Figure 4.20: The system monitoring module block's internal network

MATLAB Implementation

The MATLAB implementation of the system monitoring module is more complex than its function block side. Once again, this discussion will be opened with an image of the GUI, shown in Figure 4.21.

In this figure, a few key elements should be pointed out. First, there is a spot to type in the model name. This is a model defined by the system identification module. Next, each link has a message centre which gives a written message on the status of the manipulator each with an accompanying graph which visualizes the simulation results and sample data in real time. Finally, there is a toggle switch which permits the user to start or stop the system monitoring process on demand.

The algorithm itself is more complex in its working, and is summarized in the flow chart given in Figure 4.22. The module makes use of the model to simulate arm motion in much the same manner as the manipulator simulation operates, along with an RK4 simulation strategy, and makes use of the same control algorithm as the industrial controller. As such, these aspects will not be given much attention at this point. Of more interest is how the module goes about comparing the simulated positions with the collected data.

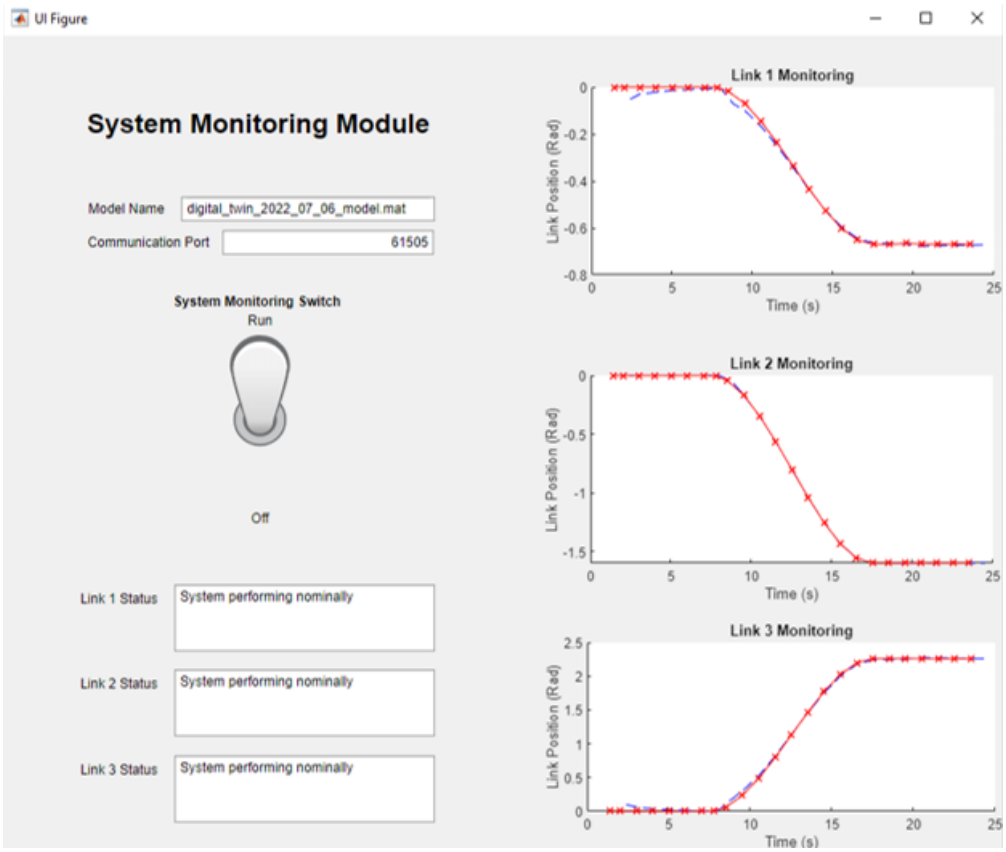

Figure 4.21: The system monitoring module's GUI in MATLAB

Once it becomes time to compare the motion of the arm with the simulation done by the monitoring module, as activated by the function blocks, a series of actions take place. First, the time in the simulation and the actual time are compared and any deviation is corrected. Then the module checks if a new motion has started by comparing the target destinations with the ones previously recorded. Should they be different, the module will update the position of all links to the measured values and thus note that a new motion has begun. Should they be the same, then the module will conduct a comparison between the simulated and measured values.

The comparison is largely arbitrary, and the criteria used here is selected here to attempt to alert of a failure occurring. Those applying this approach could, of course, adjust the criteria based on the application. In this work, at each point where a comparison takes place, two errors are calculated. These are a relative error and absolute error, defined in

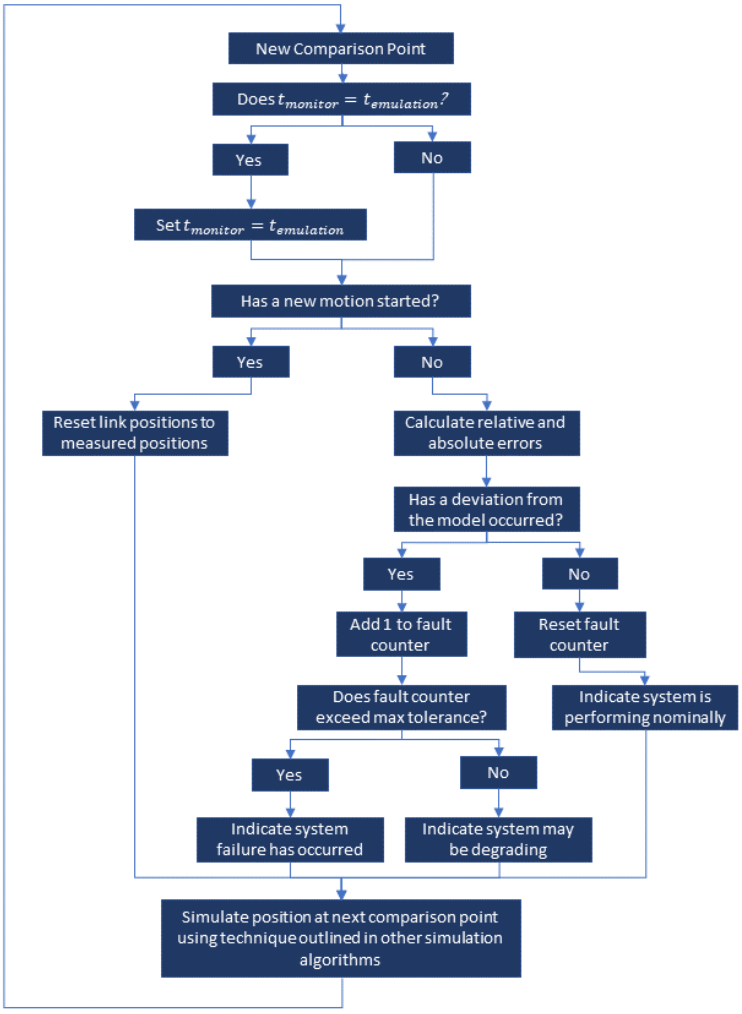

Figure 4.22: A flow chart describing the algorithm that the system monitoring module uses

Equations 4.19-4.20.

$$Rel.Error = \frac{|\theta_{sim} - \theta_{measured}|}{\theta_{measured}} \tag{4.19}$$

$$Abs.Error = \theta_{sim} - \theta_{measured} \tag{4.20}$$

If both of these error values exceed a set tolerance, then the monitoring module alerts the user that a deviation from the digital twin has been detected. However, this is not necessarily cause for alarm. For example, sensor noise may be the cause of a deviation and will vanish at the next comparison point. Thus, the module only alerts the operator via the GUI that the system failure has likely occurred if a consecutive set of deviations is detected, and it manages this through a counter. This counter is initially set to zero, and is reset once

deviations stop.

The module will then go about calculating the anticipated location at the next comparison point. It is through the comparison of destination coordinates and updates in the simulation time that the module is able to align its simulation with the actual events occurring in the manipulator. It is reiterated here that the approach described here is very much application dependent, and other applications will require different monitoring techniques. The validity of such an approach is discussed further in the Testing Plan.

4.2.4 System Simulation Module

The final module of the digital twin is the system simulation module. It differs from the previous two in that it does not receive any signals from the manipulator itself. Rather, it only sends signals to the device. This will be elaborated on momentarily.

Function Block Implementation

The highest level of the function block is given in Figure 4.23. At this level, it is obvious that there are no data inputs from the manipulator being sent to the module. However, there are a few key data outputs including the controller gains, model parameters, and control target time. The goal in the design of this function block is to allow an operator monitoring the manipulator's operations to apply the model from the system identification module in the computed torque control function block, as well as upload new controller parameters based on the results of simulation.

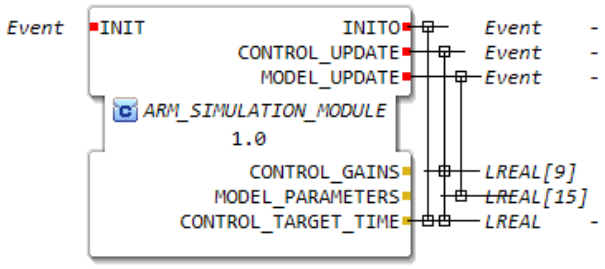

Figure 4.23: The highest level of the CFB of the system simulation block

The internal network of this function block is provided in Figure 4.24. This network is

rather simple, consisting of two server blocks as well as blocks consolidating the data outputs of those blocks into the outputs of the CFB. This demonstrates a key strength of IEC61499, in that complex functionality can be consolidated into a single block and applied to a useful function.

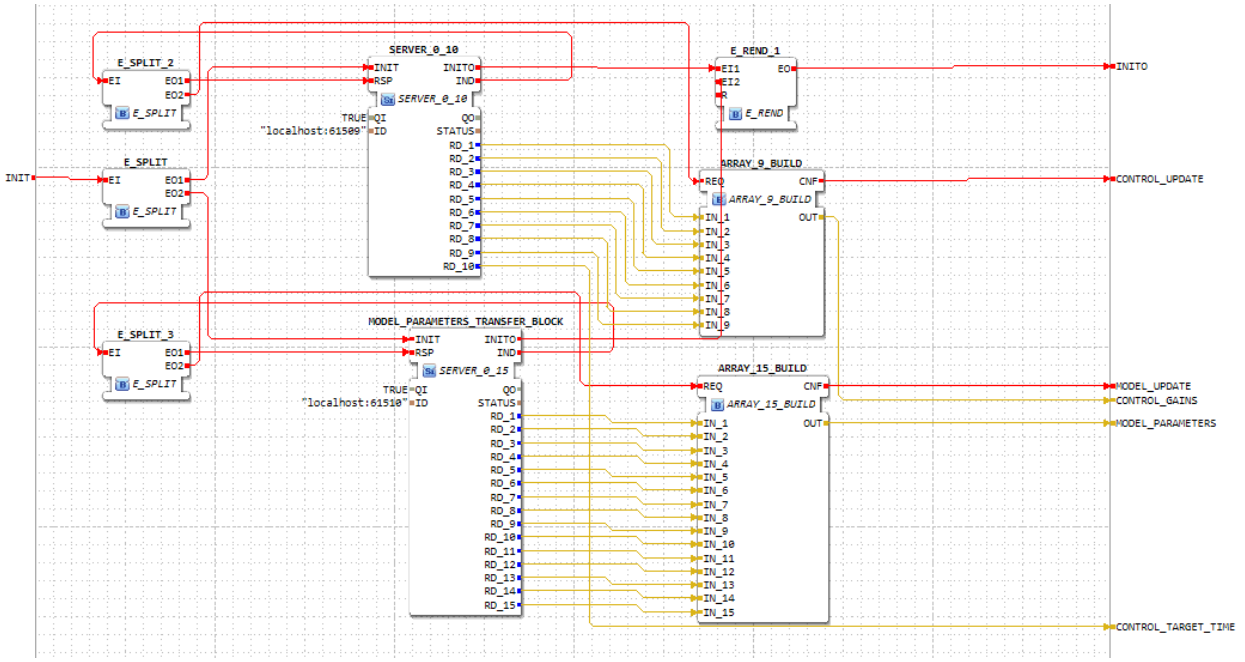

Figure 4.24: The internal network of the system simulation block, which permits the deployment of the controller and model parameters to the controller function blocks

MATLAB Implementation

Like the other modules the system simulation module possesses a GUI which conveys key data to the user, and this is shown in Figure 4.25. Within the GUI there are several key features. On the right side, there are three windows which permit the user to view the results of a simulation. Like the system monitoring module, there is an ability to select the model as well as configure the simulation parameters. Lastly, there are entries which allow the user to configure the controller and attempt to find an optimal control point.

There are two buttons located at the bottom of the module's GUI. The buttons allow the user to upload the controller parameters or the model parameters through the server blocks in the function block implementation of the module. In this manner, the digital twin has the ability to directly interact with the control system and put its model to use.

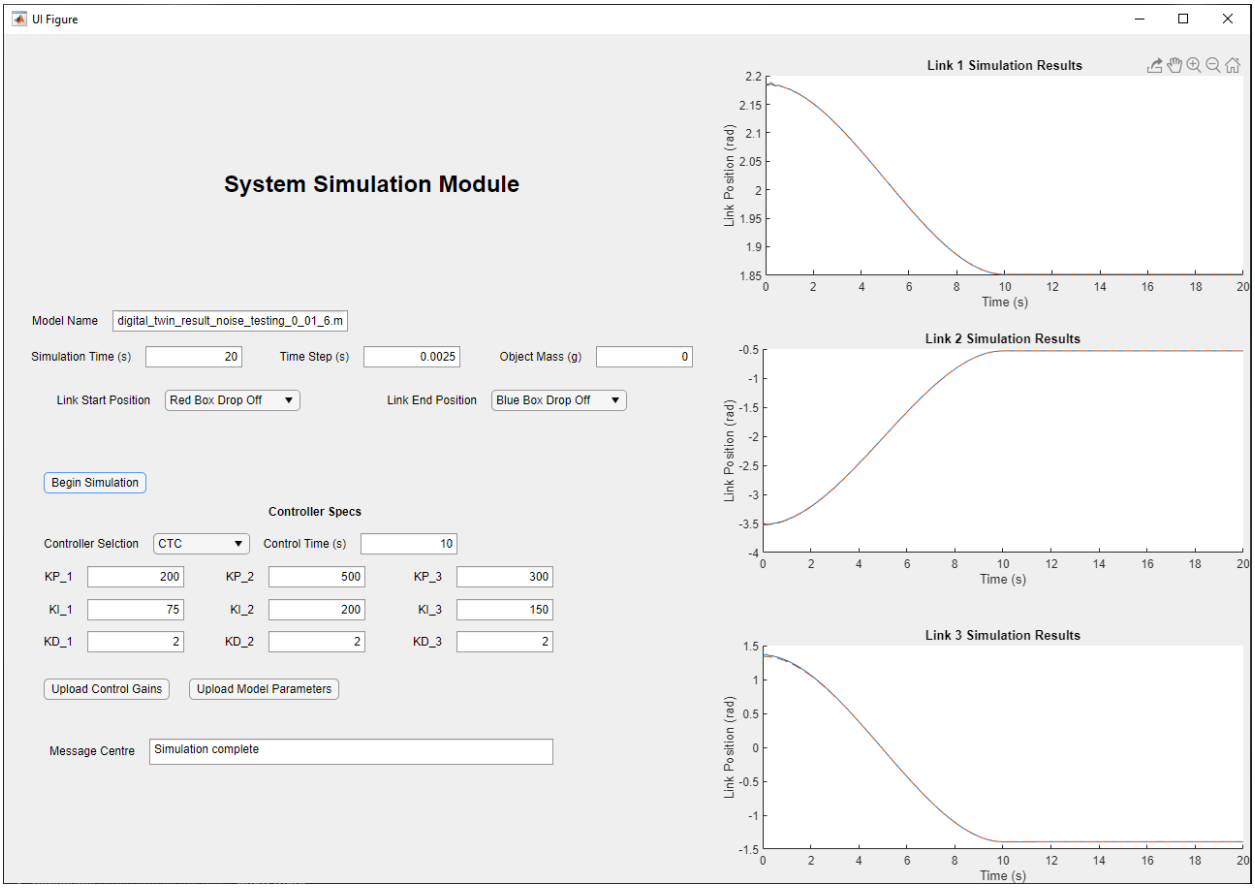

Figure 4.25: The system simulation module GUI which shows its major features

Chapter 5

Results and Discussion

This chapter contains the testing methodology, results, and discussion for all three modules developed in this thesis. This chapter intends to employ the digital twin architecture and emulated manipulator to test the limitations of the proposed architecture in the proposed modules.

5.1 System Identification Module

The model employed by the digital twin is, arguably, the most critical piece in making the representation useful. In this architecture, the system identification module is made exclusively responsible for determining this model. Given this crucial role, it is necessary to subject the identification module to intensive testing which is proposed here.

There are a number of factors which may affect the generation of the model, and the ones tested for here are listed below:

- Sensor noise
- Presence of a mass in the gripper and uncertainties with that mass
- Selection of a model

Given the importance of this module it is organized differently compared to the other modules. For each independent variable the methodology and results are given, but the

discussion is saved for a common section which considers the test results holistically.

5.1.1 Evaluation Metric for the System Identification Module

The tests described in this section all share a common evaluation metric, and hence this is presented first. Every time the system identification module runs, it estimates 18 parameters (21 in the case of the selection of model testing), and each of these parameters may be compared to the true value. This allows the definition of a percentage error as outlined in Equation 5.1.

$$Error = \frac{|TrueVal - EstimatedVal|}{TrueVal} \times 100\% \tag{5.1}$$

Especially in the case of sensor noise, which will have unique data for every single test run, it is necessary to run each test several times to obtain an average (mean) error. At least 10 tests will be run for each test described here, and more if wildly different results are found between tests. Thus, in addition to the average errors, variances can be established which establish the extent to which the error may extend.

5.1.2 Sensor Noise

Test Methodology

Noise in sensors is unavoidable. Throughout the construction of the digital twin it has not been given much consideration, but considering that the digital twin model is built through a system identification process its effect must be evaluated. The addition of sensor noise will add further uncertainty in the position measurements in the manipulator, and this will likely cause larger errors in the determined parameters along with higher variance.

To simulate the presence of sensor noise, a maximum magnitude of sensor noise is specified and multiplied by a random number between 0 and 1 through MATLAB's random number function. This creates an offset, in that the sensor noise does not cause the signal to vary uniformly about the true signal, but sensor noise is not guaranteed to be uniformly

distributed about the measurement, and thus this is approach is valid. The model of the noise can thus be described in the Equation 5.2.

$$\theta_M = \theta_T + N \times RAND \tag{5.2}$$

Where:

- θ_M is the measured link position
- θ_T is the true position of the link
- N is the maximum magnitude of the noise encountered, in radians
- $RAND$ is the random number function in MATLAB

This leaves the magnitude of the noise as an independent variable. In this set of testing, four magnitudes are used.

- $0.001rad$
- $0.005rad$
- $0.010rad$
- $0.025rad$

The presence of noise is anticipated to have two main effects. First, the accuracy of the parameters estimated will degrade and a larger error will be found. Second, the variation about the average error will be larger with greater sensor noise.

Results

For the noise magnitudes of $0.001rad$, $0.005rad$, and $0.010rad$ twenty tests were run. However, due to the inability for the identification module to always produce a model with the given data, only 11 tests were conducted with a noise magnitude of $0.025rad$. The presentation of the results are divided into two figures due to the very large ranges in which values were determined. The first of these figures, Figure 5.1, is presented below.

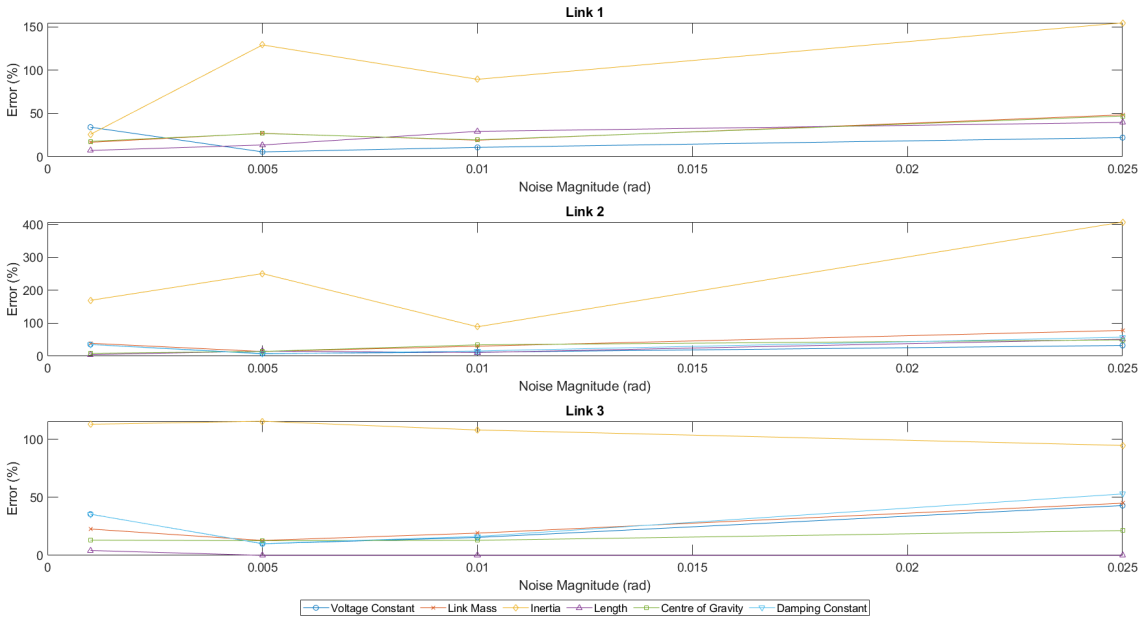

Figure 5.1: Error results for all the links under noise testing

Generally speaking, the anticipated effect of increasing error with increasing sensor noise is followed. Interestingly, some parameters determined during testing conducted with $0.010rad$ noise are more accurate compared to the parameters when determined lower amounts of noise. The expected trend continues, however, once the noise level is raised to $0.025rad$, and the average errors grow for all parameters on all links with the single exception of the moment of inertia of the third link.

The various variances in all tests are presented in Figure 5.2.

These results generally follow the same trend as before, with some exceptions. For example, in the second link the variance of the moment of inertia is greatest among the three lower levels of noise when using $0.001rad$ noise level. However, the variance is largest when $0.025rad$ noise is used, as anticipated.

While the anticipated trend is mostly observed, with some deviation, even the lowest of noise levels produces an appreciable variance in parameter estimation. The most prominent of these is the moment of inertia of the second link, and the moment of inertia of all links has the greatest variance compared to the other parameters. This demonstrated that the approach is very sensitive to noise, and thus engineers employing grey box modelling must be weary of the results they obtain whilst constructing a digital twin.

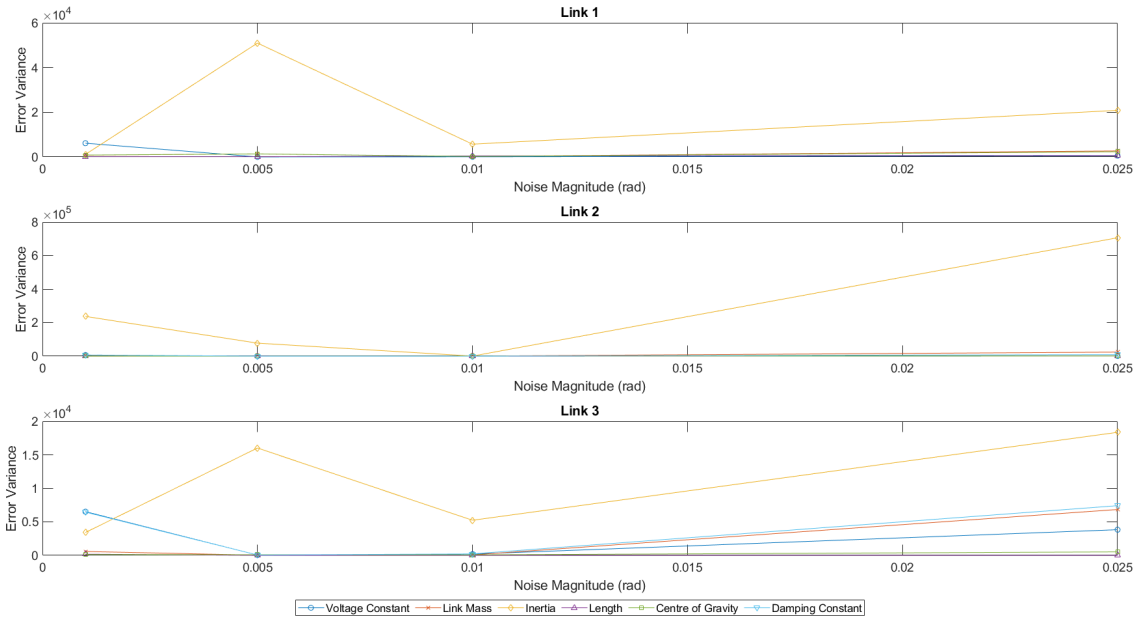

Figure 5.2: Variance results for all the links under noise testing

5.1.3 External Disturbance due to Mass in the Gripper

Test Methodology

The proposed digital twin architecture, and its model building capability, makes no restriction regarding the a mass in the gripper. The model which the system identification module fits to indeed has a term for this. Thus, it is possible for an external, unexpected, disturbance to be present when the module is used. Here, this disturbance is an uncertainty in the mass, and examining its effect is necessary in determining the limitations of the proposed approach.

Anyone engaging in a system identification procedure should have the foresight to account for a known disturbance like a mass in the gripper. Thus, the errors accommodated for here will be small, and likely attributed to measurement error. Three masses will be tested, compared to an expected mass of $200g$ of a box in the gripper. These are:

- $190g$, a $-10g$ error
- $200g$, a $0g$ error
- $210g$, a $+10g$ error

The presence of a disturbance from mass uncertainty, which is not accounted for in the system identification process, is expected to have two main effects. First, the error in the individual parameter estimation is expected to be greater with a mass error. Second, the variance about the average error is expected to be larger with a mass error.

Results

The results of mass testing are given in Figure 5.3, while the variance is given in Figure 5.4. For each mass in the gripper, ten tests were run.

