UNIVERSITY OF RIJEKA

FACULTY OF ENGINEERING

Sandi Baressi Šegota

Determining the Energy-Optimal Path of Six-Axis Industrial Robotic Manipulators Using Machine Learning and Memetic Algorithms

DOCTORAL THESIS

Rijeka, 2025.

UNIVERSITY OF RIJEKA

FACULTY OF ENGINEERING

Sandi Baressi Šegota

Determining the Energy-Optimal Path of Six-Axis Industrial Robotic Manipulators Using Machine Learning and Memetic Algorithms

DOCTORAL THESIS

Supervisor: Prof. dr. sc. Zlatan Car

Rijeka, 2025.

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Sandi Baressi Šegota

Određivanje energetski optimalne putanje šestosnih industrijskih robotskih manipulatora primjenom strojnog učenja i memetičkih algoritama

DOKTORSKI RAD

Mentor: prof. dr. sc. Zlatan Car

Rijeka, 2025.

Supervisor: Prof. dr. sc. Zlatan Car

This doctoral thesis was defended on _____ at the Faculty of Engineering, University of Rijeka, in front of the Committee consisting of::

1. _____
2. _____
3. _____

Words of Appreciation

First, I would like to thank my mentor, Prof. dr. sc. Zlatan Car, for his guidance, support, and encouragement throughout my doctoral studies. I am grateful for his pinyatience, understanding, and expertise, which have been invaluable to me.

I would also like to thank my mother, Marisa Baressi, for her unwavering love and support. Her encouragement and belief in me have been a constant source of strength and inspiration. To my late grandfather Mario for his wisdom and guidance, and to my late grandmother Ana for her love and kindness.

I would like to thank my collaborators and colleagues, especially dr.sc. Nikola Anđelić, dr.sc. Vedran Mrzljak, dr.sc. Igor Poljak, and Darin Majnarić for their continuous encouragement and support, and all the helpful discussions we had.

I would like to thank all of the members of the Laboratory for Automation and Robotics for their support and encouragement. I am grateful for the opportunity to work with such a talented and dedicated group of people.

I also must extend my thanks to the members of Astronomical Society "Istra" Pula, especially Damir Šegon, for lightning the spark of scientific curiosity in me all those years ago.

I would like to thank the University of Rijeka for providing me with the opportunity to pursue my doctoral studies. I am grateful for the support and encouragement I have received from the faculty and staff, and I am proud to be a part of this institution.

And finally, to all the absent friends and family, who have supported me in their own way, I am grateful for your love and encouragement. I hope that I have made you proud.

Sažetak — Ovaj rad bavi se energetskom optimizacijom putanja serijskih robotskih manipulatora. Zbog čestog ponavljanja putanja, njihova optimizacija može donijeti značajne uštede. Prvi korak je modeliranje potrošnje energije manipulatora u odnosu na putanju definiranu pozicijama, brzinama i akceleracijama u zglobovskom prostoru. Predstavljena su dva pristupa: klasični analitički model temeljen na Lagrange-Eulerovom algoritmu te metode strojnog učenja. Korišteni su višeslojni perceptron, strojevi potpornih vektora, pasivno-agresivni regresor i gradijentno pojačana stabla. Algoritmi su primijenjeni na tri skupa podataka: stvarni (prikupljen u laboratoriju), simulirani i sintetički (generiran iz stvarnih podataka). Modeli su trenirani višestrukom validacijom i optimizirani pretragom rešetke, a testirani na stvarnim podacima. Rezultati pokazuju da su modeli trenirani na simuliranim podacima slabiji zbog odstupanja od stvarnog industrijskog robotskog manipulatora, dok su sintetički podaci omogućili modele gotovo jednake kao stvarni. Najbolji modeli temelje se na višeslojnom perceptronu i koriste se u optimizacijskom procesu kao dio funkcije pogodnosti. Optimizacija se provodi genetskim algoritmom, pri čemu se najboljom pokazala nasumična rekombinacija gena (95%) uz mutaciju (5%). Algoritam se nadograđuje memetskim pristupom s lokalnim pretraživanjem. Informirano pretraživanje poboljšava putanju u prosjeku za 59% u odnosu na nasumične parametre, što ukazuje na potencijal memetskog pristupa za optimizaciju putanja.

Ključne riječi — energetska optimizacija; evolucijsko računarstvo; industrijski robotski manipulatori; strojno učenje; memetički algoritmi

Abstract — This dissertation focuses on the energy optimization of serial robotic manipulators' trajectories. Due to the frequent repetition of trajectories, their optimization can lead to significant energy savings. The first step is modeling the energy consumption of the manipulator concerning the trajectory defined by positions, velocities, and accelerations in joint space. Two approaches are presented: a classical analytical model based on the Lagrange-Euler algorithm and machine learning methods. The applied algorithms include a multilayer perceptron, support vector machines, a passive-aggressive regressor, and gradient-boosted trees. These algorithms were tested on three datasets: real (collected in a laboratory), simulated, and synthetic (generated from real data). The models were trained using cross-validation and optimized via grid search, then tested on real data. Results indicate that models trained on simulated data perform worse due to deviations from the real IRM, whereas synthetic data allowed for models nearly identical to those trained on real data. The best-performing models were based on a multilayer perceptron and were incorporated into the optimization process as part of the fitness function. Optimization was performed using a genetic algorithm, with the best performance achieved through random gene recombination (95%) and mutation (5%). The algorithm was further enhanced with a memetic approach featuring local search. Informed search improved trajectories by an average of 59% compared to randomly selected parameters, demonstrating the potential of the memetic approach for trajectory energy optimization.

Keywords — energy optimization; evolutionary computing; industrial robotic manipulators; machine learning; memetic algorithms

Prošireni Sažetak — Ovaj doktorski rad se fokusira na problem energetske optimizacije putanja serijskih robotskih manipulatora u načinu rada pokupi i ostavi. Zbog velikog broja manipulatora i čestog ponavljanja putanja, optimizacija može dovesti do značajnih ušteda. Prvi korak ka cilju optimizacije je određivanje modela potrošnje energije robotskog manipulatora u odnosu na putanju zadanu pozicijama, brzinama i akceleracijama u zglobovitom prostoru. Predstavljena su dva pristupa – prvi je klasični analitički pristup temeljen na Lagrange-Eulerovom iterativnom algoritmu za proračun torzije zglobova. Drugi pristup je korištenje metoda strojnog učenja. Primijenjena su četiri algoritma - višeslojni perceptron, strojevi potpornih vektora, pasivno agresivni regresor, te stabla ojačana gradijentom. Navedeni algoritmi se primjenjuju na tri odvojena skupa podataka – stvarni, prikupljen u laboratorijskom okruženju; simuliran, iz računalne simulacije istog robota; te sintetički, generiran na mikro-skupu podataka sa stvarnog robota. Svi modeli su trenirani višestrukom validacijom, te su im hiperparametri određeni procesom pretrage rešetke. Dobiveni modeli su testirani na odvojenom skupu podataka, prikupljenom sa stvarnog robota. Cilj ovoga je ispitati je li moguće razviti modele temeljene na podacima prikupljenima ili simulacijom ili na manjem skupu podataka, kako bi se smanjila invazivnost kod potencijalne primjene na robote korištene u radnim okruženjima. Evaluacijom svih modela se pokazuje da modeli trenirani na simulacijskim podacima daju slabije rezultate u usporedbi s onima treniranim na stvarnim podacima. Ovo je najvjerojatnije uzrokovano razlikama između simuliranog i stvarnog industrijskog robotskog manipulatora, što je potvrđeno statističkom analizom podataka. Modeli trenirani na sintetičkim podacima daju rezultate koji su gotovo jednaki rezultatima modela sa stvarnih podataka. Najbolji modeli su oni bazirani na višeslojnom perceptronu. Stoga se ti modeli koriste u optimizacijskom procesu, kao dio funkcije pogodnosti. Optimizacijski proces se temelji na genetskom algoritmu kod kojega su ispitane različite postavke, te je određeno da najbolje performanse daje genetski algoritam s nasumičnom rekombinacijom gena koja se događa u 95% slučajeva, te mutacijom u 5% slučaja. Ovaj algoritam se nadograđuje memetičkim algoritmom koji

koristi dva načina lokalne pretrage u okolini prostora rješenja najboljeg potencijalnog rješenja na kraju svake generacije – nasumično pretraživanje i informirano pretraživanje temeljeno na analizi utjecaja varijabli na izlaz modela. Informirano pretraživanje polučuje bolje rezultate, s prosječnim poboljšanjem putanje od 59% u odnosu na putanju s nasumično odabranim parametrima. Zaključak doktorskog rada je da ovakav, memetički, pristup potencijalno vrlo koristan za energetsku optimizaciju putanja.

Contents

CHAPTER 1

Introduction

Industrial robotic manipulators (IRMs) are programmable robotic manipulators, used in industry to perform repetitive and complex tasks for which manipulators with lower capabilities would not be satisfactory. According to the European Commission energy directive, reducing energy consumption to achieve energy savings is one of the key steps in delivering the goals of European Green Deal [1]. There are many different areas and sectors which use large amounts of energy, but one of the larger ones is industry. According to European Commission, according to data collected in 2021, industry as a sector used 25.6% of the total energy consumption in the European Union [2]. With the growing automation and robotization of the industry, the energy consumption of industrial robots is not to be neglected. IRMs have a large application in the modern automation of manufacturing processes, due to a high applicability, flexibility and adaptivity [11, 57]. According to the International Federation of Robotics (IFR) in 2023, the record of 553,052 robots were newly installed – approximately 82,958 of which were installed in Europe for a growth of more than 5%. The projected growth in 2023 is expected to be 7% or 590,000 new units worldwide. Based on the same source, it is approximated that the number of industrial robotic units in Europe is around 650,000 units [3]. With the IRMs easily reaching the power consumption of 5kW, assuming a total of 1,000 work hours a year, the power consumption at the level of European Union, just

[1] European Commission – Energy, https://energy.ec.europa.eu/, last accessed March 14th 2025.

[2] Eurostat – Energy statistics, https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Energy_statistics_-_an_overview, last accessed March 14th 2025.

[3] International Federation of Robotics – World Robotics 2023, https://ifr.org/ifr-press-releases/news/world-robotics-2023-report-asia-ahead-of-europe-and-the-americas, last accessed March 14th 2025.

for the industrial robots without any of the supporting infrastructure and equipment, can be approximated at over 15 TWh.