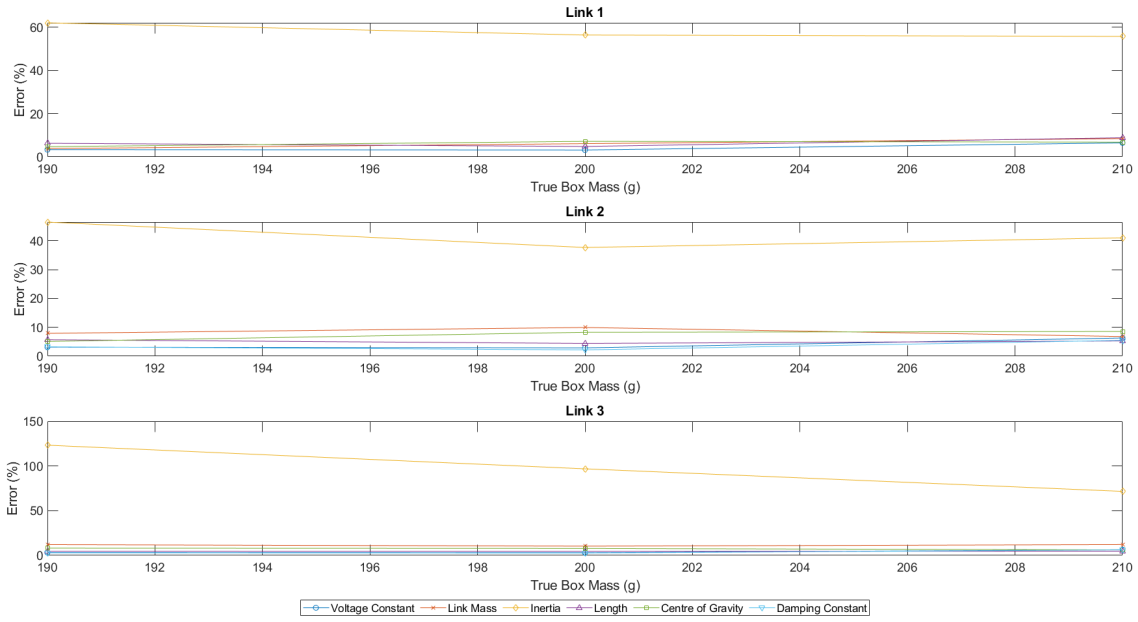

Figure 5.3: Error results for all the links under mass testing

The interesting result here is that the average errors and error variance are not necessarily smallest when a $0g$ mass error is being tested. For example, the moment of inertia error in the third link is smallest when a $+10g$ mass error in the gripper is present. Other than the inertias, however, the various estimates of all parameters are relatively accurate, with an average error of less than 11% for all object masses.

The variances show another interesting result. In particular for the first and third links, the error variances of the respective moments of inertia are orders of magnitude greater than the other parameters. This is mirrored in the second link as well, but to a lesser extent. The result demonstrates a great level of uncertainty in determining the moments of inertias.

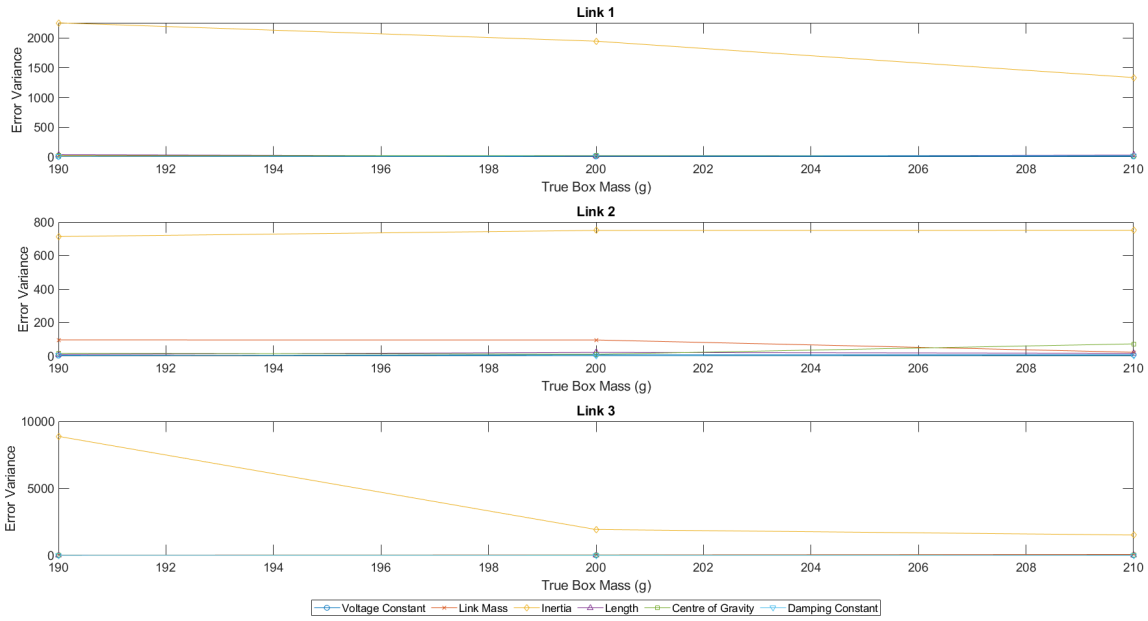

Figure 5.4: Variance results for all the links under mass testing

Overall, the results here suggest that mass error do not produce a significant effect on the success of the algorithm. It appears that even with no mass error, the algorithm still fails to estimate the parameters perfectly. The observation here, however, is not likely applicable across all digital twins, and should be taken cautiously. An external disturbance does not affect the efficacy of the algorithm here, but may affect its efficacy more severely in other applications.

5.1.4 Selection of Model

Test Methodology

Throughout the design of the system identification module, and the use of grey box modelling, it has been taken for granted that the dynamics of the system are known. The dynamics of the manipulator are entirely deterministic, and thus can be employed very easily for use in grey box identification. In this study even the friction model is known exactly, on account of the manipulator being an emulation rather than a physical device.

In reality, effects like friction, dead-zones, and hard non-linearities are not so easily known. Accurately modelling these requires estimation and curve fitting. Thus, part of building a

digital twin is determining these more complex effects, and using the system identification module to estimate them.

For these tests, two different models are considered and given to the system identification module. The first replaces the viscous friction model with coulombic friction, described in Equation 5.3.

$$\tau = M(\theta)\ddot{\theta} + B(\theta, \dot{\theta}) + G(\theta)\theta + \mu \cdot sign(\dot{\theta}) \tag{5.3}$$

Where μ are coulombic friction coefficients. Each of these coefficients are to set at $0.01N \cdot m$ as an initial estimate in testing.

The second model employed in this set of testing is similar to the first, but rather than explain all of the friction through coulombic friction, coulombic friction and viscous friction are allowed to explain the effect. This is described in Equation 5.4.

$$\tau = M(\theta)\ddot{\theta} + B(\theta, \dot{\theta}) + G(\theta)\theta + \mu \cdot sign(\dot{\theta}) + D\dot{\theta} \tag{5.4}$$

The use of the first model is anticipated to produce an inaccurate result, while the use of the second is anticipated to still produce an accurate result and reduce the Coulombic friction coefficients to zero.

Results

Coulombic Friction Model

No results are available for this model. Ultimately, the module could never fit this model to the collected data despite multiple attempts to do so. Earlier observations from developing the emulation likely explain this though. Without a viscous friction model, the emulation was never able to successfully simulate the robotic manipulator. This was due to the dynamics being too fast for the computer, which when calculating accelerations would estimate them to be many magnitudes greater than anticipated. While this may have been mitigated with a higher order ODE solver, or a smaller step size, it was not practical while attempting to maintain a pseudo-real-time emulation. The system identification module and MATLAB's internal algorithms likely encountered a similar issue.

Combined Coulombic and Viscous Friction

Two results of the combined friction model are given in Figure 5.5 in the form of a bar chart. These figures includes the average errors in estimating the 18 parameters, with accompanying error bars rather than variances, as well as the average calculated value of each of the three Coulombic friction coefficients over ten tests, again with accompanying error bars.

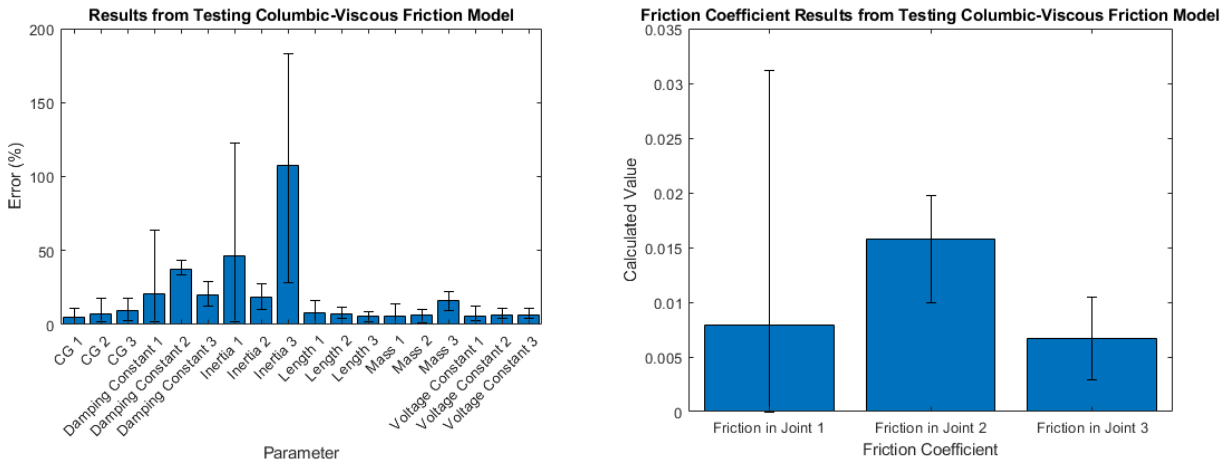

Figure 5.5: Results of the average error for each parameter with error bars when using the combined friction model, and the estimated Coulombic friction coefficients

These results show that the error on many of the physical parameters to be minimal. For example, the second voltage constant had an average error of 6.5%. With the exception of the moments of inertia and the damping, the average error on each parameter is below 16%. Until this point there has been a consistent trend in large errors being encountered when calculating the moments of inertia, and the discussion on this is left until later.

Of more interest here is the error in the damping parameters. These errors are significantly larger than in the tests with object mass uncertainty. Additionally, despite the actual emulation having no Coulombic friction, the system identification module consistently predicts that Coulombic friction is present in the system. In the second and third links Couloumbic friction is always predicted, while the error bars reveal that in at least one simulation the module successfully reduced the Coulombic friction coefficient to zero. This was unanticipated earlier.

The results from this testing reveal that an error in the selected model do not have much

effect on the ability for the module to determine physical parameters outside of the friction parameters. They do, however, have a significant effect on parameters which are not well known, in this case friction.

5.1.5 Discussion and Analysis of System Identification Results

Throughout the testing of this module, three observations can be made that are true for all testing scenarios. First, there is always some error in the physical parameters that are determined. Second, the errors found in the moments of inertia are significantly larger than other parameters. Finally, the variation in noise testing is significantly larger than tests conducted without noise. These observations are addressed here.

The first issue, of not perfectly determining the parameters of the physical system, may be explained in the form of either external disturbance, sensor error, or model selection in all of these tests. However, given that digital twins are supposed to become 1:1 representations of their physical counterparts according to some definitions, this may be problematic and bring into question the accuracy of the proposed approach. Thus, it should be established whether a perfect system, one without disturbance or sensor error, could determine the parameters perfectly.

To assist in this investigation, the results from one experiment are presented in Figure 5.6. This figure includes the parameter errors of the model, in a bar chart, as well as a simulated response comparison between the collected data and the determined model, the former of which consisted of 346 samples out of a possible 350. This model fits nearly perfectly to the collected data, which is revealed in the simulated response comparison and is, in this respect, an excellent candidate for the model which the digital twin uses. Furthermore, it had no sensor noise included or external disturbance present when it was constructed.

Despite the near perfect fit to the data, the parameters that were determined have a clearly large percentage error compared to the true parameters. This result supports a notion that a model does not need to determine all the parameters perfectly to effectively predict the behaviour of the system. To further support this notion, the system identification GUI

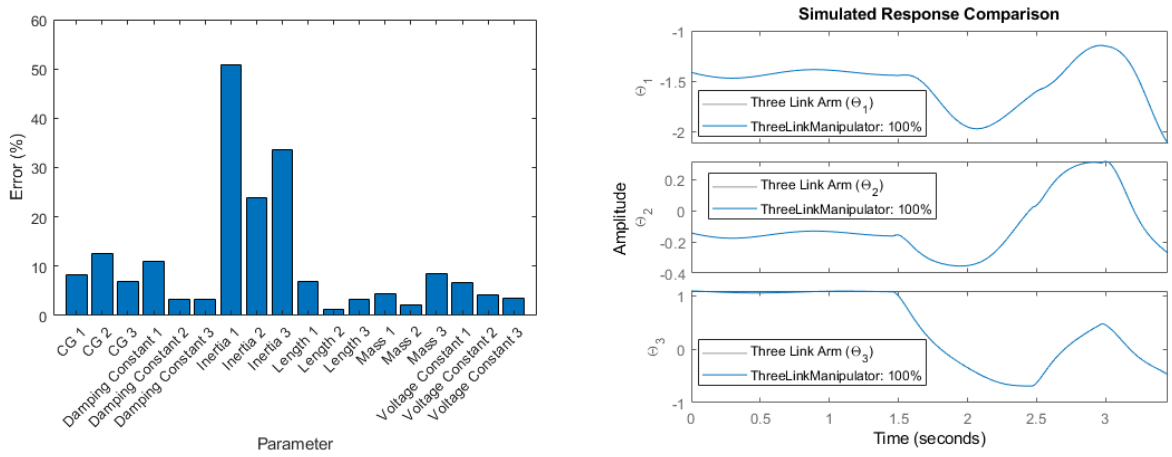

Figure 5.6: Results of the average error for each parameter for the model under examination as well as a simulated response comparison demonstrating the fit to the curve

for the third test with the combined Coulombic and viscous friction model, which contains response curves for each link, is given in Figure 5.7. Like the ideal model, this model again achieves an excellent curve fit to the data, yet with known errors in the friction parameters from earlier on.

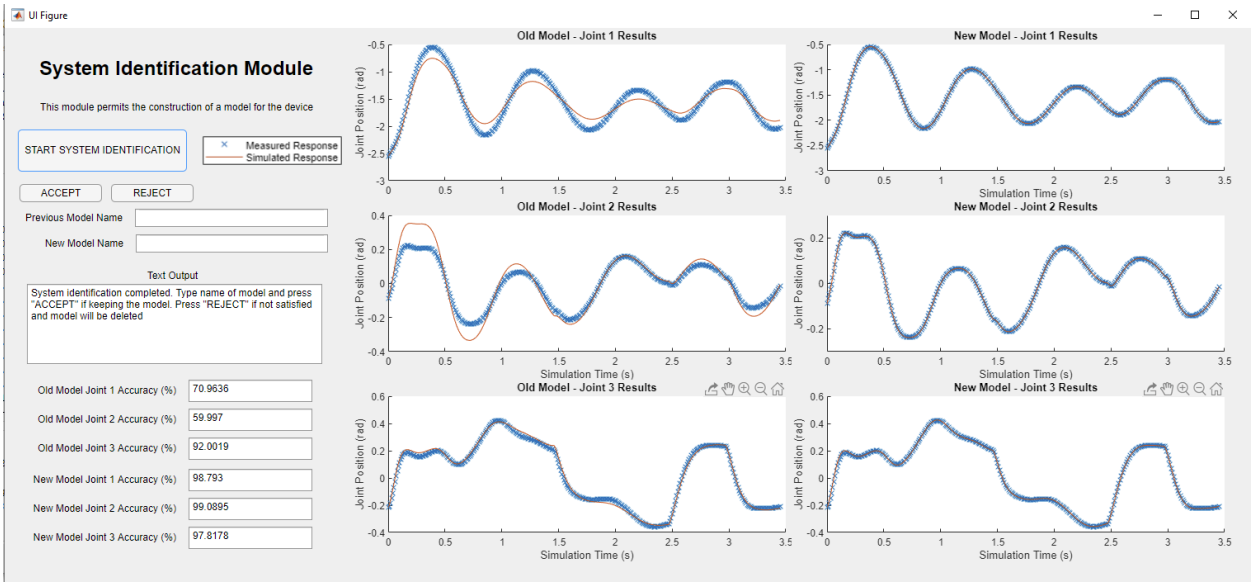

Figure 5.7: Results from the system identification module GUI for the third test with the combined Coulumbic and viscous friction model

What this establishes regarding the use of system identification for constructing the model is that the approach may generate a model which fits to the data very well, but is not necessarily correct.

This may also explain the deviations from the expected trends regarding increasing noise. Recall from earlier that larger errors were seen in some parameters at lower levels of noise. The result above shows that appreciable errors may occur in individual parameter estimation, but the determined model may still fit the collected data well. A few tests in which the determined parameters deviated significantly from the true parameters may cause a significant rise in the average error, which is observed in the high variances found at $0.005rad$ noise levels, and thus cause deviations from the expected trends.

Thus, if engineers are to apply this approach in constructing their digital twin, they should exercise caution in the determined parameters. Rather than applying system identification for each known parameter, as is done here, it would likely be wise to only apply it to determine parameters which cannot be easily measured. This may include friction, dead zones, or centres of mass. The more accurately parameters can be determined without system identification, the better chance the system identification module will have at determining the correct parameters.

The second observation from earlier was the rather large errors found in the estimates of the moments of inertia. This error still clearly exists even without external disturbances or sensor noise but, based on the previous discussion, appears to not effect the ability for the model to fit the collected data very well. This can likely be explained by the magnitude of the moment of inertia.

All links have a moment of inertia on the order of $10^{-4}kg \cdot m^2$. This is very small compared to the masses in the system, which are on the order of $10^{-1}kg$, and the positions of the centre of mass are on the order of $10^{-1}m$. Based on this, the likely reason such high errors are encountered in estimating the moments of inertia, while still producing an excellent fit to the data, is that it does not have much influence over the system dynamics relative to other parameters. This once again may be an issue for constructing a digital twin if the goal is to estimate the parameters precisely.

Finally, the massive range of errors encountered in noise testing must be addressed. This may be traced to the estimation of the initial state that the manipulator is in when the data collection begins. Recall that this manipulator only has three position sensors, but the model

makes use of three speeds and three positions in calculating the dynamics. Thus, it must use numerical differentiation in order to estimate the angular velocities of the links. The results from noise testing reveal this as an issue. Even at lower magnitudes of sensor noise, like $0.01rad$, this issue can occur. For example, the results of noise testing with $0.01rad$ noise during test 4 show a wildly different velocity in links one and two, which is presented in Figure 5.8.

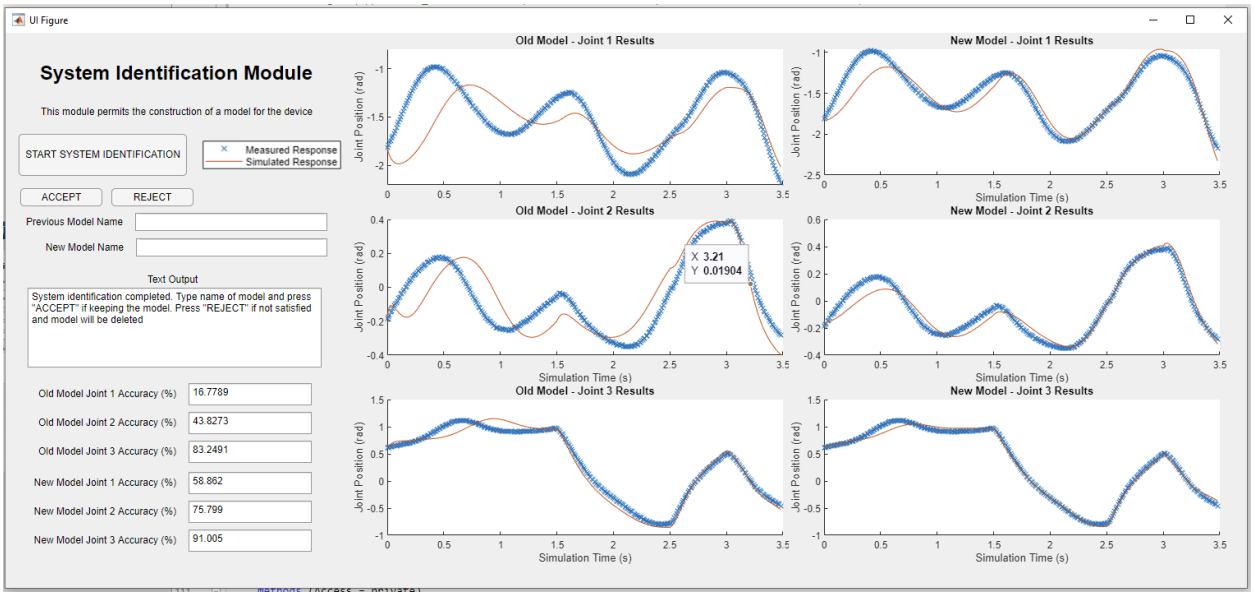

Figure 5.8: UI results from a noise test where the initial velocities have been improperly estimated

This leads to a possible challenge in the proposed approach, which may be addressed relatively easily. By minimizing the estimations required for the initial state, possibly by allowing the manipulator come to rest before sending the signal, this source of error may be minimized.

Regardless of whether or not the initial state challenge is mitigated, it does explain why in this testing such a wide range of estimations occurred. Without a reasonably accurate estimate of the system's initial state, the algorithm has little hope of accurately fitting a curve to the data. As such, it is an important lesson to those applying this approach to examine their curve fit.

A final observation should be made on the results of this testing. While it is clear that there are challenges to this approach of model building for a digital twin, it does not

mean that the determined model is ineffective. If the goal is to build an exact replica with every parameter as close as possible to the true value, then changes can be made to improve the estimation. However, if the goal is to produce a reasonable model which can accurately simulate the system, then the results here suggest that the approach may achieve that. Testing on the simulation and system monitoring module will determine if the models determined here are useful in this capacity.

5.2 System Simulation Module

Unlike the earlier system identification module which serves only one function and must be tested largely for sensitivity, the system simulation module is more of a design and purpose built tool which intends to make the digital twin useful. In this respect, it is not necessary to test its sensitivity to external factors, but rather to determine its efficacy in its purpose. These tasks are the ability to upload controller parameters into the controller and the digital twin's model into the controller, and simulate the system. This section is broken into two subsections which describe these tests, the results, and discussion of the results.

5.2.1 Simulation Accuracy Testing

Test Methodology

The ability to simulate its physical counterpart is central to the functionality of the digital twin. Thus, it is necessary to assess the accuracy of the simulation module. In this respect, such a test is not only an assessment of the simulation module, but a test of the accuracy of the digital twin model itself.

To conduct this test, the manipulator must be given a designated path in both the simulation module and the emulation itself to follow. The simulated response of the system may be compared to the response in the emulation, where the emulation controller uses the same digital twin model as the simulation module, to assess how accurate the simulation is. The more accurate the simulation, the more effective the tool may be in not only control

design, but in using the dynamics for whatever purpose the digital twin may serve.

This test requires the definition of several aspects in order to be completed. First, the defined path. Both the controller and the simulator make use of a path defined by a cubic function, so thus the same start and endpoint will suffice to define the path here. All tests will thus follow a path from the box pickup location, and end at the red box drop off. Second, a model must be selected. For the tests completed on the system simulation module, the model produced by the noise testing with $0.01rad$ noise, test number six, was employed. Figure 5.9 shows the resulting curve fit and the various parameter errors that occurred when the estimation was complete.While this model is not perfect, it fits relatively well in the system identification GUI, with all curve fits achieving a fit greater than 90%. Furthermore, there are some sizeable errors in the parameter estimation, greater than the ideal model previously presented. Testing with this model is likely to reflect a more real world scenario, where such errors would occur. Thus, it is a suitable model to test the simulation capability of the module with.

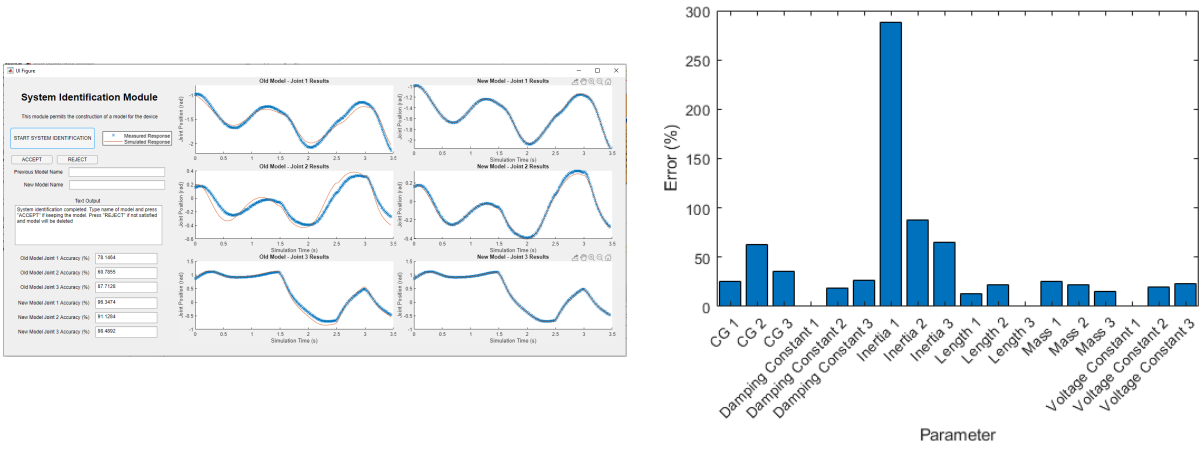

Figure 5.9: Results from the GUI and the error in parameter estimation for the $0.01rad$ noise testing, test number six

Finally, it should be considered that the controller has an influence on the dynamics of the response. Thus, the pathway should be manipulated such that the controller cannot push the manipulator to perfectly follow it. In this case, that can be achieved by manipulating the target time of the computed torque controller, which effectively becomes an independent variable. As much of the digital twin's simulation functionality focusses on the arm in use,

testing with a controller in play is necessary in evaluating its efficacy.

Thus, this set of tests will consist of moving the manipulator between the box pickup location and the red box drop off point. The following control target times are applied:

- 1s
- 2s
- 4s
- 6s
- 8s

Next, an evaluation metric must be defined. MATLAB provides internal function for evaluating the goodness of the fit, and it employs the normalized root mean square error. The same error metric is used in the system identification module, as defined in Equation 5.5.

$$fit(i) = \frac{||x_{ref}(i) - x(i)||}{||x_{ref}(i) - mean(x_{ref}(i))||} \tag{5.5}$$

Finally, there must be a way of defining the start and stop points in which the comparison takes place. As the controller in the emulation will have an imperfect model, while the simulation module will believe that is has a perfect model, the two system responses will not settle at the target at the same time.

To resolve this challenge two metrics are used in combination. The first of these is the 2% settling time. This is a common metric in evaluating control system performance, and as such is employed here. Second, if all three links reach an absolute speed below $0.01\frac{rad}{s}$, the arm will be considered as nearly motionless. These two, in combination, will be considered to be the point at which the manipulator is settled and the comparison can be ended. Whichever took longer, the module or the emulation, will be used as the period for comparison.

It is anticipated that the shorter the control time, and the correspondingly larger control effort, will lead to a worse curve fit. With more aggressive manoeuvres, the dynamics of

the arm will play a larger role and the controller will be unable to compensate for the error between the believed model and actual manipulator as well.

Results and Discussion

The resulting curve fit values of the various control times are displayed in Figure 5.10. Note that a value of 1 implies a perfect fit. All the tests have a curve fit, on all links, greater than 0.9. Furthermore, as anticipated, with an increased control target time the curve fit improves. This is because the feedback terms of the computed torque controller begin to play a larger role in the simulation, and have more time to react and fit the response of the manipulator to the desired path. As such, the curve fit improves.

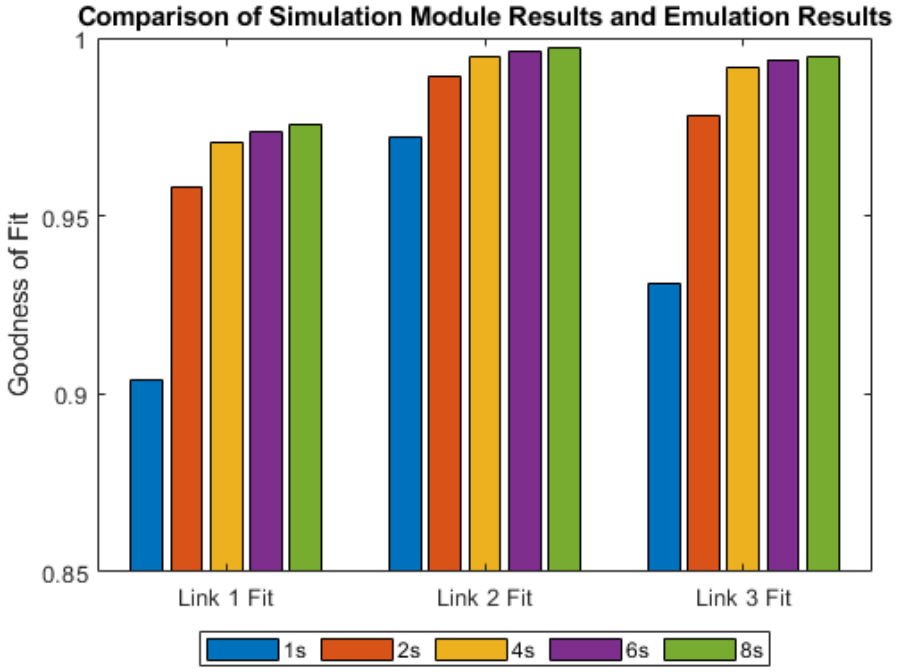

Figure 5.10: Curve fits for all of the links at the various control target times

Some examination must be dedicated to the worst of the curve fits, which in the case occurred on the first link with a one second control target time. This is provided in Figure 5.11. Interestingly, the emulation appears to have followed the designated path better than the simulation module did, even though the simulation module's controller should have a model which is perfect relative to the manipulator. This may be explained by the errors in

the damping constants, which as noted previously are not exact. The model is very sensitive to the friction model, which was observed in the system identification module tests as that module could not identify a model without the presence of viscous friction. Deviations in the applied damping parameters can have serious effects in the overall dynamics, and in an aggressive manoeuvre is likely responsible for the observed behaviour.

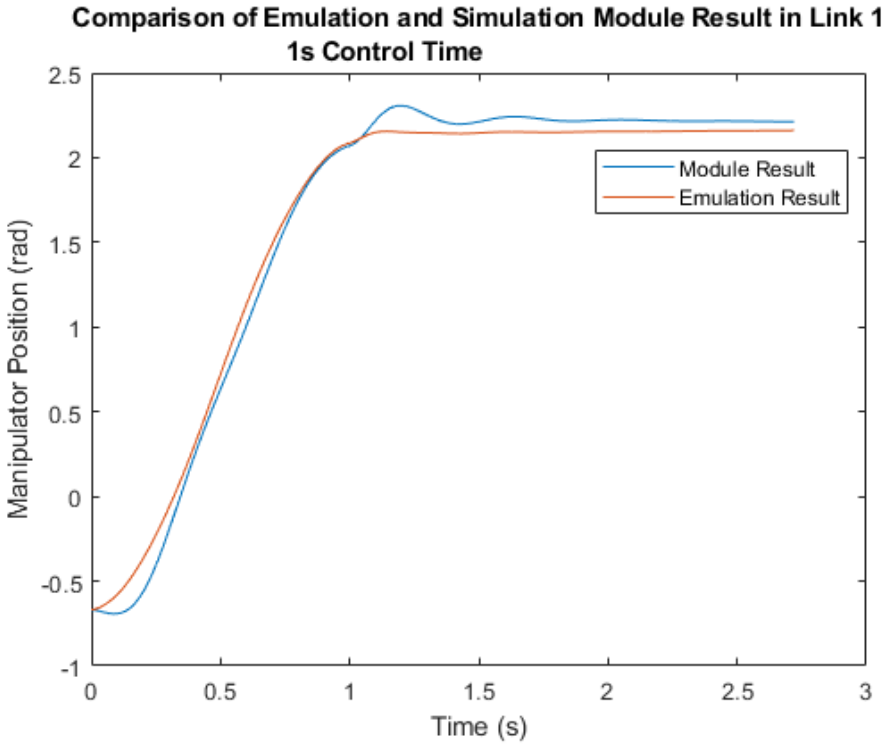

Figure 5.11: A detailed view of the result for the first link with a one second control target time.

The question of if this simulated response is accurate enough for a digital twin is a subjective one, and it is difficult to draw an exact conclusion from it. For example, if an engineer were only interested in the control response, this may be sufficiently accurate. Or, the engineer working on the system may determine that the control target time is too low based on this result at the simulation level, and may simply choose a larger control target time, which the previous results demonstrated as being more accurate anyhow. Lastly, the engineer may determine the model is not accurate enough as this dynamic response is used in a high performance system where the accuracy is paramount to predicting failure and stresses in the component.

With accuracies of over 95% in all target control times greater than 1s, though, it cannot be said that the model used here is not at least useful in those applications. Ultimately, the result with a one second control target time only further reinforces the limitations of entirely relying on the system identification module to blindly estimate all the parameters. However, the curve fit results at the higher control times suggests that the system can be accurately simulated with the proposed model and approach.

5.2.2 Uploading Control and Model Parameters

Test Methodology

The system simulation module focuses almost exclusively on control design. Its function could be expanded to simulate new trajectories and a digital twin of an industrial manipulator could conceivably be given this task. However, as the manipulator in this work is as simple as it is, this particular module is focussed only on control design.

Given this nature of the module, it would be useful on an actual production line to be able to upload the various control parameters, such as the controller gains, control target time, or the model parameters used by the computed torque controller, at the discretion of an operator or AI. Thus, it should be demonstrated that the module's function block component can complete this task and upload the parameters into the manipulator controller function block. This test intends to demonstrate that it is possible to do this.

This test will take place over a 180 second interval. An inferior set of controller gains, all set to zero, with the initial inferior digital twin model in the controller will be used at the start of the simulation. At 60 seconds, a better set of controller gains given in Table 5.1, and at 120 seconds, the ideal digital twin model will be uploaded. The test will be considered successful if the simulation displays a clear improvement in performance after uploading the previously determined controller gains, and a change in performance after uploading the model, without the emulation producing an error.

Control Gain	Link 1	Link 2	Link 3
K_p	200	500	300
K_i	75	200	150
K_d	2	2	2

Table 5.1: The controller gains to be uploaded in the parameter upload test

Results and Discussion

The resulting simulation in which the parameters were uploaded is given in Figure 5.12. The points at which the controller parameters and the model parameters were uploaded are labelled with a red and green vertical dashed line, respectively. Like the previous set of simulation module evaluation tests, the model from the $0.01rad$ noise test number six was employed.

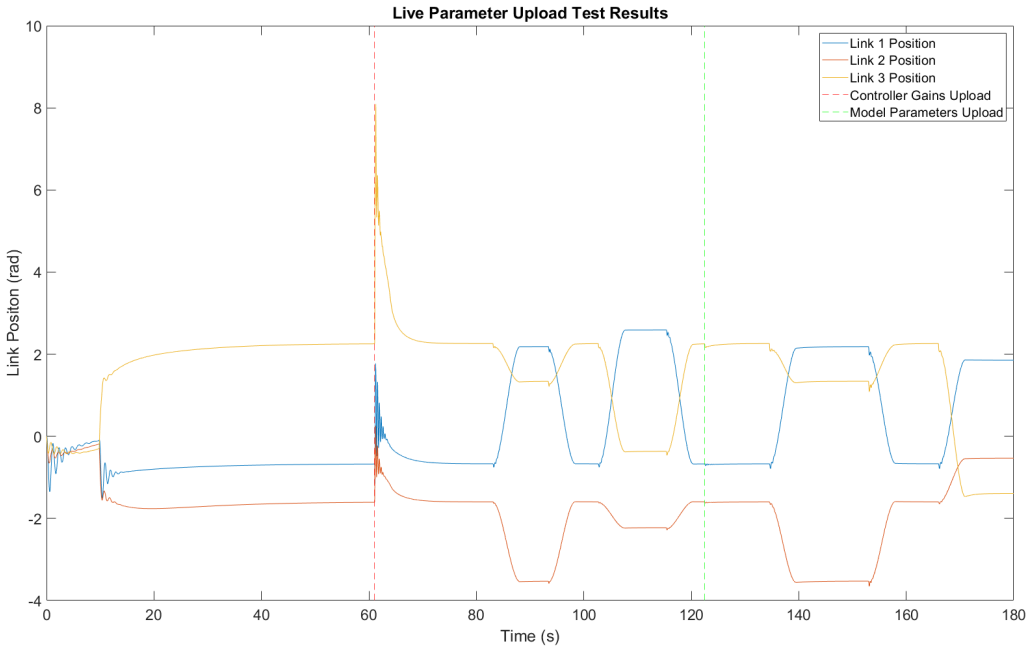

Figure 5.12: Results from the parameter upload simulation test

The results here demonstrate, clearly, that the proposed digital twin architecture is capable of live updates into the physical system. At both 60 seconds and 122 seconds, the parameters from the model and new controller are successfully uploaded. In the case of the controller parameters, after a clear disturbance, the controller becomes far more capable of completing its target motion. Before, with the very poor gains, the controller was unable to

reach the initial pickup location within 60 seconds. The model parameter upload test shows a similar result with a successful upload and minor disturbance, but without as dramatic of a change in performance compared to the controller gains.

This result supports the proposed architecture for digital twins in IEC61499 based distributed control systems. It demonstrates that the function block architecture is able to interact with the physical device through the bidirectional communication pathway, without stopping operations. This is a crucial functionality for digital twins reaching their full potential.

5.3 System Monitoring Module

The primary purpose of the system monitoring module was to detect if a fault occurred. Until now, the validity of the system monitoring module's failure detection criteria has not been given much discussion. Generally speaking, such a discussion falls under the umbrella of Failure Mode and Effect Analysis (FMEA), which is generally beyond the scope of this thesis. However, it is still important to devise some kind of test to demonstrate that the proposed architecture could be employed in such an application.

As such, it is necessary to briefly examine some of the surrounding literature for failure mode detection. The use of sensors to detect failure modes is already well established. As an example, Kuo, Chuang, and Liang employed common sensors to analyse vibrations in bearings of a shaft bearing system [39]. Their work demonstrated the use of sensors and knowledge of the frequencies which indicated failure could be applied to determine which bearing had failed and which failure mode occurred. This work indicates that sensors can be employed in a sophisticated manner for failure mode detection, as well as outlining that the manner in which sensors are used is highly specific to the particular problem at hand.

This work, however, does not support the use of a simulation based failure detection system. Sini, Passarino, Bonicelli, and Violante, in contrast to the previous work, made use of both sensors and a simulation to detect failure mode [40]. Their work involved applying a suite of sensors on a mobile robot system, of which various combinations of sensor readings

could indicate failure, in combination with a simulated nominal performance. If the actual system performance deviated greatly from the expected behaviour, in combination with the sensor readings indicating abnormalities, then failure and failure mode could be detected. Their approach was effective in detecting a majority of the possible failure modes. This work, which is more applicable to a robotic system like the one of this thesis, demonstrates that applying simulation to compare nominal and abnormal behaviour can work.

Thus, there are two sets of testing that must be completed. First, there needs to be a set of testing to determine how much data can be transmitted over through the proposed architecture. Second, it must be determined if it is plausible for the simulation capacities of this digital twin architecture to detect failure.

It is important to note that these tests are designed to determine if it is possible for the architecture to detect a failure, which was listed as a key functionality for a digital twin of a system controlled through a DCS. The method in which it detects failure is, admittedly, very basic and not state of the art. However, the tests will demonstrate if the architecture can be applied in such a capacity.

5.3.1 Sampling and Simulation Time Step Testing

Test Methodology

This test aims to evaluate the suitability of the architecture for failure mode detection, as it will determine how much useful data can be extracted. Given the method in which the system monitoring module operates, there are two independent variables. These are the sample time, and the simulation time step size. While manipulating sample time, 10ms, 100ms, 500ms, and 1s sample times will be employed, and a simulation time step size of 5ms put in place for all tests. For manipulating the simulation time step size, 1ms, 2ms, 2.5ms, 5ms, and 10ms step sizes will be employed, with a 1s sample time in place. The results of this testing will determine the limits at which the module can be effective for failure mode detection. For this testing, an ideal digital twin model with no noise is employed, and the controller is given the true parameters of the robotic manipulator. This is done for simplicity,

as no actual evaluation of digital twin model takes place here.

Results and Discussion

For the discussions here, only four results are presented. The first two are given in Figure 5.13, which are the results from Link 1 when the sample time was set to 1 second, and the simulation time step sizes of 5ms and 2ms. These figures show the normalized error as well as the simulated path and measured positions. The former of these are not relevant for this particular discussion, but do demonstrate part of the importance in having a failure criteria setup to not determine fault after a single data point.

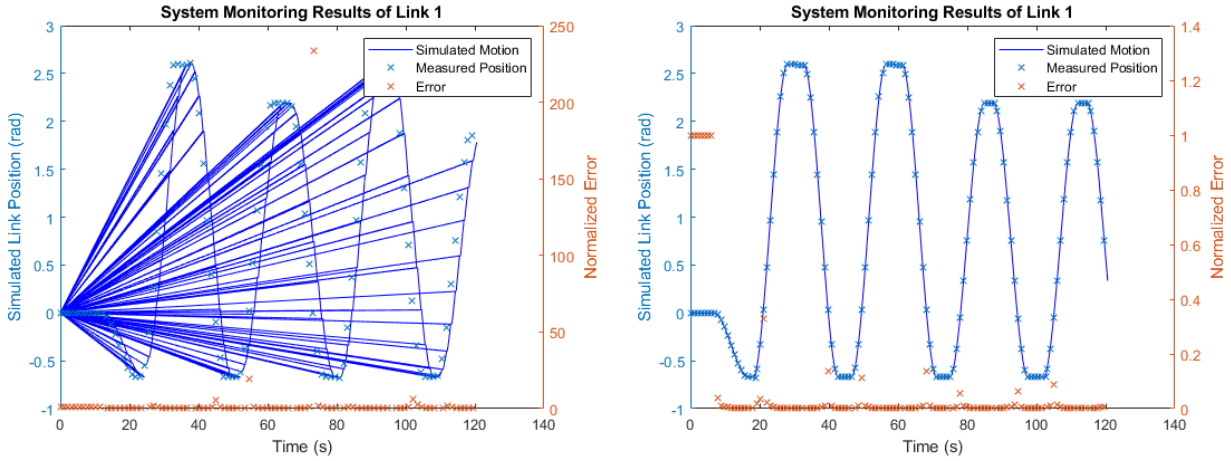

Figure 5.13: Results from testing the system monitoring module with a sampling time of 1s, and a simulation time step size of 2ms (left) and 5ms (right)

These results show a timing limitation in the simulation time step size. While at 5ms the corresponding simulation fits well with the sampled data points, a time step size of 2ms shows a jagged line that is delayed from the true signal and returns to the origin. The reason that this has occurred is because the amount of time required for the algorithm to simulate the arms motion with this time step size is longer than the actual time it takes for the emulation to move between intervals in real time. Thus, data points are missed, and the system monitoring module is unable to keep up with the system. This demonstrates the need for the simulation of the system monitoring module to be able to move sufficiently quickly to keep up with the physical device.

Given this information, the effect of the sampling time can now be examined, with a 5ms

simulation time step size employed since in can accurately simulate the manipulator without missing samples from the manipulator. Like before, the most relevant results are shown in Figure 5.14 with a sample time of 100ms and 10ms.

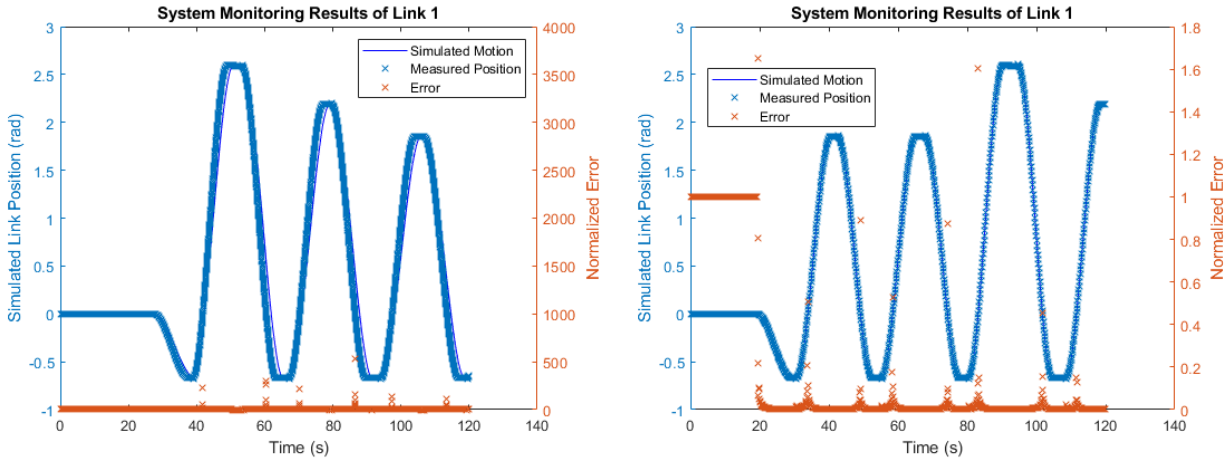

Figure 5.14: Results from testing the system monitoring module with a sampling time of 10ms (left) and 100ms (right), and a simulation time step size of 5ms

A similar result can be seen here as in the previous figure. A delay once again starts to happen when the sampling time is too low. This is because the time required to obtain the measurement is larger than the sampling time, leading to a disagreement between the simulation result in the sampled data. Thus, there is a limitation again in the system.

Overall, these results here show two key results, and it is noted that these results are specific to this setup and the computer on which the module is run. First, the tests demonstrate that those designing a digital twin using this approach need to consider the effect of these parameters. Second, on this particular setup, the system monitoring module can be counted on to be effective with a time step size of 5ms and sample time of 100ms.

5.3.2 Failure Detection Test

Test Methodology

Now, it must be determined if it is plausible for the simulation capacities of this digital twin architecture to detect failure. Currently, the system monitoring module compares the simulated value and the measured value at the designated comparison point. Two errors

are calculated for the comparison. These relative error, as defined in Equation 4.19, and an absolute error, as defined in Equation 4.20.

As noted earlier, if each of these errors exceeds a certain threshold, then a deviation can be detected. However, given that the digital twin model is not anticipated to be perfect a single deviation is not noteworthy, as it can be associated with sensor noise or model error. Thus, the algorithm should notify the operator after a number of deviations have taken place. To complete this discussion on failure detection, the thresholds are defined as follows:

$$Rel.Error_{max} = 0.05$$

$$Abs.Error_{max} = 0.1rad$$

$$N_{deviation_{max}} = 10$$

Now the test may be described. The system monitoring module will be activated along with the emulation and it will monitor a nominal arm for the first 45 seconds of the test. At 45 seconds, the torque voltage constant of the second link will be reduced to 10% of its original value. This should cause a deviation in the path, and the controller, with saturation limits of $\pm 8V$, will be unable to compensate for. The deviation will then be monitored for and it will be seen if the module successfully detected the error.

As a final note on this test, a model, sample time, and simulation time step size must be defined. For this test, a simulation time step size of $5ms$ is employed along with a sample time of $0.5s$. These values comply with the limits established for this module. The model can now be chosen. Once again, a model from the noise testing will be employed as it is relevant to chose a more realistic model here given the functional nature of the test. However, as the aim is not to determine the limits of model accuracy in regards to failure detection, a more accurate model compared to that employed in the system simulation testing will be employed. In this case, it's the digital twin model resulting from noise testing with $0.001rad$ noise, test number one. The GUI results for this test are presented in Figure 5.15. This shows a very good curve fit, making it a suitable choice for this test.

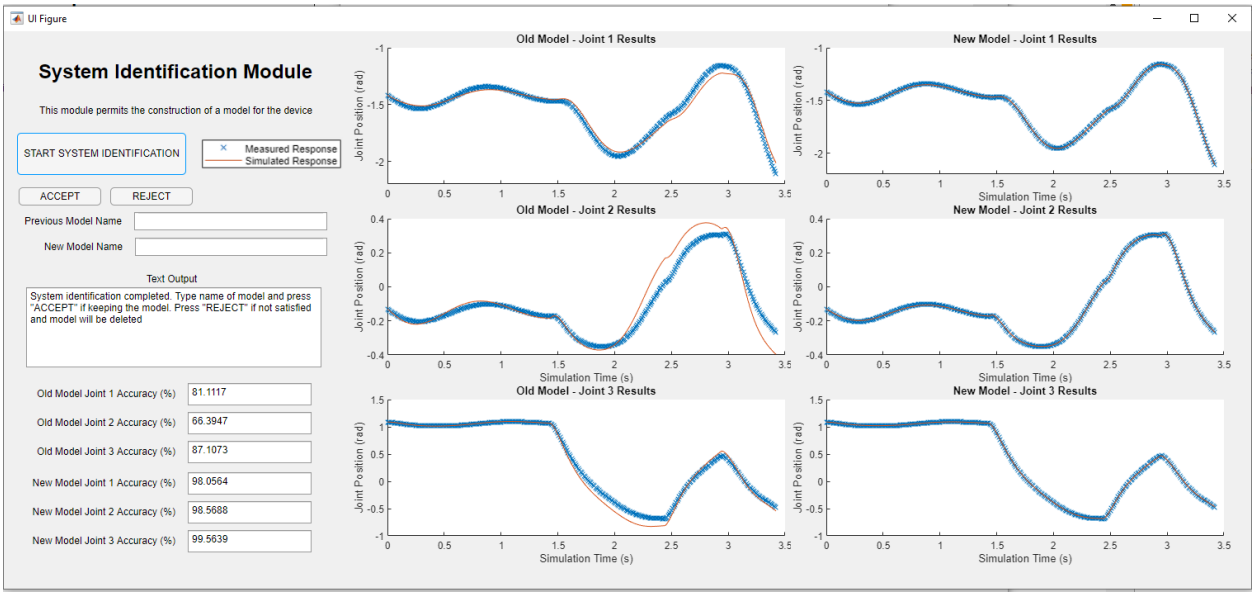

Figure 5.15: GUI result from noise testing with 0.001rad, test number one, which will be employed for the system monitoring test

Results and Discussion

Two results are presented for the test. First, the formal result of the simulation, with the failure onset and detection points marked using dashed vertical lines, is provided in Figure 5.16. Second, the resulting GUI from the system monitoring module is provided in Figure 5.17. Of important note here is that in the GUI result both notes that a deviation has occurred in the message centre, and the collected data points in red clearly deviate from the blue line which notes the anticipated path of the model.

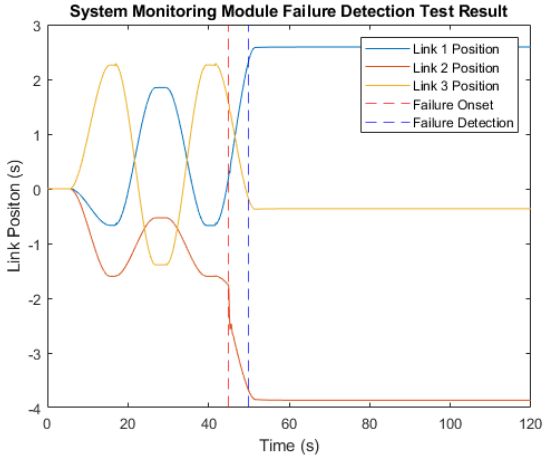

Figure 5.16: The resulting simulation in which the catastrophic failure occurs

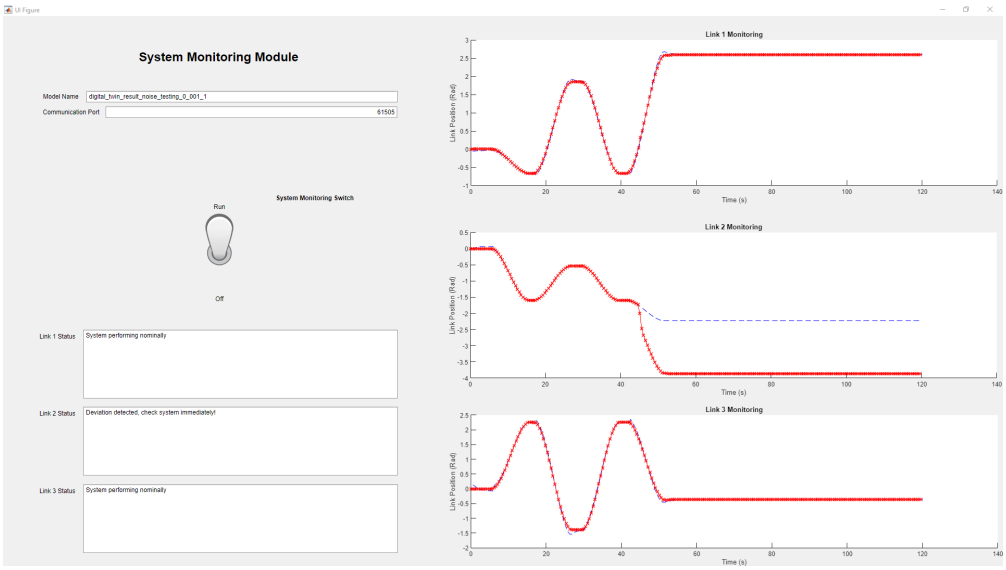

Figure 5.17: The result of the test viewed in the system monitoring module GUI. Note on the left side that the GUI has detected that failure occurred in the second link

The results here show that the proposed architecture can be applied in a failure detection capacity, one of the key functionalities for digital twins employed in distributed control systems. The failure detection method is very rudimentary, but the use of a simulation based approach for failure detection is indeed usable with the proposed IEC61499 architecture.