The high power use of industrial robotic systems has not gone unnoticed by the researchers. Sihag and Sangwan note that the tuning of machines for energy efficiency has become an integral part in planning of industrial processes with the goal of economic and environmental performance improvements, because machine tools are a major energy consumer with a very low efficiency and a dynamic energy consumption behavior [67]. Chutima notes how the energy consumption of IRMs, mainly six degree-of-freedom (DOF) robotic arms, is by far the highest energy consumption element of all elements of robotic assembly lines [17]. Swanborn and Malavolta state that one of the biggest source of energy innefficiency in industrial robotic environments is the inneficient movement of the IRMs. They also note a large interest within the research community regarding the focus on optimization of industrial robots, despite them having a constant power supply compared to mobile robots – despite functional unimportance of energy consumption [69]. According to previously mentioned research, this corroborates the fact that energy use of IRMs has a high economical and environmental impact. Lakshmi Srinivas and Jabed note that there are three ways of optimizing the energy use of IRMs – optimizing the topology of the production space, use of lightweight components and trajectory optimization [68]. Topology optimization is limited in two ways. First, it cannot easily be applied to existing solutions due to the need to rearrange the production space which requires significant ti me investment requiring the production to be paused for a longer length of time, and potentially financial investment as well, depending on the environment. In addition to that, the topology of the production environment is usually optimized for the speed of production, and most factories would be unwilling to sacrifice the optimization in the area of production speed to preserve energy. Similar issues plague the use of lightweight robot parts – sometimes light end-effectors can be designed, but replacing entire IRMs can be far from economically viable, and can only be considered when the manipulator currently in use has reached the end of it's lifecycle. Additionally, the lightweight components are

not necessarily applicable in many cases, as the production processes may need more robust components. This leaves the trajectory optimization as the only viable solution for applications on existing robotic manipulators, where the modification of the tools or environment is not possible.

Based on the above research, the goal of this doctoral dissertation is to establish a framework for robot trajectory optimization. Further sections will provide a deeper overview of the techniques and results applied by the researchers in the past.

1.1 Overview of existing research

This section provides a look into the state-of-the-art of the research in optimization of IRMs. The review will start with comparing and contrasting the path optimization techniques which were used by the authors of published work in the field, followed by reviews into other elements of the research.

1.1.1 Optimization of robotic manipulators with regards to energy and related quantities

There are multiple papers that indicate the possibility of tuning the path of a robotic arm with the goal of optimizing the energy expenditure. Vysockyy et al. [73] present the application of a particle swarm optimization (PSO) algorithm. The main goal of the researchers is to execute a point-to-point robot path optimization. Instead of optimizing the whole path of the robotic manipulator, the authors instead focus on optimizing just non-technological movements, or in other words, the parts of the movement where the robot is not actively performing an action related to the industrial, technical process it is performing. The authors apply a PSO algorithm to generate the parameters which are then used for the generation of a Bezier curve which is used as a path. The authors verify the results on the UR3 industrial robotic arm and demonstrate that energy savings range between 10 and 40%, when the non-technological paths are considered. Based on the presented research it can be concluded that additional savings can be achieved

using a similar approach but applying it to the technological parts of the industrial processes as well, if the process in question allows for it. Garriz and Domingo [22] optimize the process of applying sealant in automotive industry. Three separate tasks of sealant application with an IRM are observed by the researchers, and a Kalman algorithm is applied on the simulated paths. The authors limit the results of the algorithm to ensure that the paths taken will results in a sealant application of satisfying quality. The predicted energy for the robot path is then compared between the original path and the planned path, showing that the path can be optimized with regards to the energy, obtaining an optimization of up to 20% depending on the trajectory, given the limitations of the environment, but at the cost of increased operation time. The authors note that the increased operation time is not necessarily good, depending on the optimization and note that possible savings should be weighed against the lower production values – especially in an assembly line environment where the following tasks are influenced by the increased time. A solution that achieves similar results, without increasing the operation time, would be more applicable to a wider range of scenarios. Shrivastava and Dalla [66] illustrate another use of genetic algorithm (GA) for the energy optimization of a multi-axes manipulator. The authors base their research on simulation obtained kinematic and dynamic data based on the bond graph technique. The optimization is performed using a classic recombination GA, which uses the positions of the joints in the tool space as the tuned parameters which define the trajectory of the robot. In the same manner as the authors in [22], Shristava and Dalla propose a system for compliance using realistic robot limits, but also demonstrate that the GA can be tuned with different limits in an automated manner in case the generated path is not viable. The authors achieve an improvement of 38% compared to direct path planning. Nonoyama et al. [54] also apply the tuning of trajectories with the goal of increasing the energy efficiency. The authors demonstrate the use of K-ROSET simulation environment for the application of GA, as that can significantly lower the optimization time and measurement time in comparison to using an experimental setup. The authors apply PSO and GA algorithms, with the goal of tuning the parameters of the Proportional - Integral

- Derivative (PID) controller. The simulated movement of the robot arm is verified using a real robot. The authors show that, for a real robot, the GA algorithm is less computationally expensive and shows better results, improving the energy consumption by 18%, compared to the untuned trajectory. The main limitation of the presented work is that it is performed for a SCARA robot which is a configuration used significantly less compared to the articulated IRMs, in addition to tuning PID controller parameters which may not necessarily be easy to adjust for most robots in comparison to trajectory adjustments. While not focused on IRMs, but instead hexapod robots, some conclusions can be drawn from Luneckas et al. [46], who applied heuristic algorithms to lower the energy consumption during movement. The authors focus on the application of red fox algorithm to adjust the gait switching behavior of the hexapod robot. The authors show that an improvement in energy consumption of up to 21% can be achieved with the application of this type of algorithm, compared to other heuristic algorithms. Based on the demonstrated information, it can be concluded that the application of more advanced algorithms, compared to the classical GA and PSO algorithms can show improvements in the optimization performance, and should be tested if possible.