Chapter 6

Conclusion

At the beginning of this thesis, a set of contributions were presented that would be made over the course of the document. These are repeated here.

- A definition for a digital twin
- An itemized list of the key functions a digital twin should be capable of for industrial production
- A general architecture which may be replicated as necessary for constructing a digital twin of a physical asset managed through a distributed control system
- A proposed approach for constructing digital twins through grey box modelling, with an accompanying evaluation of the approach, the novelty laying in the application of existing grey box modelling algorithms towards constructing digital twins
- An evaluation of the International Electrotechincal Commission 61499 standard in the task of constructing a digital twin and interfacing it with a DCS

In the third chapter, specifically Section 3.2, a definition for a digital twin was provided. In this work, a digital twin is considered to fundamentally be a software centred on a high fidelity model, built from physical models and sensor data, of a physical assent, with an accompanying bi-directional communication pathway. It is not the model itself that achieves

useful tasks but rather its application through the software built around it, and hence why a digital twin is defined as such.

Digital twin software requires a certain level of functionality. The literature makes it clear that this functionality is dependent on the application, but in Section 3.2 a list of essential functions for a digital twin involved with industrial production is provided. This narrows down the list of functions that designers should aim to develop should they develop a digital twin.

Throughout Section 4.2 an architecture for a digital twin of a device controlled through the IEC61499 standard is presented. This includes an element constructed using IEC61499 function blocks working in tandem with a higher level program language. In this manner, the architecture enables the creation of a digital twin fitting with the definition provided and achieves the functions listed.

In Section 4.2.2, the use of grey box modelling for constructing digital twins of devices controlled through a DCS is proposed, as it fulfils the core concept of a digital twin. Throughout Chapter 5 the efficacy of this approach is evaluated by comparing the models generated by the grey box approach with the true parameters, by comparing dynamic responses of grey box models with the actual system, and by examining the grey box model's use in system monitoring tasks. The results of these tests reveal nuances with the approach, but establish that the approach can be effective in the functions of a digital twin.

However, the last contribution is not yet completed. This is left for the next section as it sets up the future work.

6.1 Evaluating IEC61499 in Constructing and Interfacing With Digital Twins

At the beginning of this thesis it was postulated that digital twins would be a key element in constructing a DISCS, that is transforming a DCS into a intelligent system. Early on, it was also proposed a manner in which that could be achieved, roughly speaking, using a

collection of digital twins to create an overall twin for an entire production system. Having created a digital twin architecture here that is capable of the essential tasks outlined earlier and possessing additional knowledge relative to the start of the thesis, and evaluation of the use of the the IEC61499 standard for constructing and interfacing with digital twins can be conducted.

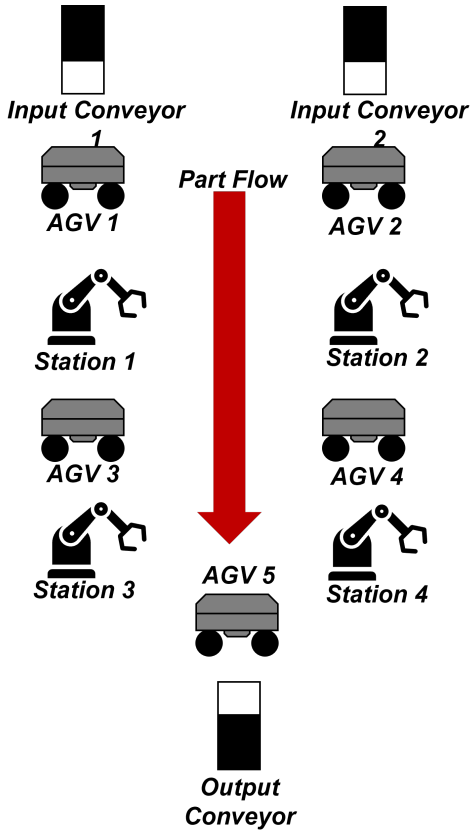

Figure 6.1: The fictitious flexible production system used for this example

There are a few obvious positives that can be pointed out immediately. The event based nature of IEC61499 function blocks lends flexibility in how control algorithms are enacted, and there is the ability to send data between function blocks in line with the events rather than every time step as a more traditional PLC would do. This is leveraged throughout the thesis to ensure the digital twin can reach its full potential. Additionally, it is seen that the complex communication protocols can be enacted into function blocks like the TCP/IP protocol used here, and this can be leveraged in using higher level programming languages like in this form. Furthermore, because of the event based nature of function blocks it is simple to employ these protocols.

The points made here, however, are not just for the sake of repeating them. They are presented because they are important in achieving a DISCS. Consider a fictitious flexible production line consisting of several autonomous ground vehicles (AGVs) and several work stations where robotic manipulators are located and manufacturing operations take place. This is similar to the production system described by Jeon and Schuesslbauer [17]. The production system is shown in Figure 6.1.

The reason this hypothetical production line is presented is that it acts as a good example to explain how more complex digital twins would interface in a production system and achieve a DISCS. Take a scenario such as fault recovery, say that an arm fails at one of the assembly stations. Recovering from this fault would likely involved reconfiguring one of the other arm stations to conduct two assembly operations in parallel, and have the two AGVs ferrying parts to the newly reconfigured station and other AGVs go to this new station and retrieve these parts.

Enacting that recovery strategy requires the overall control system in place for the AGVs to be capable of reconfiguring the destinations to which they travel to and from, and the manipulator needs to be capable of navigating to new positions and completing a new set of operations. This means having a flexible manner in which data can be transferred and new control algorithms can be implemented is absolutely necessary.

This example highlights a strength that IEC61499 possesses over other traditional automation strategies. Possessing multiple algorithms, reconfiguring destinations, storing data, and enacting specific algorithms based on the requirements at hand are characteristic of objects from object oriented programming. Recall the discussion on function blocks in IEC61499 that drew many parallels between basic function blocks and objects, and was able to establish that function blocks are effectively objects.

Now, consider this in combination with the earlier points on the event based nature allowing for flexibility in how the control algorithms are enacted and what data is transmitted and when. This means that there is a mechanism within the standard to precisely control how function blocks go about conducting control processes. This means that a tremendous amount of functionality can be built into a single function block, and the mechanism for

enacting specific functionality exists through the events of the standard.

Functions necessary for fault recovery, optimization, and dynamic reallocation can all be programmed into the control system using IEC61499. In the scenario described here, this would include functions like new assembly operations in the manipulator and the ability to dynamically reconfigure destinations for the AGVs.

Digital twins are at the centre of both enabling and determining this functionality. Humans or AI would use digital twins of the production system and its various components to for various purposes like to simulate faults and determine possible responses, or optimize the production system and propose manners in which optimal configurations could be implemented. Given the tremendous flexibility that function blocks possess, these various configurations can then be programmed and embedded in the function block or, possibly, be dynamically enacted with an existing function block. Finally, the function block element of the digital twin that is proposed in this thesis would be responsible for telling the function blocks of the control system if and when to implement the strategies previously determined through the digital twin. This is all achievable with function blocks and their various events.

This illuminates many of the strengths that IEC61499 has in interfacing with digital twins, and in implementing part of them using function block component. However, to the fault of IEC61499, the standard is incapable of completing the complex operations one would hope to achieve using a digital twin due to the more simplistic nature of function blocks. This means that the digital twin cannot be implemented exclusively as a function block. The standard does address this though, in the form of the aforementioned capability to implement complex communication protocols to communicate with higher level programming languages capable of constructing the software necessary to achieve these more complex functions. Thus, the standard still provides a method for achieving all functions of the digital twin.

To conclude this evaluation, the key points are summarized. The object based nature of function blocks enables the creation of many control strategies and collapsing them into a single point which interfaces with a device in a production facility. This means that complex strategies necessary for advanced tasks like fault recovery may be incorporated into the control system. The event based nature of IEC61499 means that there is a ready method

to enact those strategies as needed through a digital twin's function block architecture. Finally, the ability for IEC61499 function blocks to interface with higher level programming languages means that the full functionality of a digital twin can be realized and applied in a DCS. These features make IEC61499 a very effective standard in creating and interfacing with digital twins. This completes the final contribution of this thesis.

6.2 Future Work

With the previous discussion complete future work can now be discussed. In line with the work in this thesis, this discussion will focus on working towards establishing a DISCS. This requires several advancements not yet completed.

First, the creation of zone twins and overall twins from Chapter 3 are needed. These are needed to complete more sophisticated simulation tasks, such as fault recovery and overall optimization. They will also open the gateway to incorporating algorithms needed for fault recovery and optimization into function blocks. These should be the next step after the work established here, as it will establish the groundwork needed to allow an AI unit to control the production line.

Next, the interface between digital twins and AI must be constructed. This interface is necessary for the self management aspects of DISCS, and permits the creation of a self managing unit. As described previously, the AI will be responsible for determining responses to conditions and implementing them on a production line through function blocks.

The work presented in this thesis is the first step in working towards using digital twins to establish a DISCS. Future work should build on this work to achieve this goal.

Bibliography

[1] M. Liu, S. Fang, H. Dong, and C. Xu, "Review of digital twin about concepts, technologies, and industrial applications," *Journal of Manufacturing Systems*, vol. 58, pp. 346–361, 1 2020.

[2] B. Schleich, N. Anwer, L. Mathieu, and S. Wartzack, "Shaping the digital twin for design and production engineering," *CIRP Annals - Manufacturing Technology*, vol. 66, pp. 141–144, 2017.

[3] S. Datta, "Multiple forms of digital transformation are imminent," *Journal of Innovation Management*, vol. 5, pp. 14–33, 2017. [Online]. Available: https://dspace.mit.edu/handle/1721.1/104429

[4] Q. Qi and F. Tao, "Digital twin and big data towards smart manufacturing and industry 4.0: 360 degree comparison," *IEEE Access*, vol. 6, pp. 3585–3593, 1 2018.

[5] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital twin in manufacturing: A categorical literature review and classification," in *IFAC-PapersOnLine*, vol. 51. Elsevier B.V., 1 2018, pp. 1016–1022.

[6] A. M. Madni, C. C. Madni, and S. D. Lucero, "Leveraging digital twin technology in model-based systems engineering," *Systems*, vol. 7, 3 2019.

[7] R. Rosen, G. V. Wichert, G. Lo, and K. D. Bettenhausen, "About the importance of autonomy and digital twins for the future of manufacturing," in *IFAC-PapersOnLine*, vol. 28, 5 2015, pp. 567–572.

[8] F. Tao and M. Zhang, "Digital twin shop-floor: A new shop-floor paradigm towards smart manufacturing," *IEEE Access*, vol. 5, pp. 20 418–20 427, 9 2017.

[9] R. He, G. Chen, C. Dong, S. Sun, and X. Shen, "Data-driven digital twin technology for optimized control in process systems," *ISA Transactions*, vol. 95, pp. 221–234, 12 2019.

[10] Q. Liu, J. Leng, D. Yan, D. Zhang, L. Wei, A. Yu, R. Zhao, H. Zhang, and X. Chen, "Digital twin-based designing of the configuration, motion, control, and optimization model of a flow-type smart manufacturing system," *Journal of Manufacturing Systems*, vol. 58, pp. 52–64, 1 2020.

[11] N. Meier, R. Müller-Polyzou, L. Brach, and A. Georgiadis, "Digital twin support for laser-based assembly assistance," in *Procedia CIRP*, vol. 99. Elsevier B.V., 2021, pp. 460–465.

[12] Z. Liu, W. Chen, C. Zhang, C. Yang, and Q. Cheng, "Intelligent scheduling of a feature-process-machine tool supernetwork based on digital twin workshop," in *Journal of Manufacturing Systems*, vol. 58. Elsevier B.V., 1 2020, pp. 157–167.

[13] K. Zhang, T. Qu, D. Zhou, H. Jiang, Y. Lin, P. Li, H. Guo, Y. Liu, C. Li, and G. Q. Huang, "Digital twin-based opti-state control method for a synchronized production operation system," *Robotics and Computer-Integrated Manufacturing*, vol. 63, 6 2020.

[14] J. Vachalek, L. Bartalsky, O. Rovny, D. Sismisova, M. Morhac, and M. Loksik, "The digital twin of an industrial production line within the industry 4.0 concept," in *Proceedings of the 2017 21st International Conference on Process Control, PC 2017*. Institute of Electrical and Electronics Engineers Inc., 7 2017, pp. 258–262.

[15] Y. Qamsane, J. Moyne, M. Toothman, I. Kovalenko, E. C. Balta, J. Faris, D. M. Tilbury, and K. Barton, "A methodology to develop and implement digital twin solutions for manufacturing systems," *IEEE Access*, vol. 9, pp. 44 247–44 265, 2021.

[16] R. Zhao, D. Yan, Q. Liu, J. Leng, J. Wan, X. Chen, and X. Zhang, "Digital twin-driven cyber-physical system for autonomously controlling of micro punching system," *IEEE Access*, vol. 7, pp. 9459–9469, 2019.

[17] S. M. Jeon and S. Schuesslbauer, "Digital twin application for production optimization," in *IEEE International Conference on Industrial Engineering and Engineering Management*, vol. 2020-December. IEEE Computer Society, 12 2020, pp. 542–545.

[18] W. Lin, Y. Low, C. Y.T, and C. Teo, "Integrated cyber physical simulation modelling environment for manufacturing 4.0," in *2018 IEEE International Conference on Industrial Engineering and Engineering Management*, 2018, pp. 1861–1865.

[19] C. Wu, Y. Zhou, M. V. P. Pessôa, Q. Peng, and R. Tan, "Conceptual digital twin modeling based on an integrated five-dimensional framework and triz function model," *Journal of Manufacturing Systems*, vol. 58, pp. 79–93, 1 2021.

[20] J. Leng, H. Zhang, D. Yan, Q. Liu, X. Chen, and D. Zhang, "Digital twin-driven manufacturing cyber-physical system for parallel controlling of smart workshop," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, pp. 1155–1166, 3 2019.

[21] C. Zhuang, T. Miao, J. Liu, and H. Xiong, "The connotation of digital twin, and the construction and application method of shop-floor digital twin," *Robotics and Computer-Integrated Manufacturing*, vol. 68, 4 2021.

[22] T. H. Uhlemann, C. Lehmann, and R. Steinhilper, "The digital twin: Realizing the cyber-physical production system for industry 4.0," in *Procedia CIRP*, vol. 61. Elsevier B.V., 2017, pp. 335–340.

[23] F. Biesinger, D. Meike, B. Kraß, and M. Weyrich, "A digital twin for production planning based on cyber-physical systems: A case study for a cyber-physical system-based creation of a digital twin," in *Procedia CIRP*, vol. 79. Elsevier B.V., 2019, pp. 355–360.

[24] B. A. Talkhestani, D. Braun, W. Schloegl, and M. Weyrich, "Qualitative and quantita-

tive evaluation of reconfiguring an automation system using digital twin," in *Procedia CIRP*, vol. 93. Elsevier B.V., 2020, pp. 268–273.

[25] T. Borangiu, S. Raileanu, A. Silisteanu, S. Anton, and F. Anton, "Smart manufacturing control with cloud-embedded digital twins," in *2020 24th International Conference on System Theory, Control and Computing, ICSTCC 2020 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 10 2020, pp. 915–920.

[26] L. Xia, J. Lu, and H. Zhang, "Research on construction method of digital twin workshop based on digital twin engine," in *Proceedings of 2020 IEEE International Conference on Advances in Electrical Engineering and Computer Applications, AEECA 2020*. Institute of Electrical and Electronics Engineers Inc., 8 2020, pp. 417–421.

[27] D. Preuveneers, W. Joosen, and E. Ilie-Zudor, "Robust digital twin compositions for industry 4.0 smart manufacturing systems," in *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOCW*, vol. 2018-October. Institute of Electrical and Electronics Engineers Inc., 11 2018, pp. 69–78.

[28] Q. Yu-Ming, X. Bing, and D. San-Peng, "Research on intelligent manufacturing flexible production line system based on digital twin," in *Proceedings - 2020 35th Youth Academic Annual Conference of Chinese Association of Automation, YAC 2020*. Institute of Electrical and Electronics Engineers Inc., 10 2020, pp. 854–862.

[29] A. Ait-Alla, M. Kreutz, D. Rippel, M. Lütjen, and M. Freitag, "Simulation-based analysis of the interaction of a physical and a digital twin in a cyber-physical production system," in *IFAC-PapersOnLine*, vol. 52. Elsevier B.V., 9 2019, pp. 1331–1336.

[30] Z. Zhu, C. Liu, and X. Xu, "Visualisation of the digital twin data in manufacturing by using augmented reality," in *Procedia CIRP*, vol. 81. Elsevier B.V., 2019, pp. 898–903.

[31] S. E. Abramkin and S. E. Dushin, "Prospects for the development of control systems for gas producing complexes," in *Proceedings of 2017 IEEE 2nd International Conference on Control in Technical Systems*, 2017, pp. 150–153.

[32] S. P. Kovalyov and A. A. Nebera, "A platform-based approach to implementation of future smart distributed energy control systems," in *Proceedings - 2020 2nd International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency, SUMMA 2020*. Institute of Electrical and Electronics Engineers Inc., 11 2020, pp. 608–613.

[33] N. Jazdi, B. A. Talkhestani, B. Maschler, and M. Weyrich, "Realization of ai-enhanced industrial automation systems using intelligent digital twins," in *Procedia CIRP*, vol. 97. Elsevier B.V., 2020, pp. 396–400.

[34] M. Azangoo, A. Taherkordi, and J. O. Blech, "Digital twins for manufacturing using uml and behavioral specifications," in *IEEE International Conference on Emerging Technologies and Factory Automation*, 2020, pp. 1035–1038.

[35] G. Landolfi, A. Barni, S. Menato, F. A. Cavadini, D. Rovere, and G. D. Maso, "Design of a multi-sided platform supporting cps deployment in the automation market," in *Proceedings - 2018 IEEE Industrial Cyber-Physical Systems, ICPS 2018*. Institute of Electrical and Electronics Engineers Inc., 6 2018, pp. 684–689.

[36] V. Vyatkin, *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*, 1st ed. O3neida, 1 2007.

[37] E. Foundation, "Eclipse 4diac," 2007, accessed on July 1, 2021. [Online]. Available: https://www.eclipse.org/4diac/

[38] M. Jakobi, "Github tcpip4diac," 2017, accessed on July 1, 2021. [Online]. Available: https://github.com/MrcJkb/tcpip4diac

[39] C. H. Kuo, Y. F. Chuang, and S. H. Liang, "Failure mode detection and validation of a shaft-bearing system with common sensors," *Sensors*, vol. 22, 8 2022.

[40] J. Sini, A. Passarino, S. Bonicelli, and M. Violante, "A simulation-based approach to aid development of software-based hardware failure detection and mitigation algorithms of a mobile robot system," *Sensors*, vol. 22, 7 2022.

Appendix A - Detailed Equations of Motion of the Manipulator

The equations of motion presented in the body of this were presented in the form:

$$\tau = M(\theta)\ddot{\theta} + B(\theta)\dot{\theta}_i\dot{\theta}_j + C(\theta)\dot{\theta}^2 + D\dot{\theta} + G(\theta)$$

Where:

- θ is the joint angular position vector
- $\dot{\theta}$ is the joint angular velocity vector
- $\ddot{\theta}$ is the joint angular acceleration vector
- τ is the joint torques
- $M(\theta)$ is the mass matrix
- $B(\theta)$ is the Coriolis matrix
- $C(\theta)$ is the centripetal matrix
- $G(\theta)$ is the gravity vector

For the a three link manipulator of this thesis the various matrices and vectors are all of the dimension $3 \times n$, where n is either 1 or 3 depending on if a vector or matrix is in

question. The various terms are as follows:

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$\dot{\theta} = \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix}$$

$$\ddot{\theta} = \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \\ \ddot{\theta}_3 \end{bmatrix}$$

$$M(\theta) = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}$$

Where:

$$\begin{aligned}
M_{11} =& I_{z1} + I_{z2} + I_{z3} + {l_1}^2 m_2 + {l_1}^2 m_3 + {l_2}^2 m_3 + {cg_1}^2 m_1 + {cg_2}^2 m_2 + {cg_3}^2 m_3 \\
&+ 2 l_1 cg_3 m_3 \cos(\theta_2 + \theta_3) + 2 l_1 l_2 m_3 \cos(\theta_2) + 2 l_1 cg_2 m_2 \cos(\theta_2) + 2 l_2 cg_3 m_3 \cos(\theta_3) \\
M_{12} =& m_3 {l_2}^2 + 2 m_3 \cos(\theta_3) l_2 cg_3 + l_1 m_3 \cos(\theta_2) l_2 + m_2 {cg_2}^2 \\
&+ l_1 m_2 \cos(\theta_2) cg_2 + m_3 {cg_3}^2 + l_1 m_3 \cos(\theta_2 + \theta_3) cg_3 + I_{z2} + I_{z3} \\
M_{13} =& I_{z3} + {cg_3}^2 m_3 + l_1 cg_3 m_3 \cos(\theta_2 + \theta_3) + l_2 cg_3 m_3 \cos(\theta_3) \\
M_{22} =& m_3 {l_2}^2 + 2 m_3 \cos(\theta_3) l_2 cg_3 + m_2 {cg_2}^2 + m_3 {cg_3}^2 + I_{z2} + I_{z3} \\
M_{23} =& m_3 {cg_3}^2 + l_2 m_3 \cos(\theta_3) cg_3 + I_{z3} \\
M_{33} =& m_3 {cg_3}^2 + I_{z3} \\
M_{ij} =& M_{ji}
\end{aligned}$$

$$C(\theta) = \begin{bmatrix} 0 & C_{12} & C_{13} \\ C_{21} & 0 & C_{23} \\ C_{31} & C_{32} & 0 \end{bmatrix}$$

Where:

$$C_{12} = -cg_2 l_1 m_2 \sin(\theta_2) - cg_3 m_3 l_1 \sin(\theta_2 + \theta_3) - l_1 l_2 m_3 \sin(\theta_2)$$

$$C_{13} = -cg_3 l_1 m_3 \sin(\theta_2 + \theta_3) - cg_3 l_2 m_3 \sin(\theta_3)$$

$$C_{23} = -cg_3 l_2 m_3 \sin(\theta_3)$$

$$C_{ij} = -C_{ji}$$

$$\dot{\theta}_i \dot{\theta}_j = \begin{bmatrix} \dot{\theta}_1 \dot{\theta}_2 \\ \dot{\theta}_1 \dot{\theta}_3 \\ \dot{\theta}_2 \dot{\theta}_3 \end{bmatrix}$$

$$B(\theta) = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ 0 & B_{22} & B_{23} \\ B_{31} & 0 & 0 \end{bmatrix}$$

Where:

$$B_{11} = -2cg_2 l_1 m_2 \sin(\theta_2) - 2cg_3 l_1 m_3 \sin(\theta_2 + \theta_3) - 2l_1 l_2 m_3 \sin(\theta_2)$$

$$B_{31} = 2cg_3 m_3 l_2 \sin(\theta_3)$$

$$B_{13} = B_{12}$$

$$B_{22} = -B_{31}$$

$$B_{23} = -B_{31}$$

$$D = \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ 0 & 0 & D_3 \end{bmatrix}$$

$$G(\theta) = \begin{bmatrix} G_1 \\ G_2 \\ G_3 \end{bmatrix}$$

Where:

$$G_3 = cg_3 m_3 g \cos(\theta_1 + \theta_2 + \theta_3) + F_{load} l_3 \sin(\theta_4)$$

$$G_2 = cg_2 g m_2 \cos(\theta_1 + \theta_2) + cg_3 g m_3 \cos(\theta_1 + \theta_2 + \theta_3) + g l_2 m_3 \cos(\theta_1 + \theta_2)$$
$$+ F_{load}(l_2 \sin(\theta_3 + \theta_4) + l_3 \sin(\theta_4))$$

$$G_1 = cg_1 g m_1 \cos(\theta_1) + cg_2 g m_2 \cos(\theta_1 + \theta_2) + cg_3 g m_3 \cos(\theta_1 + \theta_2 + \theta_3) + g l_1 (m_2 + m_3) \cos(\theta_1)$$
$$+ g l_2 m_3 \cos(\theta_1 + \theta_2) + F_{load}(l_1 \sin(\theta_2 + \theta_3 + \theta_4) + l_2 \sin(\theta_3 + \theta_4) + l_3 \sin(\theta_4))$$

$$\theta_4 = \frac{\pi}{2} - \theta_1 - \theta_2 - \theta_3$$

Appendix B - MATLAB Code of the Manipulator Emulation and Digital Twin

Manipulator Emulation Application Code

```
% Callbacks that handle component events
    methods (Access = private)

        % Button pushed function: StartSimulationButton
        function StartSimulationButtonPushed(app, event)
            app.UIAxes.cla
            port=app.PortEditField.Value;
            time_delay_interval=0.20;
            simulation_time=app.SimulationTimesEditField.Value;
            time_step=app.TimeStepsEditField.Value;
            no_time_steps=simulation_time/time_step;
            time_delay_steps=round(time_delay_interval/time_step);
            time_delay_counter=1;
            real_time_delay=0;
            dataOutputs = {'LREAL'; 'LREAL'; 'LREAL';'LREAL';'BOOL'};
            dataInputs = {'LREAL'; 'LREAL'; 'LREAL';'LREAL';'BOOL';'STRING
    '};
            t = tcpip4diac('client','localhost',port,'DataInputs',
    dataInputs,'DataOutputs',dataOutputs);
            time=zeros(1,no_time_steps);
            input_voltage=zeros(4,no_time_steps);
            output_voltages=zeros(4,no_time_steps);
            estimated_velocity=zeros(3,no_time_steps);
            arm=three_link_arm;
            arm.three_link_arm_initialize;
            box_generation_time=round(15*rand,2)+0.5;
            boxes=cell(1);
            load_force=0;
            load_force_history=zeros(1,no_time_steps);
            box_count=0;
            box_queue = false;
```

```
30            box_generation_location=[0.20;-0.20];
31            gripper_status=false;
32            gripper_status_previous=false;
33            active_box=0;
34            queued_box=0;
35            queued_box_colour='grey';
36            g=9.81;
37            %init(t,1)
38            %app.UIAxes;
39            manipulator_position_1=animatedline(app.UIAxes);
40            manipulator_position_1.Color='red';
41            manipulator_position_2=animatedline(app.UIAxes);
42            manipulator_position_2.Color='blue';
43            manipulator_position_3=animatedline(app.UIAxes);
44            manipulator_position_3.Color='green';
45            %legend(app.UIAxes,'Link 1','Link 2','Link 3','Location','
    southoutside')
46            arm.position_speed_history(1,1)=pi/2;
47            %counter=1;
48            for counter=1:no_time_steps
49               tic
50          time(counter)=(counter-1)*time_step;
51              %box generation routine
52              if abs(time(counter) - box_generation_time) < 0.001
53               box_generation_time=box_generation_time+round(30*rand,2);
54               if box_queue == false
55                box_count=box_count+1;
56                boxes{box_count}=box;
57                boxes{box_count}.box_initialize(time(counter),
    box_generation_location);
58                queued_box=box_count;
59                queued_box_colour=boxes{box_count}.colour;
60                box_queue=true;
61               end
62              end
63
64              %data exchange and control loop
65              if mod(time(counter),0.010)==0
66           noise_1=0.0*rand;
67           noise_2=0.0*rand;
68           noise_3=0.0*rand;
69           noise_4=0.0*rand;
70           %inputData={arm.position_speed_history(1,counter)+noise_1,arm.
    position_speed_history(2,counter)+noise_2,arm.position_speed_history(3,
    counter)+noise_3,arm.position_speed_history(7,counter)+noise_4,
    box_queue,queued_box_colour};
71          %[input_voltage(1,counter),input_voltage(2,counter),
    input_voltage(3,counter),input_voltage(4,counter),gripper_status]=req(t
    ,inputData);
72            end
73            if counter<no_time_steps
74           input_voltage(:,counter+1)=input_voltage(:,counter);
75           estimated_velocity(:,counter+1)=estimated_velocity(:,counter);
76              end
77
78              %box status change
79              if gripper_status == true && gripper_status_previous == false
```

```
80          active_box = box_count;
81          boxes{active_box}.status='grb';
82          box_queue = false;
83          load_force=boxes{active_box}.mass*g;
84          queued_box=0;
85          queued_box_colour='grey';
86             elseif gripper_status == false && gripper_status_previous ==
    true
87          boxes{active_box}.status='stn';
88          load_force=0;
89          active_box=0;
90             end
91             load_force_history(counter)=load_force;
92                   %arm and box position updates
93                   %load_force=0.210*9.81;
94                   arm.rk4_three_link_arm(input_voltage(:,counter),
    time_step,load_force)
95             if active_box~=0
96                      boxes{active_box}.box_position(counter,arm,time(
    counter),time_step);
97             end
98             if queued_box~=0
99                      boxes{queued_box}.box_position(counter,arm,time(
    counter),time_step);
100            end
101                  gripper_status_previous=gripper_status;
102                  addpoints(manipulator_position_1,time(counter),arm.
    position_speed_history(1,counter));
103                  addpoints(manipulator_position_2,time(counter),arm.
    position_speed_history(2,counter));
104                  addpoints(manipulator_position_3,time(counter),arm.
    position_speed_history(3,counter));
105                  drawnow limitrate nocallbacks
106                  real_timer=toc;
107            real_time_delay=(time_step-real_timer)+real_time_delay;
108                  if mod(time_delay_counter,time_delay_steps)==0
109                      if real_time_delay>0
110                          pause(real_time_delay)
111                          real_time_delay=0;
112                          time_delay_counter=1;
113                      else
114                          real_time_delay=0;
115                          time_delay_counter=1;
116                      end
117                  else
118                      time_delay_counter=time_delay_counter+1;
119                  end
120                  %if time(counter)==45
121                  %    arm.voltage_constant_2=0.1*arm.voltage_constant_2;
122                  %end
123            end
124            %init(t,0)
125            assignin('base','arm',arm)
126       end
127    end
```

Manipulator Object Code

```
classdef three_link_arm < handle
  properties
    inertia_z_1;
    inertia_z_2;
    inertia_z_3;
    gripper_inertia;
    mass_1;
    mass_2;
    mass_3;
    link_length_1;
    link_length_2;
    link_length_3;
    cg_1;
    cg_2;
    cg_3;
    voltage_constant_1;
    voltage_constant_2;
    voltage_constant_3;
    position_speed_history;
    time_history;
    current_speed_acceleration;
    runge_kutta_k_augment;
  end
  methods
    function three_link_arm_initialize(obj)
      %primary initialization algorithm
      %note that all units must be in m-kg-s
      %no_steps=round(simulation_time/simulation_time_step);
      obj.inertia_z_1=0.00053325;
      obj.inertia_z_2=0.00037766;
      obj.inertia_z_3=0.00024296;
      obj.gripper_inertia=0.1;
      obj.mass_1=0.11804;
      obj.mass_2=0.11604;
      obj.mass_3=0.12993861;
      obj.link_length_1=0.15786938;
      obj.link_length_2=0.13246938;
      obj.link_length_3=0.1608;
      obj.cg_1=0.12;
      obj.cg_2=0.1;
      obj.cg_3=0.14;
      obj.voltage_constant_1=0.235333333;
      obj.voltage_constant_2=0.157333333;
      obj.voltage_constant_3=0.157333333;
      obj.position_speed_history=zeros(8,1);
      obj.position_speed_history(1:3,1)=[0;0;0];
      obj.position_speed_history(7,1)=0.9;
      obj.runge_kutta_k_augment=zeros(8,1);
      obj.time_history=zeros(1,1);
    end

    function [speed_acceleration] = eom_three_link_arm(obj,input_voltage,
   load_force)
      g=9.81;
```

```
54      %first, the input voltages must be converted to a value that the
    power supply circuit would produce
55      %this, in effect, completes the function of a SIFB
56      possible_voltages=0:0.0243756:6.03;
57      %lm317_voltage=interp1(possible_voltages,possible_voltages,abs(
    input_voltage),'nearest').*sign(input_voltage);
58      lm317_voltage=input_voltage;
59      [~,n]=size(obj.position_speed_history); %this retrieves the latest
    index and permits storing the position and speed history of the arm in
    the object
60      %obj.position_speed_history(:,n)
61      %obj.runge_kutta_k_augment
62      position_speed=obj.position_speed_history(:,n)+obj.
    runge_kutta_k_augment;
63      %sort the dynamic inputs
64      joint_torques=[lm317_voltage(1)*obj.voltage_constant_1;lm317_voltage
    (2)*obj.voltage_constant_2;lm317_voltage(3)*obj.voltage_constant_3];
65      angular_speed=position_speed(4:6);
66      theta=zeros(4,1);
67      theta(1:3)=position_speed(1:3);
68      theta(4)=pi/2-theta(3)-theta(2)-theta(1);
69      gripper_angle=position_speed(7);
70      gripper_speed=position_speed(8);
71      gripper_acceleration=0;
72
73
74      %construct the four matrices first
75      mass_matrix=zeros(3);
76      centrifugal_matrix=zeros(3);
77      coriolis_matrix=zeros(3);
78      gravity_matrix=zeros(3,1);
79      angular_speed_ij_matrix=zeros(3,1);
80      angular_speed_squared_matrix=zeros(3,1);
81      torque_rate=zeros(3,1);
82
83
84      %complete the calculation of the angular accelerations
85      %define the mass matrix which relates angular accelerations and
    torques
86      mass_matrix(3,3)=obj.inertia_z_3+obj.cg_3^2*obj.mass_3;
87      mass_matrix(3,2)=obj.inertia_z_3+obj.cg_3^2*obj.mass_3+obj.cg_3*obj.
    mass_3*obj.link_length_2*cos(theta(3));
88      mass_matrix(3,1)=obj.inertia_z_3+obj.cg_3^2*obj.mass_3+obj.cg_3*obj.
    mass_3*obj.link_length_2*cos(theta(3))+obj.cg_3*obj.mass_3*obj.
    link_length_1*cos(theta(2)+theta(3));
89      mass_matrix(2,3)=mass_matrix(3,2);
90      mass_matrix(2,2)=obj.inertia_z_2+ obj.inertia_z_3+ obj.cg_2^2*obj.
    mass_2 + obj.cg_3^2*obj.mass_3 + 2*obj.cg_3*obj.link_length_2*obj.
    mass_3*cos(theta(3)) + obj.link_length_2^2*obj.mass_3;
91      mass_matrix(2,1)=obj.inertia_z_2 + obj.inertia_z_3 + obj.cg_2^2*obj.
    mass_2 + obj.cg_2*obj.link_length_1*obj.mass_2*cos(theta(2)) + obj.cg_3
    ^2*obj.mass_3 + obj.cg_3*obj.link_length_1*obj.mass_3*cos(theta(2) +
    theta(3)) + 2*obj.cg_3*obj.link_length_2*obj.mass_3*cos(theta(3)) + obj
    .link_length_1*obj.link_length_2*obj.mass_3*cos(theta(2)) + obj.
    link_length_2^2*obj.mass_3;
92      mass_matrix(1,3)=mass_matrix(3,1);
93      mass_matrix(1,2)=mass_matrix(2,1);
```

```
94      mass_matrix(1,1)=obj.inertia_z_1 + obj.inertia_z_2 + obj.inertia_z_3
     + obj.cg_1^2*obj.mass_1 + obj.cg_2^2*obj.mass_2 + 2*obj.cg_2*obj.
    link_length_1*obj.mass_2*cos(theta(2)) + obj.cg_3^2*obj.mass_3 + 2*obj.
    cg_3*obj.link_length_1*obj.mass_3*cos(theta(2) + theta(3)) + 2*obj.cg_3
    *obj.link_length_2*obj.mass_3*cos(theta(3)) + obj.link_length_1^2*obj.
    mass_2 + obj.link_length_1^2*obj.mass_3 + 2*obj.link_length_1*obj.
    link_length_2*obj.mass_3*cos(theta(2)) + obj.link_length_2^2*obj.mass_3
    ;
95
96      %define the centrifugal matrix
97      centrifugal_matrix(3,1)=obj.mass_3*obj.cg_3*(obj.link_length_1*sin(
    theta(2)+theta(3))+obj.link_length_2*sin(theta(3)));
98      centrifugal_matrix(3,2)=obj.mass_3*obj.cg_3*obj.link_length_2*sin(
    theta(3));
99      centrifugal_matrix(2,1)=obj.cg_2*obj.mass_2*obj.link_length_1*sin(
    theta(2))+obj.cg_3*obj.link_length_1*obj.mass_3*sin(theta(2)+theta(3))+
    obj.link_length_1*obj.link_length_2*obj.mass_3*sin(theta(2));
100     centrifugal_matrix(2,3)=-obj.cg_3*obj.link_length_2*obj.mass_3*sin(
    theta(3));
101     centrifugal_matrix(1,2)=-obj.cg_2*obj.link_length_1*obj.mass_2*sin(
    theta(2))-obj.cg_3*obj.mass_3*obj.link_length_1*sin(theta(2)+theta(3))-
    obj.link_length_1*obj.link_length_2*obj.mass_3*sin(theta(2));
102     centrifugal_matrix(1,3)=-obj.cg_3*obj.link_length_1*obj.mass_3*sin(
    theta(2)+theta(3))-obj.cg_3*obj.link_length_2*obj.mass_3*sin(theta(3));
103
104     %define the coriolis matrix
105     coriolis_matrix(3,1)=obj.cg_3*obj.mass_3*2*obj.link_length_2*sin(
    theta(3));
106     coriolis_matrix(2,2)=-coriolis_matrix(3,1);
107     coriolis_matrix(2,3)=coriolis_matrix(2,2);
108     coriolis_matrix(1,1)=-2*obj.cg_2*obj.link_length_1*obj.mass_2*sin(
    theta(2))-2*obj.cg_3*obj.link_length_1*obj.mass_3*sin(theta(2)+theta(3)
    )-2*obj.link_length_1*obj.link_length_2*obj.mass_3*sin(theta(2));
109     coriolis_matrix(1,2)=-2*obj.cg_3*obj.link_length_1*obj.mass_3*sin(
    theta(2)+theta(3))-2*obj.cg_3*obj.link_length_2*obj.mass_3*sin(theta(3)
    );
110     coriolis_matrix(1,3)=-2*obj.cg_3*obj.link_length_1*obj.mass_3*sin(
    theta(2)+theta(3))-2*obj.cg_3*obj.link_length_2*obj.mass_3*sin(theta(3)
    );
111
112     %define gravity matrix
113     gravity_matrix(3)=obj.cg_3*obj.mass_3*g*cos(theta(1)+theta(2)+theta
    (3))+load_force*obj.link_length_3*sin(theta(4));
114     gravity_matrix(2)=obj.cg_2*g*obj.mass_2*cos(theta(1)+theta(2))+obj.
    cg_3*g*obj.mass_3*cos(theta(1)+theta(2)+theta(3))+g*obj.link_length_2*
    obj.mass_3*cos(theta(1)+theta(2))+load_force*(obj.link_length_2*sin(
    theta(3)+theta(4))+obj.link_length_3*sin(theta(4)));
115     gravity_matrix(1)=obj.cg_1*g*obj.mass_1*cos(theta(1))+obj.cg_2*g*obj
    .mass_2*cos(theta(1)+theta(2))+obj.cg_3*g*obj.mass_3*cos(theta(1)+theta
    (2)+theta(3))+g*obj.link_length_1*(obj.mass_2+obj.mass_3)*cos(theta(1))
    +g*obj.link_length_2*obj.mass_3*cos(theta(1)+theta(2))+load_force*(obj.
    link_length_1*sin(theta(2)+theta(3)+theta(4))+obj.link_length_2*sin(
    theta(3)+theta(4))+obj.link_length_3*sin(theta(4)));
116     %disp(gravity_matrix)
117     %define angular speed matrices and damping matrix
118     damping_matrix=[0.05  0 0; 0 0.05 0; 0 0 0.05];
119     angular_speed_ij_matrix=[angular_speed(1)*angular_speed(2);
```

```
   angular_speed(1)*angular_speed(3);angular_speed(2)*angular_speed(3)];
120    angular_speed_squared_matrix=angular_speed.^2;
121
122    %calculate the angular acceleration
123    angular_acceleration=mass_matrix\(joint_torques - damping_matrix*
   angular_speed - centrifugal_matrix*angular_speed_squared_matrix -
   coriolis_matrix*angular_speed_ij_matrix - gravity_matrix);
124
125    %gripper dynamics
126    if lm317_voltage(4) < 0
127      if gripper_angle <= 0.39265
128        %applied_force=2000000*0.000080645*(0.0508-(gripper_angle
   *0.0517+0.0305))/0.0508;
129        %gripper_acceleration=(lm317_voltage(4)*0.0118/0.075+
   applied_force*0.0508-0.05*gripper_speed)/obj.gripper_inertia;
130        if gripper_speed ~= 0
131          gripper_acceleration= -gripper_speed/0.001;
132        else
133          gripper_acceleration=0;
134        end
135      else
136        gripper_acceleration=(lm317_voltage(4)*0.0118/0.075-0.05*
   gripper_speed)/obj.gripper_inertia;
137      end
138    elseif lm317_voltage(4)>=0
139      if gripper_angle >= 0.9
140        if gripper_speed ~= 0
141          gripper_acceleration= -gripper_speed/0.001;
142        else
143          gripper_acceleration=0;
144        end
145      else;
146        gripper_acceleration=(lm317_voltage(4)*0.0118/0.075-0.05*
   gripper_speed)/obj.gripper_inertia;
147      end
148    end
149    speed_acceleration=[angular_speed;angular_acceleration;gripper_speed
   ;gripper_acceleration];
150   end
151   function rk4_three_link_arm(obj,input_voltage,time_step,load_force)
152    k1=time_step*obj.eom_three_link_arm(input_voltage,load_force);
153    obj.runge_kutta_k_augment=k1/2;
154    k2=time_step*obj.eom_three_link_arm(input_voltage,load_force);
155    obj.runge_kutta_k_augment=k2/2;
156    k3=time_step*obj.eom_three_link_arm(input_voltage,load_force);
157    obj.runge_kutta_k_augment=k3;
158    k4=time_step*obj.eom_three_link_arm(input_voltage,load_force);
159    obj.runge_kutta_k_augment=zeros(8,1);
160    [~,n]=size(obj.position_speed_history);
161    obj.position_speed_history(:,n+1)=obj.position_speed_history(:,n)+(
   k1+2*k2+2*k3+k4)/6;
162    obj.time_history(n+1)=obj.time_history(n)+time_step;
163   end
```

System Identification Module Application Code

```
    % Callbacks that handle component events
    methods (Access = private)

        % Button pushed function: STARTSYSTEMIDENTIFICATIONButton
        function STARTSYSTEMIDENTIFICATIONButtonPushed(app, event)
            module=arm_digital_twin_system_identification_module_v2;
            app.TextArea.Value='System identification now running';
            drawnow
            if isempty(app.PreviousModelNameEditField.Value)
                module.initialize;
            else
                module.initialize(app.PreviousModelNameEditField.Value);
            end
            req(module.system_communication_object,true);
            module.model_build;
            new_model_result=sim(module.system_model,module.
system_identification_data);
            old_model_result=sim(module.system_model_prev,module.
system_identification_data);
            measured_output=module.system_identification_data;
            measured_theta_1=measured_output.y(:,1);
            measured_theta_2=measured_output.y(:,2);
            measured_theta_3=measured_output.y(:,3);
            old_theta_1=old_model_result.y(:,1);
            old_theta_2=old_model_result.y(:,2);
            old_theta_3=old_model_result.y(:,3);
            new_theta_1=new_model_result.y(:,1);
            new_theta_2=new_model_result.y(:,2);
            new_theta_3=new_model_result.y(:,3);
            time_samples=module.system_identification_data.
SamplingInstants;
            plot(app.UIAxes,time_samples,measured_theta_1,'x',time_samples
,old_theta_1)
            %legend(app.UIAxes,'Measured Response','Simulated Response','
Location','southoutside')
            plot(app.UIAxes_2,time_samples,measured_theta_2,'x',
time_samples,old_theta_2)
            %legend(app.UIAxes_2,'Measured Response','Simulated Response
','Location','southoutside')
            plot(app.UIAxes_3,time_samples,measured_theta_3,'x',
time_samples,old_theta_3)
            %legend(app.UIAxes_3,'Measured Response','Simulated Response
','Location','southoutside')
            plot(app.UIAxes_4,time_samples,measured_theta_1,'x',
time_samples,new_theta_1)
            %legend(app.UIAxes_4,'Measured Response','Simulated Response
','Location','southoutside')
            plot(app.UIAxes_5,time_samples,measured_theta_2,'x',
time_samples,new_theta_2)
            %legend(app.UIAxes_5,'Measured Response','Simulated Response
','Location','southoutside')
            plot(app.UIAxes_6,time_samples,measured_theta_3,'x',
time_samples,new_theta_3)
            %legend(app.UIAxes_6,'Measured Response','Simulated Response
```

```
    ','Location','southoutside')
41              %hold on
42              new_result_1=goodnessOfFit(new_theta_1,measured_theta_1,'NRMSE
    ')*100;
43              new_result_2=goodnessOfFit(new_theta_2,measured_theta_2,'NRMSE
    ')*100;
44              new_result_3=goodnessOfFit(new_theta_3,measured_theta_3,'NRMSE
    ')*100;
45              old_result_1=goodnessOfFit(old_theta_1,measured_theta_1,'NRMSE
    ')*100;
46              old_result_2=goodnessOfFit(old_theta_2,measured_theta_2,'NRMSE
    ')*100;
47              old_result_3=goodnessOfFit(old_theta_3,measured_theta_3,'NRMSE
    ')*100;
48              app.OldModelJoint1AccuracyTextArea.Value=num2str(old_result_1)
    ;
49              app.OldModelJoint2AccuracyTextArea.Value=num2str(old_result_2)
    ;
50              app.OldModelJoint3AccuracyTextArea.Value=num2str(old_result_3)
    ;
51              app.NewModelJoint1AccuracyTextArea.Value=num2str(new_result_1)
    ;
52              app.NewModelJoint2AccuracyTextArea.Value=num2str(new_result_2)
    ;
53              app.NewModelJoint3AccuracyTextArea.Value=num2str(new_result_3)
    ;
54              drawnow
55              assignin('base','module',module)
56              %drawnow
57              app.TextArea.Value='System identification completed. Type name
     of model and press "ACCEPT" if keeping the model. Press "REJECT" if
    not satisfied and model will be deleted';
58          end
59
60          % Button pushed function: ACCEPTButton
61          function ACCEPTButtonPushed(app, event)
62              module=evalin('base','module');
63              save(app.NewModelNameEditField.Value,module.system_model,'-mat
    ')
64              app.TextArea.Value='New model saved';
65          end
66
67          % Button pushed function: REJECTButton
68          function REJECTButtonPushed(app, event)
69
70          end
71      end
```

System Identification Module Object Code

```
classdef arm_digital_twin_system_identification_module_v2 < handle
  properties
    system_model;
    system_model_prev;
    system_identification_data;
    estimation_data;
    universal_counter;
    universal_time_step;
    system_communication_object;
    initialization_status;
    controller_override;
    system_identification_request;
    z_data;
  end
  methods
    function initialize(obj,old_model_name)
      obj.universal_counter=0;
      obj.universal_time_step=0.01;
      digital_twin_Outputs = {};
      digital_twin_Inputs = {'BOOL'};
      obj.system_communication_object = tcpip4diac('client','localhost'
    ,61501,'DataInputs',digital_twin_Inputs,'DataOutputs',
    digital_twin_Outputs);
      init(obj.system_communication_object,1)
      obj.controller_override=false;
      obj.system_identification_request=false;
      if nargin<2
        obj.model_initialization;
      else
        obj.model_initialization(old_model_name);
      end
      obj.initialization_status=true;
    end

    function model_initialization(obj,old_model_name)
      %disp(nargin)
      if nargin<2
        disp(1)
        %in the case that no old model is specified, the default model is
    loaded
        InputName = {'Voltage applied to motor moving arm 1'; ...
                     'Voltage applied to motor moving arm 2'; ...
                     'Voltage applied to motor moving arm 3'; ...
                     'Weight of payload'};
        InputUnit = {'V'; 'V'; 'V'; 'N'};
        StateName  = {'\Theta_1'; ...   % Relative angle between fundament
    and arm 1
                      '\Theta_2'; ...   % Relative angle between arm 1 and
    arm 2
                      '\Theta_3'; ...   % Relative angle between arm 2 and
    arm 3
                      'Vel_1'; ...     % Relative velocity between
    fundament and arm 1
                      'Vel_2'; ...     % Relative velocity between arm 1
```

```
and arm 2
                    'Vel_3'...       % Relative velocity between arm 2
and arm 3
                    };
     StateUnit  = {'rad'; 'rad'; 'rad'; 'rad/s'; 'rad/s'; 'rad/s'};
     OutputName = StateName(1:3);
     OutputUnit = StateUnit(1:3);
     ParName  = {'Voltage-force constants of motor';           ... % Fc
, 3-by-1 vector, for motor 1, 2, 3.
                 'Mass of link 1, 2 and 3';          ... % m, 3-by-1
vector, for arms 1,2 and 3.
                 'Moment of inertia arm 1, 2 and 3';    ... % Izz, 3-by
-1 vector scalar.
                 'Lengths of arm 1, 2 and 3';        ... % L, 3-by-1
vector, for arm 1, 2 and 3.
                 'Center of mass coordinates of arm 1, 2 and 3'; ... %
com, 3-by-1 matrix,
                 'Damping constants'; ... %viscous damping in the
various motors
                 'Friction constands';... %coloumbic friction in the
various motors
                 'Gravity constant';                          ... % g,
scalar.
                };
     ParUnit  = {'N*m/V'; 'kg'; 'kg'; 'm'; 'm'; 'kg*m^2';'N*m';'m/s^2'
};
     %these are initial guess values, they will be reconfigured once
the model optimization is run
     ParValue = {[0.2353; 0.1573; 0.1573]; [0.12; 0.12; 0.13];
[0.00053; 0.00038; 0.00024]; ...
                 [0.15786938;0.13246938; 0.1608]; [0.12;0.1;0.14];
[0.07;0.03;0.04]; [0.01;0.01;0.01]; 9.81};
     %ParValue = {[0.2353; 0.1573; 0.1573]; [0.11804; 0.11604;
0.12993861]; [0.00053325; 0.00037766; 0.00024296.
     %            [0.15786938;0.13246938; 0.1608]; [0.12;0.1;0.14];
[0.05;0.05;0.05]; 9.81};
     ParMin   = { -Inf(3, 1); eps(0)*ones(3, 1); eps(0)*ones(3,1); ...
                 eps(0)*ones(3, 1); eps(0)*ones(3, 1); eps(0)*ones(3,
1);eps(0)*ones(3, 1); eps(0)};
     ParMax   = {[1;1;1];[0.15;0.15;0.15];[0.1;0.1;0.1];...
           [0.2;0.2;0.2];[0.15;0.15;0.15];[0.1;0.1;0.1];[0.1;0.1;0.1];
Inf};   % No maximum constraint.%      %
     %ParMax   = Inf;   % No maximum constraint.%      %
     FileName      = 'three_link_arm_m_v2';      % File describing the
model structure.
     Order         = [3 4 6];                 % Model orders [ny nu nx
].
     Parameters    = struct('Name', ParName, 'Unit', ParUnit, 'Value',
ParValue, ...
                            'Minimum', ParMin, 'Maximum', ParMax, '
Fixed', false);
     InitialStateValue={0;0;0;0;0;0};
     InitialStates = struct('Name', StateName, 'Unit', StateUnit, '
Value', InitialStateValue, ...
                           'Minimum', -Inf, 'Maximum', Inf, 'Fixed',
true);
     Ts            = 0;                        %continous time system
```

```
81        nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,
    ...
82                        'Name', 'ThreeLinkManipulator', 'InputName',
  InputName, ...
83                        'InputUnit', InputUnit, 'OutputName', OutputName,
    ...
84                        'OutputUnit', OutputUnit,  'TimeUnit', 's');%
85        obj.system_model_prev=nlgr;
86      else
87        %in the case that an old model is specified, load this and use it
  as the model to compare it
88        disp(2)
89        saved_model=load(old_model_name);
90        obj.system_model_prev=saved_model.system_model;
91      end
92    end
93
94    function model_build(obj)
95      dataOutputs_system_identification = {'LREAL'; 'LREAL'; 'LREAL'; '
  LREAL'; 'LREAL'; 'LREAL'; 'BOOL'};
96      dataInputs_system_identification = {};
97      system_identification_communication_object = tcpip4diac('client','
  localhost',61504,'DataInputs',dataInputs_system_identification,'
  DataOutputs',dataOutputs_system_identification);
98
99      init(system_identification_communication_object,1)
100     system_identifaction_counter=1;
101     cycle_completion=false;
102
103     recorded_position=zeros(3,1);
104     recorded_voltage=zeros(3,1);
105     [recorded_position(1,system_identifaction_counter),recorded_position
  (2,system_identifaction_counter),recorded_position(3,
  system_identifaction_counter),...
106     recorded_voltage(1,system_identifaction_counter),recorded_voltage(2,
  system_identifaction_counter),recorded_voltage(3,
  system_identifaction_counter),...
107     cycle_completion]=req(system_identification_communication_object);
108
109     while cycle_completion==false
110       system_identifaction_counter=system_identifaction_counter+1;
111       [recorded_position(1,system_identifaction_counter),
  recorded_position(2,system_identifaction_counter),recorded_position(3,
  system_identifaction_counter),...
112       recorded_voltage(1,system_identifaction_counter),recorded_voltage
  (2,system_identifaction_counter),recorded_voltage(3,
  system_identifaction_counter),...
113       cycle_completion]=req(system_identification_communication_object);
114     end
115     init(system_identification_communication_object,0)
116     initial_velocity_1=(1/obj.universal_time_step)*(-49/20*
  recorded_position(1,1)+6*recorded_position(1,2)-15/2*recorded_position
  (1,3)+20/3*recorded_position(1,4)-15/4*recorded_position(1,5)+6/5*
  recorded_position(1,6)-1/6*recorded_position(1,7));
117     initial_velocity_2=(1/obj.universal_time_step)*(-49/20*
  recorded_position(2,1)+6*recorded_position(2,2)-15/2*recorded_position
  (2,3)+20/3*recorded_position(2,4)-15/4*recorded_position(2,5)+6/5*
```

```
   recorded_position(2,6)-1/6*recorded_position(2,7));
118      initial_velocity_3=(1/obj.universal_time_step)*(-49/20*
   recorded_position(3,1)+6*recorded_position(3,2)-15/2*recorded_position
   (3,3)+20/3*recorded_position(3,4)-15/4*recorded_position(3,5)+6/5*
   recorded_position(3,6)-1/6*recorded_position(3,7));
119
120      InitialStateValue=[recorded_position(1,1); recorded_position(2,1);
   recorded_position(3,1); initial_velocity_1; initial_velocity_2;
   initial_velocity_3];
121      [~,no_measures]=size(recorded_position);
122      recorded_position=recorded_position(:,1:no_measures-1);
123      recorded_voltage=recorded_voltage(:,1:no_measures-1);
124      load_force_recorded=ones(1,no_measures-1)*0.200*9.81;
125      measured_data=recorded_position';
126      disp(measured_data)
127      input_data=[recorded_voltage',load_force_recorded'];
128      obj.system_identification_data=iddata(measured_data,input_data,obj.
   universal_time_step)
129      newInitialStateValue=InitialStateValue';
130      for i=1:6
131        obj.system_model_prev.InitialStates(i).Value=newInitialStateValue(
   i);
132      end
133      nlgr=obj.system_model_prev;
134      obj.system_identification_data.Name = 'Three Link Arm';
135      obj.system_identification_data.InputName = nlgr.InputName;
136      obj.system_identification_data.InputUnit = nlgr.InputUnit;
137      obj.system_identification_data.OutputName = nlgr.OutputName;
138      obj.system_identification_data.OutputUnit = nlgr.OutputUnit;
139      obj.system_identification_data.Tstart = 0;
140      obj.system_identification_data.TimeUnit = 's';
141      nlgr.SimulationOptions.RelTol = 1e-5;
142      nlgr.Parameters(end).Fixed=true;
143      opt = nlgreyestOptions('SearchMethod', 'lm', 'Display', 'on');
144      disp(obj.system_identification_data)
145      nlgr = nlgreyest(obj.system_identification_data, nlgr, opt);
146      obj.system_model=nlgr;
147    end
148
149    function twin_run(obj)
150      %this function is used to manage the digital twin
151      %tic
152      if obj.initialization_status~=true
153        obj.initialize;
154      end
155
156      [obj.system_identification_request]=req(obj.
   system_communication_object);
157
158      %disp('system communication successful')
159
160      if obj.system_identification_request==true
161        obj.model_build;
162      end
163    end
164  end
165 end
```