Lu et al. [45] show the application of a memetic algorithm (MA) for the energy optimization of an IRM tuned for the application in a collaborative welding process. The authors applied a MA, which is set to combine two algorithms in two stages – a GA as the first stage algorithm for wide search of the solutions space and variable neighborhood search (VNS) to search the immediate neighborhood of the solution found by the GA. Due to the complex and highly-variable search spaces of the area of application, the authors propose that searching the immediate area can result in finding a solution with a lower energy use. It is shown that the application of the second stage can improve the optimization results by up to 10%. Still, the authors do not test the performance of different algorithm types for the first and second stage, despite some research demonstrating that different evolutionary algorithms, such as differential evolution (DE) can provide significantly better results [8]. The overview of the best achieved results, expressed as the improvement between optimized and unoptimized paths has been given

in Table 1.1.

Table 1.1: The best results from the reviewed research focused on application of evolutionary computing algorithms on robot energy efficiency improvement. Improvement between the unoptimized and optimized paths is expressed as percentage and rounded to closest value.

Reference	Approach	Improvement [%]
Vysocky et al. [73]	GA, PSO, Bezier curves	40
Garriz and Domingo [22]	Kalman	20
Shrivastava and Dalla [66]	GA	38
Nonoyama et al. [54]	K-ROSET, GA, PSO	18
Luneckas et al. [46]	Red Fox	21
Lu et al. [45]	MA – GA+VNS	10

1.1.2 Energy modeling of robotic manipulators

Gadaleta et al. [19] demonstrate a creation of an experimental setup for energy optimization of IRMs. The goal of the paper is to demonstrate the possibility of creating an experimental dataset, that can be analyzed with the goal of optimizing the energy use. The authors note the importance of creating a robust framework for energy savings during the operation of IRMs. Through the analysis of this dataset, based on experiments performed in a test cell in which a KUKA KR210 R2700 Prime IRM performed different operations on various loads, the authors conclude that base level optimizations in operation, especially in temperature control, can be utilized to achieve large optimizations – up to 50% of energy can be saved. In addition to that, the authors note that further optimizations, such as path planning level optimizations can be used to save even more energy during the operation of the robot. This paper demonstrates the need for creation of well-rounded datasets and their analysis to form conclusions as to which variables have the greatest influence on the energy use. Zhang and Yan [77] demonstrate an example of a direct applications of artificial neural networks (ANNs) for the purposes of energy consumption prediction. The authors collect a dataset which uses different

velocities (5, 10, and 15% of the maximum velocity) and 25 or 50% of maximum acceleration to perform three cycles of operation on the Epson C4 IRM. The prediction is then performed using the ANN, with velocity and acceleration as inputs. The authors demonstrate that the R-value of the data can reach 0.9699 using the methodology on the predicted dataset. The presented work could be significantly improved using more modern approaches to modeling. Additionally, the lack of additional procedures such as cross-validation (CV) along with the presented data shapes indicate a possible overfitting issue. Gao et al. [20] apply a Long-short term memory (LSTM) ANN for the prediction of a planar parallel manipulator (PPM). Their approach is based on torque optimization. The dataset consists of the change of total torque of the robotic manipulator in time, as calculated based on the Newton-Euler method for dynamics calculation of IRMs. The authors achieve the root mean squared error (RMSE) error of $41 \cdot 10^{-6}$ and mean absolute percentage error (MAPE) of 2.5%. Lin et al. [38] also apply a type of LSTM to the problem of predicting the energy consumption of the robot, opting for a BN (batch-normalized) LSTM. The authors compare the results and the modeling on 64,600 data points of a public dataset collected on the Yaskawa GP7 dataset, achieving the best RMSE score of 3.67 – a 23% improvement compared to state-of-the-art results on the dataset. LSTM is applied yet again, as a basis of a custom deep ANN applied by Jiang et al. [35], on a dataset collected on a KUKA KR60-3 IRM, measuring the energy use of the robot in time. The authors achieve a MAPE score of 4.21% on the collected dataset using the described methodology. While the application of LSTMs seems to be extremelly popular for energy prediction, it has its shortcomings. Jaramillo-Morales et al. [34] demonstrate the application of a direct numeric model for the prediction of the power use in a differential drive robot. The authors note a significant benefit of using a model which can estimate the momentary energy use in comparison to a time-series model, such as the possibility of using such models in rapidly changing environments or for optimization purposes. The authors perform the modeling of Nomad Super Scout II robot, but their results are lower compared to the ML-based approaches, with accuracy falling as low as 81.25% along curved trajectories. From the reviewed papers, it

is easy to conclude that an ML-based model, developed for a prediction of momentary energy use of an industrial manipulator could be applied in much the same way, while achieving significantly higher scores. As before, the results achieved in the reviewed research are provided in Table 1.2.

Table 1.2: The best results from reviewed research focused on the energy use modeling of robots. RMSE – Root Mean Squared Error, MAPE – Mean Absolute Percentile Error.