System Monitoring Module Application Code

```
% Callbacks that handle component events
    methods (Access = private)

        % Value changed function: SystemMonitoringSwitch
        function SystemMonitoringSwitchValueChanged(app, event)
            value = app.SystemMonitoringSwitch.Value;
            module=arm_digital_twin_system_monitoring_module;
            module.initialize(app.ModelNameEditField.Value,app.
    CommunicationPortEditField.Value);
            app.Link1StatusTextArea.Value=module.system_monitoring_message
    {1};
            app.Link2StatusTextArea.Value=module.system_monitoring_message
    {2};
            app.Link3StatusTextArea.Value=module.system_monitoring_message
    {3};
            drawnow
            %counter=1;
            simulated_link_1_position=animatedline(app.UIAxes,'LineStyle',
    '--','Color','blue');
            measured_link_1_position=animatedline(app.UIAxes,'Marker','x',
    'Color','red');
            simulated_link_2_position=animatedline(app.UIAxes2,'LineStyle'
    ,'--','Color','blue');
            measured_link_2_position=animatedline(app.UIAxes2,'Marker','x'
    ,'Color','red');
            simulated_link_3_position=animatedline(app.UIAxes3,'LineStyle'
    ,'--','Color','blue');
            measured_link_3_position=animatedline(app.UIAxes3,'Marker','x'
    ,'Color','red');
            while value=='Run'
                module.run;
                app.Link1StatusTextArea.Value=module.
    system_monitoring_message{1};
                app.Link2StatusTextArea.Value=module.
    system_monitoring_message{2};
                app.Link3StatusTextArea.Value=module.
    system_monitoring_message{3};
                drawnow
                addpoints(simulated_link_1_position,module.
    system_monitoring_troubleshooting(4,module.counter_thing-1),module.
    system_monitoring_troubleshooting(1,module.counter_thing-1))
                addpoints(measured_link_1_position,module.
    system_monitoring_reading_history(4,module.
    system_monitoring_reading_history_counter-1),module.
    system_monitoring_reading_history(1,module.
    system_monitoring_reading_history_counter-1))
                addpoints(simulated_link_2_position,module.
    system_monitoring_troubleshooting(4,module.counter_thing-1),module.
    system_monitoring_troubleshooting(2,module.counter_thing-1))
                addpoints(measured_link_2_position,module.
    system_monitoring_reading_history(4,module.
    system_monitoring_reading_history_counter-1),module.
    system_monitoring_reading_history(2,module.
    system_monitoring_reading_history_counter-1))
```

```
30                addpoints(simulated_link_3_position,module.
   system_monitoring_troubleshooting(4,module.counter_thing-1),module.
   system_monitoring_troubleshooting(3,module.counter_thing-1))
31                addpoints(measured_link_3_position,module.
   system_monitoring_reading_history(4,module.
   system_monitoring_reading_history_counter-1),module.
   system_monitoring_reading_history(3,module.
   system_monitoring_reading_history_counter-1))
32                drawnow limitrate nocallbacks
33                value=app.SystemMonitoringSwitch.Value;
34            end
35            if value=='Off'
36                init(module.system_monitoring_object_receiving,0);
37                app.Link1StatusTextArea.Value='System monitoring not
   running';
38                app.Link2StatusTextArea.Value='System monitoring not
   running';
39                app.Link3StatusTextArea.Value='System monitoring not
   running';
40                drawnow
41                assignin('base','module',module);
42            end
43
44        end
45    end
```

System Monitoring Module Object Code

```
classdef arm_digital_twin_system_monitoring_module < handle
  properties
    system_model;
    system_model_prev;
    system_identification_data;
    estimation_data;
    universal_counter;
    universal_time_step;
    system_communication_object;
    initialization_status;
    controller_override;
    system_identification_request;
    system_monitoring_history;
    system_monitoring_initialization;
    system_monitoring_initialization_counter;
    monitoring_degredation_tolerance;
    monitoring_error_tolerance;
    control_error;
    control_error_integral;
    simulation_time_step;
    system_monitoring_counter;
    system_monitoring_override;
    system_monitoring_object_sending;
    system_monitoring_object_receiving;
    system_monitoring_message;
    system_monitoring_reading;
    system_monitoring_state;
    system_monitoring_troubleshooting;
    system_monitoring_time;
    controller_gains;
    trajectory_parameters;
    control_target_time;
    control_time;
    theta_f_prev;
    counter_thing;
    set_point_troubleshooting;
    counter_thing_2;
    system_monitoring_reading_history;
    system_monitoring_reading_history_counter;
    error_history;
    error_history_counter;
    t_sim;
    t_samp;
    cntrl_freq;
    end_count;
    degradation_count;
    degradation_count_max;
    position_error_relative;
    position_error;
    relative_error_tolerance;
    absolute_error_tolerance;
  end
  methods
    function initialize(obj,model_name,port)
```

```
55      system_monitoring_receiving_Outputs = {'LREAL','LREAL','LREAL','
    LREAL','LREAL','LREAL','BOOL','LREAL'};
56      system_monitoring_receiving_Inputs = {};
57      obj.system_monitoring_object_receiving = tcpip4diac('client','
    localhost',port,'DataInputs',system_monitoring_receiving_Inputs,'
    DataOutputs',system_monitoring_receiving_Outputs);
58      init(obj.system_monitoring_object_receiving,1)
59      obj.system_identification_request=false;
60      obj.initialization_status=true;
61      obj.system_monitoring_history=zeros(3,3);
62      obj.monitoring_degredation_tolerance=0.02*ones(3,1);
63      obj.monitoring_error_tolerance=0.05*ones(3,1);
64      obj.absolute_error_tolerance=0.1;
65      obj.relative_error_tolerance=0.05;
66      obj.system_monitoring_initialization=false;
67      obj.system_monitoring_initialization_counter=1;
68      obj.control_error=zeros(3,3);
69      obj.control_error_integral=zeros(3,1);
70      obj.simulation_time_step=0.010;
71      obj.system_monitoring_override=false;
72      %obj.model_initialization(model_name);
73      obj.system_monitoring_state=zeros(6,3);
74      obj.controller_gains=[200,500,300;2,2,2,;75,200,150];
75      obj.system_monitoring_counter=1;
76      obj.control_time=0;
77      obj.control_target_time=10;
78      obj.theta_f_prev=zeros(3,1);
79      obj.trajectory_parameters=zeros(3,3);
80      obj.system_monitoring_troubleshooting=zeros(4,1);
81      obj.system_monitoring_time=0;
82      obj.counter_thing=1;
83      obj.counter_thing_2=1;
84      obj.universal_time_step=0.010;
85      obj.set_point_troubleshooting=zeros(4,15000);
86      obj.system_monitoring_reading=zeros(3,1);
87      obj.system_monitoring_reading_history=zeros(4,1);
88      obj.system_monitoring_reading_history_counter=1;
89      obj.error_history=zeros(4,1);
90      obj.error_history_counter=1;
91      obj.t_sim=0.005;
92      obj.t_samp=0.5;
93      obj.cntrl_freq=round(0.010/obj.t_sim);
94      obj.end_count=round(obj.t_samp/obj.t_sim);
95      obj.system_monitoring_message={'System performing nominally';'System
     performing nominally';'System performing nominally'}
96      saved_model=load(model_name);
97      obj.system_model=saved_model.module.system_model;
98      obj.degradation_count=zeros(3,1);
99      obj.degradation_count_max=10;
100     %disp(obj.end_count)
101
102   end
103
104   function model_initialization(obj,model_name)
105     InputName = {'Voltage applied to motor moving arm 1'; ...
106                  'Voltage applied to motor moving arm 2'; ...
107                  'Voltage applied to motor moving arm 3'; ...
```

```
108                  'Weight of payload'};
109     InputUnit = {'V'; 'V'; 'V'; 'N'};%
110
111     StateName  = {'\Theta_1'; ...   % Relative angle between fundament
    and arm 1
112                  '\Theta_2'; ...   % Relative angle between arm 1 and
    arm 2
113                  '\Theta_3'; ...   % Relative angle between arm 2 and
    arm 3
114                  'Vel_1'; ...      % Relative velocity between fundament
     and arm 1
115                  'Vel_2'; ...      % Relative velocity between arm 1 and
     arm 2
116                  'Vel_3'...        % Relative velocity between arm 2 and
     arm 3
117                  };
118     StateUnit  = {'rad'; 'rad'; 'rad'; 'rad/s'; 'rad/s'; 'rad/s'};
119     OutputName = StateName(1:3);
120     OutputUnit = StateUnit(1:3);%
121
122     ParName  = {'Voltage-force constants of motor';            ... % Fc,
    3-by-1 vector, for motor 1, 2, 3.
123                 'Mass of link 1, 2 and 3';          ... % m, 3-by-1
    vector, for arms 1,2 and 3.
124                 'Moment of inertia arm 1, 2 and 3';    ... % Izz, 3-by-1
     vector scalar.
125                 'Lengths of arm 1, 2 and 3';        ... % L, 3-by-1
    vector, for arm 1, 2 and 3.
126                 'Center of mass coordinates of arm 1, 2 and 3'; ... %
    com, 3-by-1 matrix,
127                 'Damping constants'; ... %viscous damping in the various
     motors
128                 'Gravity constant';                            ... % g,
    scalar.
129                };
130     ParUnit  = {'N*m/V'; 'kg'; 'kg'; 'm'; 'm'; 'kg*m^2';'m/s^2'};
131     %these are initial guess values, they will be reconfigured once the
    model optimization is run
132     %ParValue = {[0.2353; 0.1573; 0.1573]; [0.12; 0.12; 0.13]; [0.00053;
     0.00038; 0.00024]; ...
133     %            [0.15786938;0.13246938; 0.1608]; [0.12;0.1;0.14];
    [0.07;0.03;0.04]; 9.81};
134     ParValue = {[0.2353; 0.1573; 0.1573]; [0.11804; 0.11604;
    0.12993861]; [0.00053325; 0.00037766; 0.00024296]; ...
135                 [0.15786938;0.13246938; 0.1608]; [0.12;0.1;0.14];
    [0.05;0.05;0.05]; 9.81};
136     ParMin   = { -Inf(3, 1); eps(0)*ones(3, 1); eps(0)*ones(3,1); ...
137                  eps(0)*ones(3, 1); eps(0)*ones(3, 1); eps(0)*ones(3, 1);
     eps(0)};
138     ParMax   = Inf;   % No maximum constraint.%
139
140     FileName       = 'three_link_arm_m';      % File describing the model
     structure.
141     Order          = [3 4 6];                 % Model orders [ny nu nx].
142     Parameters     = struct('Name', ParName, 'Unit', ParUnit, 'Value',
    ParValue, ...
143                             'Minimum', ParMin, 'Maximum', ParMax, 'Fixed'
```

```
, false);
144       InitialStateValue={0;0;0;0;0;0}
145       InitialStates = struct('Name', StateName, 'Unit', StateUnit, 'Value'
, InitialStateValue, ...
146                              'Minimum', -Inf, 'Maximum', Inf, 'Fixed', true
);
147       Ts             = 0;                          %continous time system
148       nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,
...
149                       'Name', 'ThreeLinkManipulator', 'InputName',
InputName, ...
150                       'InputUnit', InputUnit, 'OutputName', OutputName,
...
151                       'OutputUnit', OutputUnit,  'TimeUnit', 's');%
152       saved_model=load('model_name');
153       obj.system_model=saved_model.system_model;
154     end
155
156     function system_monitoring(obj)
157       %sample the present position knowing that the controller has a new
    pathway
158       theta_f=zeros(3,1);
159       [obj.system_monitoring_reading(1),obj.system_monitoring_reading(2),
    obj.system_monitoring_reading(3),theta_f(1),theta_f(2),theta_f(3),
    gripper_status,system_time]=req(obj.system_monitoring_object_receiving)
    ;
160       obj.system_monitoring_reading_history(1:3,obj.
    system_monitoring_reading_history_counter)=obj.
    system_monitoring_reading;
161       obj.system_monitoring_reading_history(4,obj.
    system_monitoring_reading_history_counter)=system_time;
162       obj.system_monitoring_reading_history_counter=obj.
    system_monitoring_reading_history_counter+1;
163       %disp(theta_f)
164       if obj.system_monitoring_time~=system_time
165         obj.system_monitoring_time=system_time;
166         obj.counter_thing=ceil(obj.system_monitoring_time*obj.end_count);
167       end
168
169       if obj.theta_f_prev(1)~=theta_f(1) || obj.theta_f_prev(2)~=theta_f
    (2) || obj.theta_f_prev(3)~=theta_f(3)
170         %in the case that the reset signal is true, we must reset the
    system monitoring state and the controller
171         obj.control_time=0;
172         obj.control_error_integral=zeros(3,1);
173         obj.control_error=zeros(3,3);
174         obj.system_monitoring_state(1:3,3)=obj.system_monitoring_reading
    (1:3);
175         obj.system_monitoring_state(1:3,1)=obj.system_monitoring_state
    (1:3,3);
176         obj.system_monitoring_state(1:3,2)=obj.system_monitoring_state
    (1:3,3);
177         obj.system_monitoring_state(4:6,1:3)=zeros(3);
178       else
179         %should the reset signal not be true, we must determine if there
    are any deviations
180         position_error_relative=abs((obj.system_monitoring_reading(1:3)-
```

```
    obj.system_monitoring_state(1:3,3))./obj.system_monitoring_reading(1:3)
    );
181       position_error=abs((obj.system_monitoring_reading(1:3)-obj.
    system_monitoring_state(1:3,3)));
182       obj.error_history(1:3,obj.error_history_counter)=position_error;
183       obj.error_history(4,obj.error_history_counter)=obj.
    system_monitoring_time;
184       obj.error_history_counter=obj.error_history_counter+1;
185       for counter=1:3
186         if obj.degradation_count(counter)>=obj.degradation_count_max
187           obj.system_monitoring_message{counter}='Deviation detected,
    check system immediately!';
188         elseif position_error(counter)>obj.absolute_error_tolerance &&
    position_error_relative(counter)>obj.relative_error_tolerance
189           obj.system_monitoring_message{counter}='System may be
    degrading!';
190           obj.degradation_count(counter)=obj.degradation_count(counter)
    +1;
191           if obj.degradation_count(counter)>=obj.degradation_count_max
192             obj.system_monitoring_message{counter}='Deviation detected,
    check system immediately!';
193           end
194         else
195           obj.system_monitoring_message{counter}='System performing
    nominally';
196           obj.degradation_count(counter)=0;
197         end
198       end
199     end
200     if gripper_status==true
201       load_force=0.20*9.81;
202     else
203       load_force=0;
204     end
205
206     for counter=1:obj.end_count
207       if mod(counter-1,obj.cntrl_freq)==0
208         obj.system_monitoring_state(1:6,1)=obj.system_monitoring_state
    (1:6,2);
209         obj.system_monitoring_state(1:6,2)=obj.system_monitoring_state
    (1:6,3);
210         [set_point,desired_acceleration]=obj.path_planner(theta_f,obj.
    system_monitoring_state(1:3,3),0.010);
211         control_voltage=obj.computed_torque_controller(obj.
    system_monitoring_state(1:3,1:3),set_point,desired_acceleration,
    load_force);
212         obj.counter_thing_2=obj.counter_thing_2+1;
213         if counter-1==0
214           obj.theta_f_prev=theta_f;
215         end
216       end
217     obj.system_monitoring_state(1:6,3)=obj.rk4_solver(obj.t_sim,obj.
    system_monitoring_state(1:6,3),control_voltage,load_force);
218     obj.system_monitoring_time=obj.system_monitoring_time+obj.t_sim;
219     obj.system_monitoring_troubleshooting(1:3,obj.counter_thing)=obj.
    system_monitoring_state(1:3,3);
220     obj.system_monitoring_troubleshooting(4,obj.counter_thing)=obj.
```

```
system_monitoring_time;
221       obj.counter_thing=obj.counter_thing+1;
222       end
223     end
224
225     function [link_accelerations]=system_simulator(obj,
    position_velocity_data,voltage,load_force)
226       %this function is used to model the system in dynamic simulations,
    you use it
227
228       param=obj.system_model.Parameters;
229       voltage_force_constant=param(1).Value;
230       link_mass=param(2).Value;
231       link_inertia=param(3).Value;
232       link_length=param(4).Value;
233       link_cg=param(5).Value;
234       damping=param(6).Value;
235
236       g=9.81;
237       position_data=position_velocity_data(1:3);
238       angular_speed=position_velocity_data(4:6);
239       pv_4=1.570796-position_data(1)-position_data(2)-position_data(3);
240
241       %calculation of the mass matrix terms
242       mass_matrix=zeros(3,3);
243       mass_matrix(3,3)=link_inertia(3)+link_cg(3)^2*link_mass(3);
244       mass_matrix(3,2)=mass_matrix(3,3)+link_cg(3)*link_mass(3)*
    link_length(2)*cos(position_data(3));
245       mass_matrix(3,1)=mass_matrix(3,2)+link_cg(3)*link_mass(3)*
    link_length(1)*cos(position_data(2)+position_data(3));
246       mass_matrix(2,3)=mass_matrix(3,2);
247       mass_matrix(2,2)=link_inertia(2)+ link_inertia(3)+ link_cg(2)^2*
    link_mass(2) + link_cg(3)^2*link_mass(3) + 2*link_cg(3)*link_length(2)*
    link_mass(3)*cos(position_data(3)) + link_length(2)^2*link_mass(3);
248       mass_matrix(2,1)=link_inertia(2) + link_inertia(3) + link_cg(2)^2*
    link_mass(2) + link_cg(2)*link_length(1)*link_mass(2)*cos(position_data
    (2)) + link_cg(3)^2*link_mass(3) + link_cg(3)*link_length(1)*link_mass
    (3)*cos(position_data(2) + position_data(3)) + 2*link_cg(3)*link_length
    (2)*link_mass(3)*cos(position_data(3)) + link_length(1)*link_length(2)*
    link_mass(3)*cos(position_data(2)) + link_length(2)^2*link_mass(3);
249       mass_matrix(1,3)=mass_matrix(3,1);
250       mass_matrix(1,2)=mass_matrix(2,1);
251       mass_matrix(1,1)=link_inertia(1) + link_inertia(2) + link_inertia(3)
     + link_cg(1)^2*link_mass(1) + link_cg(2)^2*link_mass(2) + 2*link_cg(2)
    *link_length(1)*link_mass(2)*cos(position_data(2)) + link_cg(3)^2*
    link_mass(3) + 2*link_cg(3)*link_length(1)*link_mass(3)*cos(
    position_data(2) + position_data(3)) + 2*link_cg(3)*link_length(2)*
    link_mass(3)*cos(position_data(3)) + link_length(1)^2*link_mass(2) +
    link_length(1)^2*link_mass(3) + 2*link_length(1)*link_length(2)*
    link_mass(3)*cos(position_data(2)) + link_length(2)^2*link_mass(3);


252
253       %calculation of the centrifugal matrix terms
254       centrifugal_matrix=zeros(3,3);
255       centrifugal_matrix(3,1)=link_mass(3)*link_cg(3)*(link_length(1)*sin(
    position_data(2)+position_data(3))+link_length(2)*sin(position_data(3))
    );
```

```
256     centrifugal_matrix(3,2)=link_mass(3)*link_cg(3)*link_length(2)*sin(
    position_data(3));
257     centrifugal_matrix(2,1)=link_cg(2)*link_mass(2)*link_length(1)*sin(
    position_data(2))+link_cg(3)*link_length(1)*link_mass(3)*sin(
    position_data(2)+position_data(3))+link_length(1)*link_length(2)*
    link_mass(3)*sin(position_data(2));
258     centrifugal_matrix(2,3)=-link_cg(3)*link_length(2)*link_mass(3)*sin(
    position_data(3));
259     centrifugal_matrix(1,2)=-link_cg(2)*link_length(1)*link_mass(2)*sin(
    position_data(2))-link_cg(3)*link_mass(3)*link_length(1)*sin(
    position_data(2)+position_data(3))-link_length(1)*link_length(2)*
    link_mass(3)*sin(position_data(2));
260     centrifugal_matrix(1,3)=-link_cg(3)*link_length(1)*link_mass(3)*sin(
    position_data(2)+position_data(3))-link_cg(3)*link_length(2)*link_mass
    (3)*sin(position_data(3));
261
262     %calculation of the coriolis matrix terms
263     coriolis_matrix=zeros(3,3);
264     coriolis_matrix(3,1)=link_cg(3)*link_mass(3)*2*link_length(2)*sin(
    position_data(3));
265     coriolis_matrix(2,2)=-coriolis_matrix(3,1);
266     coriolis_matrix(2,3)=coriolis_matrix(2,2);
267     coriolis_matrix(1,1)=-2*link_cg(2)*link_length(1)*link_mass(2)*sin(
    position_data(2))-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
    position_data(2)+position_data(3))-2*link_length(1)*link_length(2)*
    link_mass(3)*sin(position_data(2));
268     coriolis_matrix(1,2)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
    position_data(2)+position_data(3))-2*link_cg(3)*link_length(2)*
    link_mass(3)*sin(position_data(3));
269     coriolis_matrix(1,3)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
    position_data(2)+position_data(3))-2*link_cg(3)*link_length(2)*
    link_mass(3)*sin(position_data(3));
270
271     %calculation of the gravity matrix terms
272     gravity_vector=zeros(3,1);
273     gravity_vector(3)=link_cg(3)*link_mass(3)*g*cos(position_data(1)+
    position_data(2)+position_data(3))+load_force*link_length(3)*sin(pv_4);
274     gravity_vector(2)=link_cg(2)*g*link_mass(2)*cos(position_data(1)+
    position_data(2))+link_cg(3)*g*link_mass(3)*cos(position_data(1)+
    position_data(2)+position_data(3))+g*link_length(2)*link_mass(3)*cos(
    position_data(1)+position_data(2))+load_force*(link_length(2)*sin(
    position_data(1)+position_data(2))+link_length(3)*sin(pv_4));
275     gravity_vector(1)=link_cg(1)*g*link_mass(1)*cos(position_data(1))+
    link_cg(2)*g*link_mass(2)*cos(position_data(1)+position_data(2))+
    link_cg(3)*g*link_mass(3)*cos(position_data(1)+position_data(2)+
    position_data(3))+g*link_length(1)*(link_mass(2)+link_mass(3))*cos(
    position_data(1))+g*link_length(2)*link_mass(3)*cos(position_data(1)+
    position_data(2))+load_force*(link_length(1)*sin(position_data(2)+
    position_data(3)+pv_4)+link_length(2)*sin(position_data(3)+pv_4)+
    link_length(3)*sin(pv_4));
276
277     %calculation of the torques
278     joint_torques=voltage.*voltage_force_constant;
279
280     damping_matrix=[damping(1)  0 0; 0 damping(2) 0; 0 0 damping(3)];
281     angular_speed_ij_matrix=[angular_speed(1)*angular_speed(2);
    angular_speed(1)*angular_speed(3);angular_speed(2)*angular_speed(3)];
```

```
282       angular_speed_squared_matrix=angular_speed.^2;
283
284       %calculate accelerations
285       link_accelerations=zeros(6,1);
286       link_accelerations(1:3,1)=angular_speed;
287       link_accelerations(4:6,1)=mass_matrix\(joint_torques -
      damping_matrix*angular_speed - centrifugal_matrix*
      angular_speed_squared_matrix - coriolis_matrix*angular_speed_ij_matrix
      - gravity_vector);
288
289    end
290
291    function [voltage] = computed_torque_controller(obj,position_data,
      set_point,desired_acceleration,load_force)
292       %this function takes all
293       param=obj.system_model.Parameters;
294       voltage_force_constant=param(1).Value;
295       link_mass=param(2).Value;
296       link_inertia=param(3).Value;
297       link_length=param(4).Value;
298       link_cg=param(5).Value;
299
300       g=9.81;
301
302       KP_1=obj.controller_gains(1,1); KP_2=obj.controller_gains(1,2); KP_3
      =obj.controller_gains(1,3);
303       KD_1=obj.controller_gains(2,1); KD_2=obj.controller_gains(2,2); KD_3
      =obj.controller_gains(2,3);
304       KI_1=obj.controller_gains(3,1); KI_2=obj.controller_gains(3,2); KI_3
      =obj.controller_gains(3,3);
305
306       obj.control_error(:,1)=obj.control_error(:,2);
307       obj.control_error(:,2)=obj.control_error(:,3);
308
309       obj.control_error(1,3)=set_point(1)-position_data(1,3);
310       obj.control_error(2,3)=set_point(2)-position_data(2,3);
311       obj.control_error(3,3)=set_point(3)-position_data(3,3);
312
313       error_derivative=(3*obj.control_error(:,3)-4*obj.control_error(:,2)+
      obj.control_error(:,1))/(2*obj.universal_time_step);
314
315       obj.control_error_integral=obj.control_error_integral+obj.
      universal_time_step*(obj.control_error(:,3)+obj.control_error(:,2))/2;
316
317       link_speeds=(3*position_data(:,3)-4*position_data(:,2)+position_data
      (:,1))/(2*obj.universal_time_step);
318
319       pv_4=1.570796-position_data(1,3)-position_data(2,3)-position_data
      (3,3);
320
321       %calculation of the mass matrix terms
322       mass_matrix=zeros(3,3);
323       mass_matrix(3,3)=link_inertia(3)+link_cg(3)^2*link_mass(3);
324       mass_matrix(3,2)=mass_matrix(3,3)+link_cg(3)*link_mass(3)*
      link_length(2)*cos(position_data(3,3));
325       mass_matrix(3,1)=mass_matrix(3,2)+link_cg(3)*link_mass(3)*
      link_length(1)*cos(position_data(2,3)+position_data(3,3));
```

```
326       mass_matrix(2,3)=mass_matrix(3,2);
327       mass_matrix(2,2)=link_inertia(2)+ link_inertia(3)+ link_cg(2)^2*
      link_mass(2) + link_cg(3)^2*link_mass(3) + 2*link_cg(3)*link_length(2)*
      link_mass(3)*cos(position_data(3,3)) + link_length(2)^2*link_mass(3);
328       mass_matrix(2,1)=link_inertia(2) + link_inertia(3) + link_cg(2)^2*
      link_mass(2) + link_cg(2)*link_length(1)*link_mass(2)*cos(position_data
      (2,3)) + link_cg(3)^2*link_mass(3) + link_cg(3)*link_length(1)*
      link_mass(3)*cos(position_data(2,3) + position_data(3,3)) + 2*link_cg
      (3)*link_length(2)*link_mass(3)*cos(position_data(3,3)) + link_length
      (1)*link_length(2)*link_mass(3)*cos(position_data(2,3)) + link_length
      (2)^2*link_mass(3);
329       mass_matrix(1,3)=mass_matrix(3,1);
330       mass_matrix(1,2)=mass_matrix(2,1);
331       mass_matrix(1,1)=link_inertia(1) + link_inertia(2) + link_inertia(3)
       + link_cg(1)^2*link_mass(1) + link_cg(2)^2*link_mass(2) + 2*link_cg(2)
      *link_length(1)*link_mass(2)*cos(position_data(2,3)) + link_cg(3)^2*
      link_mass(3) + 2*link_cg(3)*link_length(1)*link_mass(3)*cos(
      position_data(2,3) + position_data(3,3)) + 2*link_cg(3)*link_length(2)*
      link_mass(3)*cos(position_data(3,3)) + link_length(1)^2*link_mass(2) +
      link_length(1)^2*link_mass(3) + 2*link_length(1)*link_length(2)*
      link_mass(3)*cos(position_data(2,3)) + link_length(2)^2*link_mass(3);

332
333       %calculation of the centrifugal matrix terms
334       centrifugal_matrix=zeros(3,3);
335       centrifugal_matrix(3,1)=link_mass(3)*link_cg(3)*(link_length(1)*sin(
      position_data(2,3)+position_data(3,3))+link_length(2)*sin(position_data
      (3,3)));
336       centrifugal_matrix(3,2)=link_mass(3)*link_cg(3)*link_length(2)*sin(
      position_data(3,3));
337       centrifugal_matrix(2,1)=link_cg(2)*link_mass(2)*link_length(1)*sin(
      position_data(2,3))+link_cg(3)*link_length(1)*link_mass(3)*sin(
      position_data(2,3)+position_data(3,3))+link_length(1)*link_length(2)*
      link_mass(3)*sin(position_data(2,3));
338       centrifugal_matrix(2,3)=-link_cg(3)*link_length(2)*link_mass(3)*sin(
      position_data(3,3));
339       centrifugal_matrix(1,2)=-link_cg(2)*link_length(1)*link_mass(2)*sin(
      position_data(2,3))-link_cg(3)*link_mass(3)*link_length(1)*sin(
      position_data(2,3)+position_data(3,3))-link_length(1)*link_length(2)*
      link_mass(3)*sin(position_data(2,3));
340       centrifugal_matrix(1,3)=-link_cg(3)*link_length(1)*link_mass(3)*sin(
      position_data(2,3)+position_data(3,3))-link_cg(3)*link_length(2)*
      link_mass(3)*sin(position_data(3,3));
341
342       %calculation of the coriolis matrix terms
343       coriolis_matrix=zeros(3,3);
344       coriolis_matrix(3,1)=link_cg(3)*link_mass(3)*2*link_length(2)*sin(
      position_data(3,3));
345       coriolis_matrix(2,2)=-coriolis_matrix(3,1);
346       coriolis_matrix(2,3)=coriolis_matrix(2,2);
347       coriolis_matrix(1,1)=-2*link_cg(2)*link_length(1)*link_mass(2)*sin(
      position_data(2,3))-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
      position_data(2,3)+position_data(3,3))-2*link_length(1)*link_length(2)*
      link_mass(3)*sin(position_data(2,3));
348       coriolis_matrix(1,2)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
      position_data(2,3)+position_data(3,3))-2*link_cg(3)*link_length(2)*
      link_mass(3)*sin(position_data(3,3));
```

```
349     coriolis_matrix(1,3)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
    position_data(2,3)+position_data(3,3))-2*link_cg(3)*link_length(2)*
    link_mass(3)*sin(position_data(3,3));
350
351     %calculation of the gravity matrix terms
352     gravity_vector=zeros(3,1);
353     gravity_vector(1)=link_cg(3)*link_mass(3)*g*cos(position_data(1,3)+
    position_data(2,3)+position_data(3,3))+load_force*link_length(3)*sin(
    pv_4);
354     gravity_vector(2)=link_cg(2)*g*link_mass(2)*cos(position_data(1,3)+
    position_data(2,3))+link_cg(3)*g*link_mass(3)*cos(position_data(1,3)+
    position_data(2,3)+position_data(3,3))+g*link_length(2)*link_mass(3)*
    cos(position_data(1,3)+position_data(2,3))+load_force*(link_length(2)*
    sin(position_data(1,3)+position_data(2,3))+link_length(3)*sin(pv_4));
355     gravity_vector(3)=link_cg(1)*g*link_mass(1)*cos(position_data(1,3))+
    link_cg(2)*g*link_mass(2)*cos(position_data(1,3)+position_data(2,3))+
    link_cg(3)*g*link_mass(3)*cos(position_data(1,3)+position_data(2,3)+
    position_data(3,3))+g*link_length(1)*(link_mass(2)+link_mass(3))*cos(
    position_data(1,3))+g*link_length(2)*link_mass(3)*cos(position_data
    (1,3)+position_data(2,3))+load_force*(link_length(1)*sin(position_data
    (2,3)+position_data(3,3)+pv_4)+link_length(2)*sin(position_data(3,3)+
    pv_4)+link_length(3)*sin(pv_4));
356
357     %calculation of the control efforts based on the PID controller
    component
358     CONTROL_EFFORT_1=KP_1*obj.control_error(1,3)+KD_1*error_derivative
    (1)+KI_1*obj.control_error_integral(1);
359     CONTROL_EFFORT_2=KP_2*obj.control_error(2,3)+KD_2*error_derivative
    (2)+KI_2*obj.control_error_integral(2);
360     CONTROL_EFFORT_3=KP_3*obj.control_error(3,3)+KD_3*error_derivative
    (3)+KI_3*obj.control_error_integral(3);
361
362     %calculation of the torques
363     torques=zeros(3,1);
364     torques(1)=mass_matrix(1,:)*(desired_acceleration+CONTROL_EFFORT_1*
    ones(3,1));
365     torques(2)=mass_matrix(2,:)*(desired_acceleration+CONTROL_EFFORT_2*
    ones(3,1));
366     torques(3)=mass_matrix(3,:)*(desired_acceleration+CONTROL_EFFORT_3*
    ones(3,1));
367
368     link_speeds_ij=zeros(3,1);
369     link_speeds_ij(1)=link_speeds(1)*link_speeds(2);
370     link_speeds_ij(2)=link_speeds(1)*link_speeds(3);
371     link_speeds_ij(3)=link_speeds(2)*link_speeds(3);
372
373     torques=torques+centrifugal_matrix*link_speeds.^2+coriolis_matrix*
    link_speeds_ij+gravity_vector;
374
375     %calculate the voltages based on the motor resistances
376     voltage=torques./voltage_force_constant;
377
378  end
379
380  function [set_point,desired_acceleration] = path_planner(obj,theta_f,
    position_data,control_time_step)
381     if obj.theta_f_prev(1)~=theta_f(1) || obj.theta_f_prev(2)~=theta_f
```

```
(2) || obj.theta_f_prev(3)~=theta_f(3)
382         %this calculates trajectory in the case that the controller is
    reset for a new cycle
383         %trajectory parameters are stored as: [a_01,a_21,a_31;a_02,a_22,
    a_32;a_03,a_32,a_33]
384         obj.trajectory_parameters(1,1)=position_data(1);
385         obj.trajectory_parameters(1,2)=3*(theta_f(1)-position_data(1))/obj
    .control_target_time^2;
386         obj.trajectory_parameters(1,3)=-2*(theta_f(1)-position_data(1))/
    obj.control_target_time^3;
387         obj.trajectory_parameters(2,1)=position_data(2);
388         obj.trajectory_parameters(2,2)=3*(theta_f(2)-position_data(2))/obj
    .control_target_time^2;
389         obj.trajectory_parameters(2,3)=-2*(theta_f(2)-position_data(2))/
    obj.control_target_time^3;
390         obj.trajectory_parameters(3,1)=position_data(3);
391         obj.trajectory_parameters(3,2)=3*(theta_f(3)-position_data(3))/obj
    .control_target_time^2;
392         obj.trajectory_parameters(3,3)=-2*(theta_f(3)-position_data(3))/
    obj.control_target_time^3;
393       end
394       set_point=zeros(3,1);
395       desired_acceleration=zeros(3,1);
396       if obj.control_time<=obj.control_target_time
397         %in the case the the control time is less than the target, the
    trajectory follows the
398         set_point(1)=obj.trajectory_parameters(1,1)+obj.
    trajectory_parameters(1,2)*obj.control_time^2+obj.trajectory_parameters
    (1,3)*obj.control_time^3;
399         set_point(2)=obj.trajectory_parameters(2,1)+obj.
    trajectory_parameters(2,2)*obj.control_time^2+obj.trajectory_parameters
    (2,3)*obj.control_time^3;
400         set_point(3)=obj.trajectory_parameters(3,1)+obj.
    trajectory_parameters(3,2)*obj.control_time^2+obj.trajectory_parameters
    (3,3)*obj.control_time^3;
401         desired_acceleration(1)=-2*obj.trajectory_parameters(1,2)+6*obj.
    trajectory_parameters(1,3)*obj.control_time;
402         desired_acceleration(2)=-2*obj.trajectory_parameters(2,2)+6*obj.
    trajectory_parameters(2,3)*obj.control_time;
403         desired_acceleration(3)=-2*obj.trajectory_parameters(3,2)+6*obj.
    trajectory_parameters(3,3)*obj.control_time;
404       else
405         set_point(1)=theta_f(1);
406         set_point(2)=theta_f(2);
407         set_point(3)=theta_f(3);
408       end
409       obj.control_time=obj.control_time+control_time_step;
410
411     end
412
413     function [next_position_speed] = rk4_solver(obj,time_step,
    position_velocity_data,voltage,load_force)
414       k1=time_step*obj.system_simulator(position_velocity_data,voltage,
    load_force);
415       k2=time_step*obj.system_simulator(position_velocity_data+k1/2,
    voltage,load_force);
416       k3=time_step*obj.system_simulator(position_velocity_data+k2/2,
```

```
    voltage,load_force);
417     k4=time_step*obj.system_simulator(position_velocity_data+k3,voltage,
    load_force);
418     next_position_speed=position_velocity_data+(k1+2*k2+2*k3+k4)/6;
419    end
420
421    function run(obj)
422     obj.system_monitoring;
423    end
424  end
425 end
```

System Simulation Module Application Code

```
    % Callbacks that handle component events
    methods (Access = private)

        % Button pushed function: BeginSimulationButton
        function BeginSimulationButtonPushed(app, event)
            controller_gains=zeros(3);
            controller_gains(1,1)=app.KP_1EditField.Value;
            controller_gains(2,1)=app.KI_1EditField.Value;
            controller_gains(3,1)=app.KD_1EditField.Value;
            controller_gains(1,2)=app.KP_2EditField.Value;
            controller_gains(2,2)=app.KI_2EditField.Value;
            controller_gains(3,2)=app.KD_2EditField.Value;
            controller_gains(1,3)=app.KP_3EditField.Value;
            controller_gains(2,3)=app.KI_3EditField.Value;
            controller_gains(3,3)=app.KD_3EditField.Value;
            load_force=app.ObjectMassgEditField.Value/1000*9.81;
            theta_f=zeros(3,1);
            theta_f_prev=ones(3,1);
            if strcmp(app.LinkEndPositionDropDown.Value,'Zero')==1
                 theta_f(1)=0;
     theta_f(2)=0;
     theta_f(3)=0;
            elseif strcmp(app.LinkEndPositionDropDown.Value,'Pickup
    Location')==1
                 theta_f(1)=-0.6699;
     theta_f(2)=-1.5981;
     theta_f(3)=2.259;
            elseif strcmp(app.LinkEndPositionDropDown.Value,'Red Box Drop
    Off')==1
                theta_f(1)=2.1855;
                theta_f(2)=-3.5259;
    theta_f(3)=1.3404;
            elseif strcmp(app.LinkEndPositionDropDown.Value,'Green Box
    Drop Off')==1
                theta_f(1)=2.5907;
    theta_f(2)=-2.2251;
    theta_f(3)=-0.3656;
            elseif strcmp(app.LinkEndPositionDropDown.Value,'Blue Box Drop
     Off')==1
                theta_f(1)=1.8516;
    theta_f(2)=-0.5321;
    theta_f(3)=-1.3915;
            end
            starting_position=zeros(3,1);
            if strcmp(app.LinkStartPositionDropDown.Value,'Zero')==1
                 starting_position(1)=0;
     starting_position(2)=0;
     starting_position(3)=0;
            elseif strcmp(app.LinkStartPositionDropDown.Value,'Pickup
    Location')==1
                 starting_position(1)=-0.6699;
     starting_position(2)=-1.5981;
     starting_position(3)=2.259;
            elseif strcmp(app.LinkStartPositionDropDown.Value,'Red Box
```

```
Drop Off')==1
50              starting_position(1)=2.1855;
51              starting_position(2)=-3.5259;
52     starting_position(3)=1.3404;
53          elseif strcmp(app.LinkStartPositionDropDown.Value,'Green Box
    Drop Off')==1
54              starting_position(1)=2.5907;
55     starting_position(2)=-2.2251;
56     starting_position(3)=-0.3656;
57          elseif strcmp(app.LinkStartPositionDropDown.Value,'Blue Box
    Drop Off')==1
58              starting_position(1)=1.8516;
59     starting_position(2)=-0.5321;
60     starting_position(3)=-1.3915;
61          end
62          simulation_time=app.SimulationTimesEditField.Value;
63          time_step=app.TimeStepsEditField.Value;
64          no_time_steps=simulation_time/time_step;
65          time=zeros(1,no_time_steps);
66          arm=three_link_arm_model;
67          if isempty(app.ModelNameEditField.Value)
68              arm.initialization(controller_gains,starting_position,app.
    ControlTimesEditField.Value);
69          else
70              arm.initialization(controller_gains,starting_position,app.
    ControlTimesEditField.Value,app.ModelNameEditField.Value);
71          end
72          arm.theta_f_prev=theta_f_prev;
73          g=9.81;
74          planned_path=zeros(3,no_time_steps);
75          app.MessageCentreTextArea.Value='Simulation now running';
76          drawnow
77          for counter=1:no_time_steps
78             tic
79        time(counter)=(counter-1)*time_step;
80            if mod(time(counter),0.010)==0
81         arm.position_data(1:3,1)=arm.position_data(1:3,2);
82                    arm.position_data(4,1)=arm.position_data(4,2);
83         arm.position_data(1:3,2)=arm.position_data(1:3,3);
84                    arm.position_data(4,2)=arm.position_data(4,3);
85                    arm.position_data(1:3,3)=arm.position_speed_history
    (1:3,counter);
86                    arm.position_data(4,3)=arm.position_speed_history(7,
    counter);
87         [set_point,desired_acceleration]=arm.path_planner(theta_f,arm.
    position_data(1:3,3),0.010);
88         control_voltage=arm.computed_torque_controller(set_point,
    desired_acceleration,load_force,theta_f);
89          end
90
91                  planned_path(:,counter)=set_point;
92                  arm.rk4_three_link_arm(control_voltage,time_step,
    load_force)
93          end
94          app.MessageCentreTextArea.Value='Simulation complete';
95          drawnow
96          plot(app.UIAxes,time(:),arm.position_speed_history(1,1:counter
```

```
    ),time(:),planned_path(1,1:counter),'--')
97              plot(app.UIAxes_2,time(:),arm.position_speed_history(2,1:
    counter),time(:),planned_path(2,1:counter),'--')
98              plot(app.UIAxes_3,time(:),arm.position_speed_history(3,1:
    counter),time(:),planned_path(3,1:counter),'--')
99              assignin('base','arm',arm)
100         end
```