Reference	Approach	Score
Zhang et al. [77]	ANN	R-score = 0.97
Gao et al. [20]	LSTM	MAPE = 2.5%
Lin et al. [38]	BN-LSTM	RMSE = 3.67
Jiang et al. [35]	LSTM	MAPE = 4.24%
Jaramillo et al. [34]	Numeric	Accuracy = 82.15%

1.1.3 Synthetic data application in robot modeling using machine learning

Synthetic data can come from two main sources. The first is the use of simulations (also sometimes referred to as digital twins [72]) to simulate the real process in using the so-called *in-silico* process, while the second is using statistical methods for the creation of data which follows similar trends and statistics as the original sets of data [49]. In this doctoral thesis, the first type of the data (synthetic data sourced from simulations) will be referred to as “simulation” data, while the second type (synthetic data created with a statistical method) will be referred to as “synthetic” data. Application of simulation data as a data source for ML in robotics is extremely common practice in research – laboratory setups with IRMs can be expensive, and the IRMs installed in the industrial environments are commonly reserved for the industrial process, and they might not be available for data-collection purposes if the measurements cannot be performed during the operation [16]. Kleeberger et al [36] provide a survey on collection of data using simulated environments, with the goal of learning-based robotic grasping. The authors note how the use of simulation data allows for easier tuning and adjustment

of grapsing approaches, allowing for an iterative approach which increases the final precision of models. Another application is noted by Osinski et al. [56], who discuss the benefits (safety and precision) of using a simulated environement for the tuning of autonomous driving robots. Still, as noted by Zhao et al. [78] the models developed on simulation data must be validated in a real-world environment – either directly or using the data collected in a real-world environment, as pure simulation data in both training and validation can lead to exacerbation of errors introduced by the simulation, and generally have a smaller variation due to ignoring a number of elements of real-world data such as noisiness. Synthetic data on the other hand has been discussed at length by a number of researchers. Some of the examples related to electrical energy consumption modeling include Haghi et al. [25] who utilized synthetic data regression based on the Gaussian process to determine the energy-load of a wind turbine, with the authors pointing out how the use of hybrid data allows for a great expansion of the dataset. Rubio et al. [21] apply synthetic data generation for electrical energy consumption. This approach allows them to obtain a well performing model despite that the data they are modeling has different ranges of data. The authors note that the approach used, based on trace generation, generally performs well but has issues when the anomalous situations are attempted to be predicted based on the synthetic data. This is expected behaviour, as the synthetic methods commonly generalize the data in an attempt to model it. In the area of robotics synthetic data has not been widely applied and tested for industrial robots, despite showing promise in other areas – such as a creation of odometrical datasets consisting of inertial data for a mobile robot, as demonstrated by Schofield et al. [64]. Schoettler et al. [63] apply synthetic data for meta-reinforcement learning of IRM insertion tasks. This approach allows for the improvement of precision, through testing in environments that were not available for testing in a laboratory environment. Despite showing promise in other areas of robotics and the tasks such as energy consumption models, synthetic data of this type has not been applied in the modeling of energy efficiency for IRMs. One of the only researches published regarding the optimization based in part on synthetic data is by Barenji et

al. [7]. The authors in this paper discuss the possibility of using digital twins, or in other words applying the models developed in a simulated environment with the final goal of energy use optimization. The authors propose using a simulated environment for the purpose of both simplifying data collection in realistic setups, as well as expanding the datasets with more extreme setups. The model is created using SolidWorks computer aided design (CAD) model, along with the MATLAB software for calculating the energy use. The authors then apply a genetic algorithm for the tuning process of movement and manage to lower the energy consumption by a minimum of 4.8%, maximum of 11.0%. When applied to the real-world robot, this translates to the minimum saving of 3.8% and the maximum of 7.7%, indicating that optimizations performed on the created digital twin can translate into real-world optimization. The research shows how the simulated environments can be applied in the field of robotics to obtain valid data and test the procedures. The question arises if this approach can be repeated using a less complex system than full digital twin of the robot, or simply by using statistical data synthetization avoiding the process of simulation completely.

1.1.4 Identification of knowledge gap and setting hypotheses

Observing the EC application there are a few notable gaps in the researched areas. The main one is that most of the developed methods significantly increase the time of the trajectory. In an industrial manufacturing environment this is often times unnaceptable, as even a small increase in the time needed for a single operation can cause large delays on a daily or monthly basis. The methods that do not increase the time have a certain optimization restraint, such as only optimizing a part of the trajectory. When MA is applied to a similar problem, the performance is good – but there are a lot of unexplored configurations which could serve to improve the energy efficiency of a trajectory significantly.

Problem definition

In this a dataset will be collected using three different approaches – synthetized data, simulation-based data and real-world collected data. Through testing the performance of models developed in other parts of the presented work, it will be determined which of these approaches is more viable, and can the collection of data on real IRMs be replaced with a significantly simpler approaches. By implementing an approach that requires much shorter stopping of production to obtain measurements, the proposed solution is more viable for application. In addition to the research in comparing the datasets, presented research will aim at development of regression-based model for prediction of electrical energy being used by the operation of IRM at any given moment. Compared to time-series based approaches, the regression model will have wider applicability, especially for the problem of optimization. Finally, the viability of applying MA algorithm to the problem of energy efficiency optimization for the IRM trajectories will be tested. Different algorithms will be described and tested for both stages of the algorithm, to conclude which approach provides the best optimization results. The developed MA will consist of the regression ML-based model used as a fitness function, making this a novel hybrid approach to the development of MA-based optimization systems. Based on the above review the goals of the research described in this doctoral thesis are as follows:

- Develop a methodology for collection and creation of appropriate datasets for application of modeling the energy optimal path of a six-axis IRM,
- Determine a new model of energy use for a six-axis IRM based on the collected datasets,
- Develop an algorithm based on memetic principles and the aforementioned energy use model for determining an optimal path with regard to the energy use.

Scientific hypotheses

Based on the defined problem and the goals of this doctoral thesis, the following research hypotheses can be defined:

- The ML-based, data-driven, models achieve higher performance and are faster in determining the energy use of a six-axis IRM compared to classic mathematical models.
- Synthetic data can be generated models which can have scores as high as the models trained on real-world data.
- The application of an optimization algorithm based on a memetic principle can optimize paths better than the algorithms based just on the evolutionary principles.

The thesis consists of four parts, each aiming to describe the total scientific contribution of this doctoral thesis. In the first part, "Methodology", the focus will be given on describing the developed method for determining the energy consumption of an IRM, based on a data-driven method. For comparison, the conventional, analytical, approach will be described, allowing for better illustration and comparison of strengths of data-driven approaches. This part will also describe the idea behind the newly developed optimization algorithm. This, MA-based algorithm will use the aforementioned data-driven method as the fitness function, with the main goal of lowering energy consumption. "Dataset collection and analysis", will serve to define the third scientific contribution – methodology for the collection and creation of datasets that can be used for the development of data-driven methods. In "Results and discussion" the scientific contributions will be fully realized and presented through the presentation of individual results, and comparison of tested approaches to determine the final proposal of the system for energy optimization of six DOF IRMs. "Conclusions" will serve to summarize the presented work and its scientific contributions, along with a discussion of possible future research directions.

CHAPTER 2

Methodology

2.1 Mathematical model of industrial robotic manipulator

Energy consumption in robotic systems can often be evaluated using the product of torque and joint velocity. The torque, $M(q, \dot{q}, \ddot{q})$, is a function of the joint angle q, angular velocity $\dot{q}$, and angular acceleration $\ddot{q}$. The relationship between these variables encapsulates the dynamic and inertial properties of the robotic system, as well as external forces. Energy can then be determined by integrating the product of torque and angular velocity over time:

$$E = \int M(q, \dot{q}, \ddot{q}) \times \dot{q}\, dt. \tag{2.1}$$

Here, E represents the total energy consumed by the robotic joint over the time interval of interest. The torque T accounts for contributions from inertial forces, Coriolis and centrifugal effects, gravitational forces, and external load torques. These components are typically modeled using equations derived from the Lagrangian or Newton-Euler formulations [75]. The term $\dot{q}$, representing the angular velocity, is critical because energy consumption is proportional to the rate at which the joint moves. This implies that both high torque and high speeds significantly increase energy usage. For practical applications, understanding this relationship enables the optimization of energy efficiency in robotic systems, particularly for tasks involving repetitive or high-speed motions [5]. By substituting specific dynamic equations into $M(q, \dot{q}, \ddot{q})$, the energy model can be tailored to different robotic designs. This flexibility allows researchers to adapt the model to various use cases.

For discrete values of m and $\dot{q}$, the integral form of energy calculation is replaced by a summation. This is because, in discrete systems, the continuous variables are sampled at specific time intervals, resulting in a sequence of discrete values. The energy calculation then becomes:

$$E = \sum_{i=1}^{N} M(q_i, \dot{q}_i, \ddot{q}_i) \times \dot{q}_i \cdot \Delta t,$$

where:

- $M(q_i, \dot{q}_i, \ddot{q}_i)$ is the torque at the i-th time step,
- $\dot{q}_i$ is the angular velocity at the i-th time step,
- Δt is the time interval between consecutive data points, and
- N is the total number of data points.

In this form, the energy computation is practical for numerical implementation in simulations or real-time control systems where data is discretized. The accuracy of the calculation depends on the sampling frequency, with smaller Δt providing a closer approximation to the continuous integral.

2.1.1 Torque calculation

Inertia Tensors and Centers of Mass

Both algorithms require the use of three values dependent on the robotic manipulator itself: the link inertia tensors, link masses, and centers of mass of the links. The inertia tensor of link k is defined based on the mass of link m_k, the linear velocity of the center of mass of link k relative to the base coordinate system $L_0 - v^k$, the angular velocity of the center of mass of link k relative to $L_0 - \omega_k$, and the center of mass of link k relative to L_0. These elements are illustrated in Figure [55] 2.1.

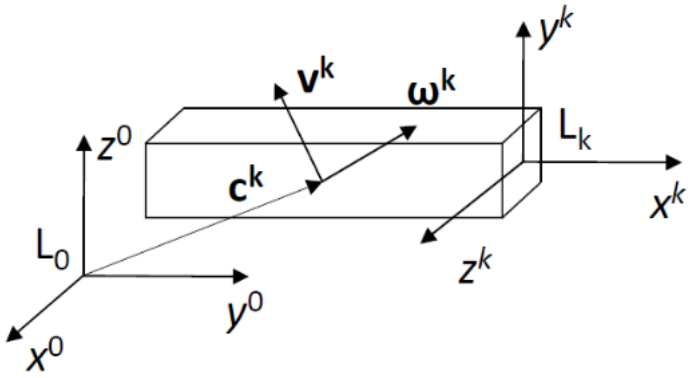


Figure 2.1: Elements of the inertia tensor (L_0 – coordinate system origin (base of the industrial robotic manipulator), L_k – coordinate system at the end of link k, c^k – center of mass of link k, ω^k – angular velocity of link k, v^k – linear velocity of link k) [6].