System Simulation Module Object Code

```
classdef three_link_arm_model < handle
  properties
    system_model;
    universal_time_step;
    system_communication_object;
    controller_override;
    system_identification_request;
    z_data;
    position_speed_history;
    time_history;
    current_speed_acceleration;
    runge_kutta_k_augment;
    position_data;
    control_error;
    control_error_integral;
    simulation_time_step;
    controller_gains;
    trajectory_parameters;
    control_target_time;
    control_time;
    theta_f_prev;
    counter_thing_2;
    error_history;
    error_history_counter;
    t_sim;
    t_samp;
    cntrl_freq;
    end_count;
    gripper_status;
    cycle_indicator;
    gripper_inertia;
  end
  methods
    function initialization(obj,controller_gains,starting_position,
   control_target_time,model_name)
      obj.position_speed_history=zeros(8,1);
      obj.position_speed_history(1:3,1)=starting_position;
      obj.position_speed_history(7,1)=0.9;
      obj.runge_kutta_k_augment=zeros(8,1);
      obj.time_history=zeros(1,1);
      obj.position_data=zeros(4,3);
      obj.controller_gains=controller_gains;
      obj.gripper_status=false;
      obj.cycle_indicator=false;
      obj.theta_f_prev=zeros(3,1);
      obj.control_error=zeros(3);
      obj.universal_time_step=0.010;
      obj.control_error_integral=zeros(3,1);
      obj.control_target_time=control_target_time;
      obj.runge_kutta_k_augment=0;
      obj.control_time=0;
      obj.trajectory_parameters=zeros(3);
      obj.gripper_inertia=0.1;
      if nargin<5
```

```
54        %disp(1)
55        %in the case that no old model is specified, the default model is
    loaded
56        InputName = {'Voltage applied to motor moving arm 1'; ...
57                     'Voltage applied to motor moving arm 2'; ...
58                     'Voltage applied to motor moving arm 3'; ...
59                     'Weight of payload'};
60        InputUnit = {'V'; 'V'; 'V'; 'N'};
61        StateName  = {'\Theta_1'; ...   % Relative angle between fundament
     and arm 1
62                      '\Theta_2'; ...   % Relative angle between arm 1 and
     arm 2
63                      '\Theta_3'; ...   % Relative angle between arm 2 and
     arm 3
64                      'Vel_1'; ...     % Relative velocity between
    fundament and arm 1
65                      'Vel_2'; ...     % Relative velocity between arm 1
    and arm 2
66                      'Vel_3'...       % Relative velocity between arm 2
    and arm 3
67                      };
68        StateUnit  = {'rad'; 'rad'; 'rad'; 'rad/s'; 'rad/s'; 'rad/s'};
69        OutputName = StateName(1:3);
70        OutputUnit = StateUnit(1:3);
71        ParName  = {'Voltage-force constants of motor';           ... % Fc
    , 3-by-1 vector, for motor 1, 2, 3.
72                    'Mass of link 1, 2 and 3';          ... % m, 3-by-1
    vector, for arms 1,2 and 3.
73                    'Moment of inertia arm 1, 2 and 3';    ... % Izz, 3-by
    -1 vector scalar.
74                    'Lengths of arm 1, 2 and 3';        ... % L, 3-by-1
    vector, for arm 1, 2 and 3.
75                    'Center of mass coordinates of arm 1, 2 and 3'; ... %
    com, 3-by-1 matrix,
76                    'Damping constants'; ... %viscous damping in the
    various motors
77                    'Gravity constant';                          ... % g,
    scalar.
78                   };
79        ParUnit  = {'N*m/V'; 'kg'; 'kg'; 'm'; 'm'; 'kg*m^2';'m/s^2'};
80        %these are initial guess values, they will be reconfigured once
    the model optimization is run
81        ParValue = {[0.2353; 0.1573; 0.1573]; [0.12; 0.12; 0.13];
    [0.00053; 0.00038; 0.00024]; ...
82                    [0.15786938;0.13246938; 0.1608]; [0.12;0.1;0.14];
    [0.07;0.03;0.04]; 9.81};
83        %ParValue = {[0.2353; 0.1573; 0.1573]; [0.11804; 0.11604;
    0.12993861]; [0.00053325; 0.00037766; 0.00024296]; ...
84        %            [0.15786938;0.13246938; 0.1608]; [0.12;0.1;0.14];
    [0.05;0.05;0.05]; 9.81};
85        ParMin   = { -Inf(3, 1); eps(0)*ones(3, 1); eps(0)*ones(3,1); ...
86                     eps(0)*ones(3, 1); eps(0)*ones(3, 1); eps(0)*ones(3,
    1); eps(0)};
87        ParMax   = Inf;   % No maximum constraint.%     %
88        FileName      = 'three_link_arm_m';      % File describing the
    model structure.
89        Order         = [3 4 6];                 % Model orders [ny nu nx
```

```
    ].
90        Parameters    = struct('Name', ParName, 'Unit', ParUnit, 'Value',
    ParValue, ...
91                                'Minimum', ParMin, 'Maximum', ParMax, '
    Fixed', false);
92        InitialStateValue={0;0;0;0;0;0};
93        InitialStates = struct('Name', StateName, 'Unit', StateUnit, '
    Value', InitialStateValue, ...
94                               'Minimum', -Inf, 'Maximum', Inf, 'Fixed',
    true);
95        Ts            = 0;                          %continous time system
96        nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,
      ...
97                        'Name', 'ThreeLinkManipulator', 'InputName',
    InputName, ...
98                        'InputUnit', InputUnit, 'OutputName', OutputName,
      ...
99                        'OutputUnit', OutputUnit,  'TimeUnit', 's');%
100       obj.system_model=nlgr;
101     else
102       %in the case that a old model is specified, load this and use it
    as the model
103       obj.position_speed_history=zeros(8,1);
104       obj.position_speed_history(1:3,1)=starting_position;
105       obj.position_speed_history(7,1)=0.9;
106       obj.runge_kutta_k_augment=zeros(8,1);
107       obj.time_history=zeros(1,1);
108       obj.position_data=zeros(4,3);
109       obj.position_data(1:3,1)=starting_position;
110       obj.position_data(1:3,2)=obj.position_data(1:3,1);
111       obj.position_data(1:3,3)=obj.position_data(1:3,2);
112       obj.controller_gains=controller_gains;
113       obj.gripper_status=false;
114       obj.cycle_indicator=false;
115       obj.theta_f_prev=zeros(3,1);
116       obj.control_error=zeros(3);
117       obj.universal_time_step=0.010;
118       obj.control_error_integral=zeros(3,1);
119       obj.control_target_time=control_target_time;
120       obj.runge_kutta_k_augment=0;
121       obj.control_time=0;
122       obj.trajectory_parameters=zeros(3);
123       obj.gripper_inertia=0.1;
124       saved_model=load(model_name);
125       obj.system_model=saved_model.module.system_model;
126     end
127   end
128
129   function [speed_acceleration] = eom_three_link_arm(obj,input_voltage,
    load_force)
130     param=obj.system_model.Parameters;
131     voltage_force_constant=param(1).Value;
132     link_mass=param(2).Value;
133     link_inertia=param(3).Value;
134     link_length=param(4).Value;
135     link_cg=param(5).Value;
136     damping=param(6).Value;
```

```
137      damping_matrix=zeros(3);
138      damping_matrix(1,1)=damping(1);
139      damping_matrix(2,2)=damping(2);
140      damping_matrix(3,3)=damping(3);
141      g=9.81;
142      %first, the input voltages must be converted to a value that the
    power supply circuit would produce
143      %this, in effect, completes the function of a SIFB
144      possible_voltages=0:0.0243756:6.03;
145      %lm317_voltage=interp1(possible_voltages,possible_voltages,abs(
    input_voltage),'nearest').*sign(input_voltage);
146      lm317_voltage=input_voltage;
147      [~,n]=size(obj.position_speed_history); %this retrieves the latest
    index and permits storing the position and speed history of the arm in
    the object
148      %obj.position_speed_history(:,n)
149      %obj.runge_kutta_k_augment
150      position_speed=obj.position_speed_history(:,n)+obj.
    runge_kutta_k_augment;
151      %sort the dynamic inputs
152      joint_torques=[0.01765*lm317_voltage(1)/0.075;0.0118*lm317_voltage
    (2)/0.075;0.0118*lm317_voltage(3)/0.075];
153      angular_speed=position_speed(4:6);
154      theta=zeros(4,1);
155      theta(1:3)=position_speed(1:3);
156      theta(4)=pi/2-theta(3)-theta(2)-theta(1);
157      pv_4=theta(4);
158      gripper_angle=position_speed(7);
159      gripper_speed=position_speed(8);
160      gripper_acceleration=0;
161
162
163      %construct the four matrices first
164      mass_matrix=zeros(3);
165      centrifugal_matrix=zeros(3);
166      coriolis_matrix=zeros(3);
167      gravity_matrix=zeros(3,1);
168      angular_speed_ij_matrix=zeros(3,1);
169      angular_speed_squared_matrix=zeros(3,1);
170      torque_rate=zeros(3,1);
171
172
173      %complete the calculation of the angular accelerations
174      %define the mass matrix which relates angular accelerations and
    torques
175      mass_matrix(3,3)=link_inertia(3)+link_cg(3)^2*link_mass(3);
176      mass_matrix(3,2)=link_inertia(3)+link_cg(3)^2*link_mass(3)+link_cg
    (3)*link_mass(3)*link_length(2)*cos(theta(3));
177      mass_matrix(3,1)=link_inertia(3)+link_cg(3)^2*link_mass(3)+link_cg
    (3)*link_mass(3)*link_length(2)*cos(theta(3))+link_cg(3)*link_mass(3)*
    link_length(1)*cos(theta(2)+theta(3));
178      mass_matrix(2,3)=mass_matrix(3,2);
179      mass_matrix(2,2)=link_inertia(2)+ link_inertia(3)+ link_cg(2)^2*
    link_mass(2) + link_cg(3)^2*link_mass(3) + 2*link_cg(3)*link_length(2)*
    link_mass(3)*cos(theta(3)) + link_length(2)^2*link_mass(3);
180      mass_matrix(2,1)=link_inertia(2) + link_inertia(3) + link_cg(2)^2*
    link_mass(2) + link_cg(2)*link_length(1)*link_mass(2)*cos(theta(2)) +
```

```
   link_cg(3)^2*link_mass(3) + link_cg(3)*link_length(1)*link_mass(3)*cos(
   theta(2) + theta(3)) + 2*link_cg(3)*link_length(2)*link_mass(3)*cos(
   theta(3)) + link_length(1)*link_length(2)*link_mass(3)*cos(theta(2)) +
   link_length(2)^2*link_mass(3);
181      mass_matrix(1,3)=mass_matrix(3,1);
182      mass_matrix(1,2)=mass_matrix(2,1);
183      mass_matrix(1,1)=link_inertia(1) + link_inertia(2) + link_inertia(3)
    + link_cg(1)^2*link_mass(1) + link_cg(2)^2*link_mass(2) + 2*link_cg(2)
   *link_length(1)*link_mass(2)*cos(theta(2)) + link_cg(3)^2*link_mass(3)
   + 2*link_cg(3)*link_length(1)*link_mass(3)*cos(theta(2) + theta(3)) +
   2*link_cg(3)*link_length(2)*link_mass(3)*cos(theta(3)) + link_length(1)
   ^2*link_mass(2) + link_length(1)^2*link_mass(3) + 2*link_length(1)*
   link_length(2)*link_mass(3)*cos(theta(2)) + link_length(2)^2*link_mass
   (3);
184      %disp('model')
185      %disp(mass_matrix)
186      %define the centrifugal matrix
187      centrifugal_matrix(3,1)=link_mass(3)*link_cg(3)*(link_length(1)*sin(
   theta(2)+theta(3))+link_length(2)*sin(theta(3)));
188      centrifugal_matrix(3,2)=link_mass(3)*link_cg(3)*link_length(2)*sin(
   theta(3));
189      centrifugal_matrix(2,1)=link_cg(2)*link_mass(2)*link_length(1)*sin(
   theta(2))+link_cg(3)*link_length(1)*link_mass(3)*sin(theta(2)+theta(3))
   +link_length(1)*link_length(2)*link_mass(3)*sin(theta(2));
190      centrifugal_matrix(2,3)=-link_cg(3)*link_length(2)*link_mass(3)*sin(
   theta(3));
191      centrifugal_matrix(1,2)=-link_cg(2)*link_length(1)*link_mass(2)*sin(
   theta(2))-link_cg(3)*link_mass(3)*link_length(1)*sin(theta(2)+theta(3))
   -link_length(1)*link_length(2)*link_mass(3)*sin(theta(2));
192      centrifugal_matrix(1,3)=-link_cg(3)*link_length(1)*link_mass(3)*sin(
   theta(2)+theta(3))-link_cg(3)*link_length(2)*link_mass(3)*sin(theta(3))
   ;
193      %disp('model')
194      %disp(centrifugal_matrix)
195      %define the coriolis matrix
196      coriolis_matrix(3,1)=link_cg(3)*link_mass(3)*2*link_length(2)*sin(
   theta(3));
197      coriolis_matrix(2,2)=-coriolis_matrix(3,1);
198      coriolis_matrix(2,3)=coriolis_matrix(2,2);
199      coriolis_matrix(1,1)=-2*link_cg(2)*link_length(1)*link_mass(2)*sin(
   theta(2))-2*link_cg(3)*link_length(1)*link_mass(3)*sin(theta(2)+theta
   (3))-2*link_length(1)*link_length(2)*link_mass(3)*sin(theta(2));
200      coriolis_matrix(1,2)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
   theta(2)+theta(3))-2*link_cg(3)*link_length(2)*link_mass(3)*sin(theta
   (3));
201      coriolis_matrix(1,3)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
   theta(2)+theta(3))-2*link_cg(3)*link_length(2)*link_mass(3)*sin(theta
   (3));
202      %disp('model')
203      %disp(coriolis_matrix)
204      %define gravity matrix
205      gravity_matrix(3)=link_cg(3)*link_mass(3)*g*cos(theta(1)+theta(2)+
   theta(3))+load_force*link_length(3)*sin(theta(4));
206      gravity_matrix(2)=link_cg(2)*g*link_mass(2)*cos(theta(1)+theta(2))+
   link_cg(3)*g*link_mass(3)*cos(theta(1)+theta(2)+theta(3))+g*link_length
   (2)*link_mass(3)*cos(theta(1)+theta(2))+load_force*(link_length(2)*sin(
   theta(3)+theta(4))+link_length(3)*sin(theta(4)));
```

```
207     gravity_matrix(1)=link_cg(1)*g*link_mass(1)*cos(theta(1))+link_cg(2)
    *g*link_mass(2)*cos(theta(1)+theta(2))+link_cg(3)*g*link_mass(3)*cos(
    theta(1)+theta(2)+theta(3))+g*link_length(1)*(link_mass(2)+link_mass(3)
    )*cos(theta(1))+g*link_length(2)*link_mass(3)*cos(theta(1)+theta(2))+
    load_force*(link_length(1)*sin(theta(2)+theta(3)+theta(4))+link_length
    (2)*sin(theta(3)+theta(4))+link_length(3)*sin(theta(4)));
208     %disp('model')
209     %disp(gravity_matrix)
210     %define angular speed matrices and damping matrix
211     %damping_matrix=[0.05  0 0; 0 0.05 0; 0 0 0.05];
212     %damping_matrix=zeros(3);
213     %damping_matrix(1,1)=
214     angular_speed_ij_matrix=[angular_speed(1)*angular_speed(2);
    angular_speed(1)*angular_speed(3);angular_speed(2)*angular_speed(3)];
215     angular_speed_squared_matrix=angular_speed.^2;
216
217     %calculate the angular acceleration
218     angular_acceleration=mass_matrix\(joint_torques - damping_matrix*
    angular_speed - centrifugal_matrix*angular_speed_squared_matrix -
    coriolis_matrix*angular_speed_ij_matrix - gravity_matrix);
219
220     %gripper dynamics
221     if lm317_voltage(4) < 0
222       if gripper_angle <= 0.39265
223         %applied_force=2000000*0.000080645*(0.0508-(gripper_angle
    *0.0517+0.0305))/0.0508;
224         %gripper_acceleration=(lm317_voltage(4)*0.0118/0.075+
    applied_force*0.0508-0.05*gripper_speed)/obj.gripper_inertia;
225         if gripper_speed ~= 0
226           gripper_acceleration= -gripper_speed/0.001;
227         else
228           gripper_acceleration=0;
229         end
230       else
231         gripper_acceleration=(lm317_voltage(4)*0.0118/0.075-0.05*
    gripper_speed)/obj.gripper_inertia;
232       end
233     elseif lm317_voltage(4)>=0
234       if gripper_angle >= 0.9
235         if gripper_speed ~= 0
236           gripper_acceleration= -gripper_speed/0.001;
237         else
238           gripper_acceleration=0;
239         end
240       else;
241         gripper_acceleration=(lm317_voltage(4)*0.0118/0.075-0.05*
    gripper_speed)/obj.gripper_inertia;
242       end
243     end
244     speed_acceleration=[angular_speed;angular_acceleration;gripper_speed
    ;gripper_acceleration];
245    end
246
247    function [voltage] = computed_torque_controller(obj,set_point,
    desired_acceleration,load_force,theta_f)
248     %this function takes all
249     position_data=obj.position_data;
```

```
250       voltage=zeros(4,1);
251       param=obj.system_model.Parameters;
252       voltage_force_constant=param(1).Value;
253       link_mass=param(2).Value;
254       link_inertia=param(3).Value;
255       link_length=param(4).Value;
256       link_cg=param(5).Value;
257
258       target_percentage=zeros(3,1);
259       target_percentage(1)=abs((position_data(1,3)-theta_f(1))/theta_f(1))
      ;
260       target_percentage(2)=abs((position_data(2,3)-theta_f(2))/theta_f(2))
      ;
261       target_percentage(3)=abs((position_data(3,3)-theta_f(3))/theta_f(3))
      ;
262
263       g=9.81;
264
265       KP_1=obj.controller_gains(1,1); KP_2=obj.controller_gains(1,2); KP_3
      =obj.controller_gains(1,3);
266       KI_1=obj.controller_gains(2,1); KI_2=obj.controller_gains(2,2); KI_3
      =obj.controller_gains(2,3);
267       KD_1=obj.controller_gains(3,1); KD_2=obj.controller_gains(3,2); KD_3
      =obj.controller_gains(3,3);
268
269
270       obj.control_error(:,1)=obj.control_error(:,2);
271       obj.control_error(:,2)=obj.control_error(:,3);
272
273       obj.control_error(1,3)=set_point(1)-position_data(1,3);
274       obj.control_error(2,3)=set_point(2)-position_data(2,3);
275       obj.control_error(3,3)=set_point(3)-position_data(3,3);
276
277       %disp(obj.control_error(:,3))
278
279       error_derivative=(3*obj.control_error(:,3)-4*obj.control_error(:,2)+
      obj.control_error(:,1))/(2*obj.universal_time_step);
280
281       obj.control_error_integral=obj.control_error_integral+obj.
      universal_time_step*(obj.control_error(:,3));
282       %disp(obj.control_error_integral)
283
284       link_speeds=(3*position_data(:,3)-4*position_data(:,2)+position_data
      (:,1))/(2*obj.universal_time_step);
285
286       pv_4=1.570796-position_data(1,3)-position_data(2,3)-position_data
      (3,3);
287
288       %calculation of the mass matrix terms
289       mass_matrix=zeros(3,3);
290       mass_matrix(3,3)=link_inertia(3)+link_cg(3)^2*link_mass(3);
291       mass_matrix(3,2)=mass_matrix(3,3)+link_cg(3)*link_mass(3)*
      link_length(2)*cos(position_data(3,3));
292       mass_matrix(3,1)=mass_matrix(3,2)+link_cg(3)*link_mass(3)*
      link_length(1)*cos(position_data(2,3)+position_data(3,3));
293       mass_matrix(2,3)=mass_matrix(3,2);
294       mass_matrix(2,2)=link_inertia(2)+ link_inertia(3)+ link_cg(2)^2*
```

```
   link_mass(2) + link_cg(3)^2*link_mass(3) + 2*link_cg(3)*link_length(2)*
   link_mass(3)*cos(position_data(3,3)) + link_length(2)^2*link_mass(3);
295      mass_matrix(2,1)=link_inertia(2) + link_inertia(3) + link_cg(2)^2*
   link_mass(2) + link_cg(2)*link_length(1)*link_mass(2)*cos(position_data
   (2,3)) + link_cg(3)^2*link_mass(3) + link_cg(3)*link_length(1)*
   link_mass(3)*cos(position_data(2,3) + position_data(3,3)) + 2*link_cg
   (3)*link_length(2)*link_mass(3)*cos(position_data(3,3)) + link_length
   (1)*link_length(2)*link_mass(3)*cos(position_data(2,3)) + link_length
   (2)^2*link_mass(3);
296      mass_matrix(1,3)=mass_matrix(3,1);
297      mass_matrix(1,2)=mass_matrix(2,1);
298      mass_matrix(1,1)=link_inertia(1) + link_inertia(2) + link_inertia(3)
    + link_cg(1)^2*link_mass(1) + link_cg(2)^2*link_mass(2) + 2*link_cg(2)
   *link_length(1)*link_mass(2)*cos(position_data(2,3)) + link_cg(3)^2*
   link_mass(3) + 2*link_cg(3)*link_length(1)*link_mass(3)*cos(
   position_data(2,3) + position_data(3,3)) + 2*link_cg(3)*link_length(2)*
   link_mass(3)*cos(position_data(3,3)) + link_length(1)^2*link_mass(2) +
   link_length(1)^2*link_mass(3) + 2*link_length(1)*link_length(2)*
   link_mass(3)*cos(position_data(2,3)) + link_length(2)^2*link_mass(3);

299      %disp('controller')
300      %disp(mass_matrix)
301      %calculation of the centrifugal matrix terms
302      centrifugal_matrix=zeros(3,3);
303      centrifugal_matrix(3,1)=link_mass(3)*link_cg(3)*(link_length(1)*sin(
   position_data(2,3)+position_data(3,3))+link_length(2)*sin(position_data
   (3,3)));
304      centrifugal_matrix(3,2)=link_mass(3)*link_cg(3)*link_length(2)*sin(
   position_data(3,3));
305      centrifugal_matrix(2,1)=link_cg(2)*link_mass(2)*link_length(1)*sin(
   position_data(2,3))+link_cg(3)*link_length(1)*link_mass(3)*sin(
   position_data(2,3)+position_data(3,3))+link_length(1)*link_length(2)*
   link_mass(3)*sin(position_data(2,3));
306      centrifugal_matrix(2,3)=-link_cg(3)*link_length(2)*link_mass(3)*sin(
   position_data(3,3));
307      centrifugal_matrix(1,2)=-link_cg(2)*link_length(1)*link_mass(2)*sin(
   position_data(2,3))-link_cg(3)*link_mass(3)*link_length(1)*sin(
   position_data(2,3)+position_data(3,3))-link_length(1)*link_length(2)*
   link_mass(3)*sin(position_data(2,3));
308      centrifugal_matrix(1,3)=-link_cg(3)*link_length(1)*link_mass(3)*sin(
   position_data(2,3)+position_data(3,3))-link_cg(3)*link_length(2)*
   link_mass(3)*sin(position_data(3,3));
309      %disp('controller')
310      %disp(centrifugal_matrix)
311      %calculation of the coriolis matrix terms
312      coriolis_matrix=zeros(3,3);
313      coriolis_matrix(3,1)=link_cg(3)*link_mass(3)*2*link_length(2)*sin(
   position_data(3,3));
314      coriolis_matrix(2,2)=-coriolis_matrix(3,1);
315      coriolis_matrix(2,3)=coriolis_matrix(2,2);
316      coriolis_matrix(1,1)=-2*link_cg(2)*link_length(1)*link_mass(2)*sin(
   position_data(2,3))-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
   position_data(2,3)+position_data(3,3))-2*link_length(1)*link_length(2)*
   link_mass(3)*sin(position_data(2,3));
317      coriolis_matrix(1,2)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
   position_data(2,3)+position_data(3,3))-2*link_cg(3)*link_length(2)*
   link_mass(3)*sin(position_data(3,3));
```

```
318     coriolis_matrix(1,3)=-2*link_cg(3)*link_length(1)*link_mass(3)*sin(
    position_data(2,3)+position_data(3,3))-2*link_cg(3)*link_length(2)*
    link_mass(3)*sin(position_data(3,3));
319     %disp('controller')
320     %disp(coriolis_matrix)
321     %calculation of the gravity matrix terms
322
323     gravity_vector=zeros(3,1);
324     gravity_vector(3)=link_cg(3)*link_mass(3)*g*cos(position_data(1,3)+
    position_data(2,3)+position_data(3,3))+load_force*link_length(3)*sin(
    pv_4);
325     gravity_vector(2)=link_cg(2)*g*link_mass(2)*cos(position_data(1,3)+
    position_data(2,3))+link_cg(3)*g*link_mass(3)*cos(position_data(1,3)+
    position_data(2,3)+position_data(3,3))+g*link_length(2)*link_mass(3)*
    cos(position_data(1,3)+position_data(2,3))+load_force*(link_length(2)*
    sin(position_data(1,3)+position_data(2,3))+link_length(3)*sin(pv_4));
326     gravity_vector(1)=link_cg(1)*g*link_mass(1)*cos(position_data(1,3))+
    link_cg(2)*g*link_mass(2)*cos(position_data(1,3)+position_data(2,3))+
    link_cg(3)*g*link_mass(3)*cos(position_data(1,3)+position_data(2,3)+
    position_data(3,3))+g*link_length(1)*(link_mass(2)+link_mass(3))*cos(
    position_data(1,3))+g*link_length(2)*link_mass(3)*cos(position_data
    (1,3)+position_data(2,3))+load_force*(link_length(1)*sin(position_data
    (2,3)+position_data(3,3)+pv_4)+link_length(2)*sin(position_data(3,3)+
    pv_4)+link_length(3)*sin(pv_4));
327     %disp('controller')
328     %disp(gravity_vector)
329     %calculation of the control efforts based on the PID controller
    component
330     CONTROL_EFFORT_1=KP_1*obj.control_error(1,3)+KD_1*error_derivative
    (1)+KI_1*obj.control_error_integral(1);
331     CONTROL_EFFORT_2=KP_2*obj.control_error(2,3)+KD_2*error_derivative
    (2)+KI_2*obj.control_error_integral(2);
332     CONTROL_EFFORT_3=KP_3*obj.control_error(3,3)+KD_3*error_derivative
    (3)+KI_3*obj.control_error_integral(3);
333
334     %calculation of the torques
335     torques=zeros(3,1);
336     torques(1)=mass_matrix(1,:)*(desired_acceleration+CONTROL_EFFORT_1*
    ones(3,1));
337     torques(2)=mass_matrix(2,:)*(desired_acceleration+CONTROL_EFFORT_2*
    ones(3,1));
338     torques(3)=mass_matrix(3,:)*(desired_acceleration+CONTROL_EFFORT_3*
    ones(3,1));
339
340     link_speeds_ij=zeros(3,1);
341     link_speeds_ij(1)=link_speeds(1)*link_speeds(2);
342     link_speeds_ij(2)=link_speeds(1)*link_speeds(3);
343     link_speeds_ij(3)=link_speeds(2)*link_speeds(3);
344
345     torques=torques+centrifugal_matrix*link_speeds(1:3).^2+
    coriolis_matrix*link_speeds_ij+gravity_vector;
346
347     %calculate the voltages based on the motor resistances
348
349     %calculate the voltages based on the motor resistances
350     voltage(1:3,1)=torques./voltage_force_constant;
351     %calculate gripper voltage required
```

```
352       if target_percentage(1)<=0.001 && target_percentage(2)<=0.001 &&
    target_percentage(3)<=0.001 && abs(link_speeds(1))<=0.001 && abs(
    link_speeds(2))<=0.001 && abs(link_speeds(3))<=0.001
353         if obj.gripper_status == false && obj.cycle_indicator == false
354           voltage(4)=-3.0;
355           if abs(link_speeds(4))<=0.01 && position_data(4,3)<=0.39625
356             obj.gripper_status=true;
357             obj.cycle_indicator=true;
358           end
359         elseif obj.gripper_status==true && obj.cycle_indicator==false
360           voltage(4)=3.0;
361           if abs(link_speeds(4))<=0.01 && position_data(4,3)>=0.9
362             obj.gripper_status=false;
363             obj.cycle_indicator=true;
364             voltage(4)=0;
365           end
366         end
367       end
368
369     end
370
371     function [set_point,desired_acceleration] = path_planner(obj,theta_f,
    position_data,control_time_step)
372       if obj.theta_f_prev(1)~=theta_f(1) || obj.theta_f_prev(2)~=theta_f
    (2) || obj.theta_f_prev(3)~=theta_f(3)
373         %this calculates trajectory in the case that the controller is
    reset for a new cycle
374         %trajectory parameters are stored as: [a_01,a_21,a_31;a_02,a_22,
    a_32;a_03,a_32,a_33]
375         obj.trajectory_parameters(1,1)=position_data(1);
376         obj.trajectory_parameters(1,2)=3*(theta_f(1)-position_data(1))/obj
    .control_target_time^2;
377         obj.trajectory_parameters(1,3)=-2*(theta_f(1)-position_data(1))/
    obj.control_target_time^3;
378         obj.trajectory_parameters(2,1)=position_data(2);
379         obj.trajectory_parameters(2,2)=3*(theta_f(2)-position_data(2))/obj
    .control_target_time^2;
380         obj.trajectory_parameters(2,3)=-2*(theta_f(2)-position_data(2))/
    obj.control_target_time^3;
381         obj.trajectory_parameters(3,1)=position_data(3);
382         obj.trajectory_parameters(3,2)=3*(theta_f(3)-position_data(3))/obj
    .control_target_time^2;
383         obj.trajectory_parameters(3,3)=-2*(theta_f(3)-position_data(3))/
    obj.control_target_time^3;
384         obj.control_time=0;
385         obj.cycle_indicator=false;
386         obj.theta_f_prev=theta_f;
387       end
388       set_point=zeros(3,1);
389       desired_acceleration=zeros(3,1);
390       if obj.control_time<=obj.control_target_time
391         %in the case the the control time is less than the target, the
    trajectory follows the
392         set_point(1)=obj.trajectory_parameters(1,1)+obj.
    trajectory_parameters(1,2)*obj.control_time^2+obj.trajectory_parameters
    (1,3)*obj.control_time^3;
393         set_point(2)=obj.trajectory_parameters(2,1)+obj.
```

```
   trajectory_parameters(2,2)*obj.control_time^2+obj.trajectory_parameters
   (2,3)*obj.control_time^3;
394        set_point(3)=obj.trajectory_parameters(3,1)+obj.
   trajectory_parameters(3,2)*obj.control_time^2+obj.trajectory_parameters
   (3,3)*obj.control_time^3;
395        desired_acceleration(1)=-2*obj.trajectory_parameters(1,2)+6*obj.
   trajectory_parameters(1,3)*obj.control_time;
396        desired_acceleration(2)=-2*obj.trajectory_parameters(2,2)+6*obj.
   trajectory_parameters(2,3)*obj.control_time;
397        desired_acceleration(3)=-2*obj.trajectory_parameters(3,2)+6*obj.
   trajectory_parameters(3,3)*obj.control_time;
398      else
399        set_point(1)=theta_f(1);
400        set_point(2)=theta_f(2);
401        set_point(3)=theta_f(3);
402        desired_acceleration(1)=0;
403        desired_acceleration(2)=0;
404        desired_acceleration(3)=0;
405      end
406      obj.control_time=obj.control_time+control_time_step;
407
408    end
409
410    function rk4_three_link_arm(obj,input_voltage,time_step,load_force)
411      k1=time_step*obj.eom_three_link_arm(input_voltage,load_force);
412      obj.runge_kutta_k_augment=k1/2;
413      k2=time_step*obj.eom_three_link_arm(input_voltage,load_force);
414      obj.runge_kutta_k_augment=k2/2;
415      k3=time_step*obj.eom_three_link_arm(input_voltage,load_force);
416      obj.runge_kutta_k_augment=k3;
417      k4=time_step*obj.eom_three_link_arm(input_voltage,load_force);
418      obj.runge_kutta_k_augment=zeros(8,1);
419      [~,n]=size(obj.position_speed_history);
420      obj.position_speed_history(:,n+1)=obj.position_speed_history(:,n)+(
   k1+2*k2+2*k3+k4)/6;
421      obj.time_history(n+1)=obj.time_history(n)+time_step;
422    end
423  end
424 end
```