To define the inertia tensor, it is necessary to define the angular momentum. If m_i represents the mass of a particle moving with angular velocity ω_i and linear velocity v_i, the angular momentum is defined as the vector product of the particle's distance from the center of mass r_i' and its linear momentum p_i', as per [55]:

$$L_i = r_i' \times p_i' = r_i' \times m_i v_i' = r_i' \times m_i(\omega \times r_i'). \tag{2.2}$$

Since a body consists of multiple individual particles, these must be summed. If we assume that the body consists of n particles, the total angular momentum can be defined as [55]:

$$L = \sum_{i=1}^{n} L_i = \sum_{i=1}^{n} r_i' \times m_i(\omega \times r_i') = \sum_{i=1}^{n} m_i r_i'^2 \omega. \tag{2.3}$$

The parameter $\sum_{i=1}^{n} m_i r_i'^2$ defines the body's moment of inertia J. Assuming the number of particles is infinitesimal, the angular momentum can be defined as [55]:

$$L = \int_m (r' \times v')dm. \tag{2.4}$$

Using the homogeneity rule of vector products, the above expression can be rewritten as [55]:

$$L = \int_m (r' \times v')dm = \int_m r' \times (\omega \times r')dm = \int_m [(r'r')\omega - (r'\omega)r']dm. \tag{2.5}$$

If the parameters r' and ω are resolved into components as $r' = x'i + y'j + z'k$ and $\omega = \omega_x i + \omega_y j + \omega_z k$, and these are substituted into the angular momentum expression, we obtain:

$$\begin{aligned} L =&(\omega_x \int_m (y'^2 + z'^2)dm - \omega_y \int_m y'x'dm - \omega_z \int_m z'x'dm)i \\ &+(-\omega_x \int_m y'x'dm + \omega_y \int_m (x'^2 + z'^2)dm - \omega_z \int_m y'z'dm)j \\ &+(-\omega_x \int_m x'z'dm - \omega_y \int_m y'z'dm + \omega_z \int_m (x'^2 + y'^2)dm)k \end{aligned} \tag{2.6}$$

This expression can be simplified as:

$$\begin{aligned} L =&(I_{xx}\omega_x - I_{xy}\omega_y - I_{xz}\omega_z)i \\ &+(-I_{yx}\omega_x + I_{yy}\omega_y - I_{yz}\omega_z)j \\ &+(-I_{zx}\omega_x - I_{zy}\omega_y + I_{zz}\omega_z)k, \end{aligned} \tag{2.7}$$

which can be defined as the product of the inertia tensor matrix D and the angular velocity vector ω [55]:

$$L = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = D\omega \tag{2.8}$$

Obtaining Values

Since robotic manipulators consist of links with complex three-dimensional shapes and non-homogeneous mass distribution within each link, calculating the inertia tensors is complex. To avoid errors and increase the accuracy of the model, the values of inertia tensors, centers of mass, and link masses were obtained using the 3D STEP model of the industrial robotic manipulator ABB IRB 120, downloaded from the manufacturer's website. The model itself is shown in Figure 2.2.

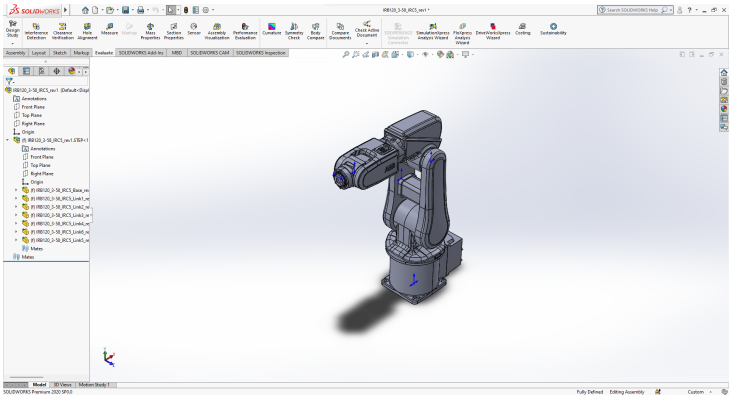


Figure 2.2: Robot model representation in the Solidworks software package.

Kinematics Model

To define dynamic models, it is also necessary to define and model the kinematics model of the industrial robotic manipulator. The kinematics model describes the movement of the robotic manipulator in space. This model defines and allows the transformation from the joint coordinate space to the tool space (i.e., the global coordinate space). In other words, the kinematics model enables calculating the position of the robotic manipulator's end-effector based on the joint angles (forward kinematics) and calculating the joint angles required to place the end-effector in a desired position (inverse kinematics). This model can be defined using the Denavit-Hartenberg method. The goal of the Denavit-Hartenberg method is to determine four kinematic parameters used to construct transformation matrices. These parameters for link k of the robotic manipulator are defined as follows [39]:

- θ_k – joint angle, describing the rotation around the z^{k-1} axis to align the x^{k-1} axis parallel to the x^k axis,
- d_k – joint distance, or the translation along the z^{k-1} axis required to intersect the x^{k-1} and x^k axes,
- a_k – link length, defined as the translation along the x^k axis needed to intersect the z^k axes (this parameter is sometimes denoted as r_k), and

- α_k – link twist, given as the rotation around the x^k axis to align the z^{k-1} axis parallel to the z^k axis.

Figure 2.3 illustrates the parameters and associated axes described above.

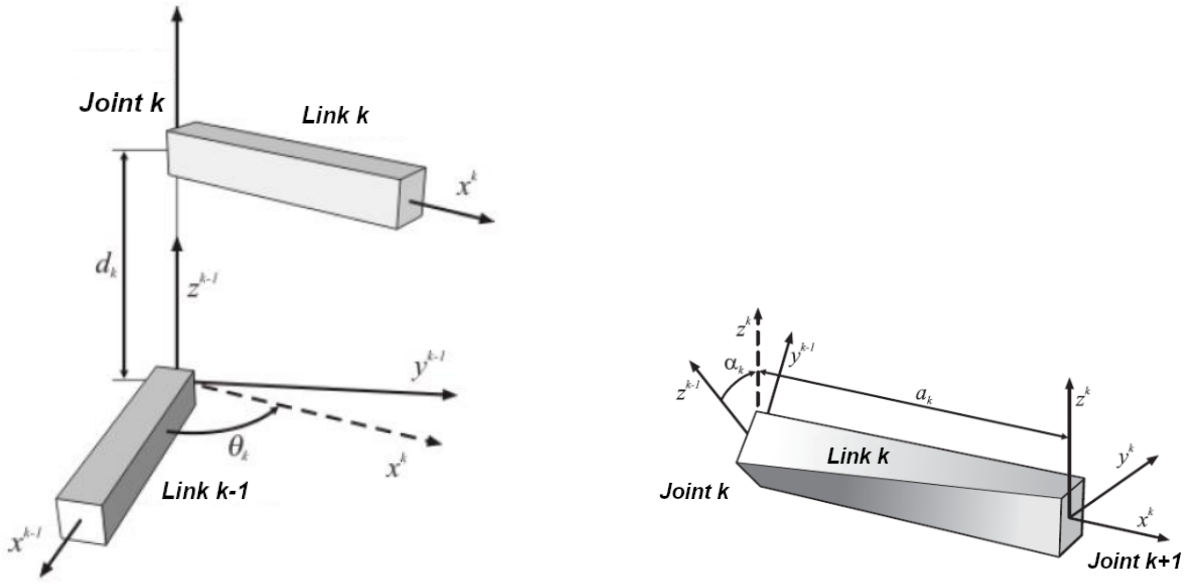


Figure 2.3: Illustration of Denavit-Hartenberg kinematic parameters [6].

The obtained parameters are placed into a transformation matrix defined as [75]:

$$T_{base}^{end} = \begin{bmatrix} R(q) & p(q) \\ v_1^T & \sigma \end{bmatrix}, \tag{2.9}$$

where:

- $R(q) = [r^1\ r^2\ r^3]$ is a 3×3 orientation matrix of the tool, with vectors:
  - r^1 – normal vector,
  - r^2 – sliding/moving vector, and
  - r^3 – approaching vector,
- $p(q)$ is the position of the robotic manipulator's end-effector (or tool) of size 3×1,
- v is the perspective vector, usually $[0\ 0\ 0]$, and

- σ is the scaling coefficient, usually set to 1.

The vectors r^1, r^2, and r^3 are also shown in Figure 2.4.

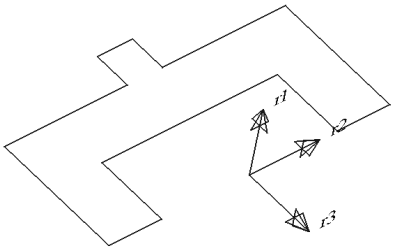


Figure 2.4: Representation of the normal vector (r^1), sliding/moving vector (r^2), and approaching vector (r^3).

The transformation matrix is defined between each two links k and $k-1$, based on the parameters:

$$T_{k-1}^{k} = \begin{bmatrix} cos(\theta_k) & -cos(\alpha_k)sin(\theta_k) & sin(\alpha_k)sin(\theta_k) & a_k cos(\theta_k) \\ sin(\theta_k) & cos(\alpha_k)cos(\theta_k) & -sin(\alpha_k)cos(\theta_k) & a_k sin(\theta_k) \\ 0 & sin(\alpha_k) & cos(\alpha_k) & d_k \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.10}$$

and to obtain the total transformation matrix, the product of all transformation matrices from joint $k = 1$ to the total number of joints is calculated [75]:

$$T_{base}^{end} = \prod_{k=1}^{n} T_{k-1}^{k} = T_0^1 \cdot T_1^2 \cdots T_{n-2}^{n-1} \cdot T_{n-1}^{n}. \tag{2.11}$$

Parameters are determined using an iterative two-part process. Before starting, each joint is numbered from 1 to n, from the manipulator base to the end-effector. At the base, a coordinate system L_0 is placed, and another at the end-effector, L_n, aligning its axes according to the approach, sliding, and normal vectors. The parameters are then determined iteratively, as described, ensuring precise alignment for all kinematic parameters.

After all joints have been assigned coordinate systems, the procedure returns to the first joint, and an additional point necessary for determining the parameter b_k is placed

at the intersection of the axes x^k and z^{k-1}. If the axes x^k and z^{k-1} do not intersect, the point b_k is placed at the intersection of the axis x^k and the common perpendicular to the axes x^k and z^{k-1}. Once the points b_k are determined for each link, the Denavit-Hartenberg parameters can be determined [50]:

- θ_k is the rotation angle around the axis z^{k-1}, from the axis x^{k-1} to the axis x^k,
- d_k is the distance from the origin of the coordinate system L_{k-1} to the point b_k, along the axis z^{k-1},
- a_k is the distance from the point b_k to the origin of L_k, along the axis x^k,
- α_k is the rotation angle around the axis x^k, from the axis z^{k-1} to the axis z^k.

It is important to note that the values of d_k, a_k, and α_k are constants for most configurations of industrial robotic manipulators, while q_k is variable. Once these parameters are determined for each joint, the procedure is complete. An example of the Denavit-Hartenberg method applied to the ABB IRB 120 robotic manipulator is presented below. A simplified robot scheme is shown in Figure 2.5, illustrating the placement of the coordinate systems in joints of the IRM, with the determined parameters provided in Table 2.1.

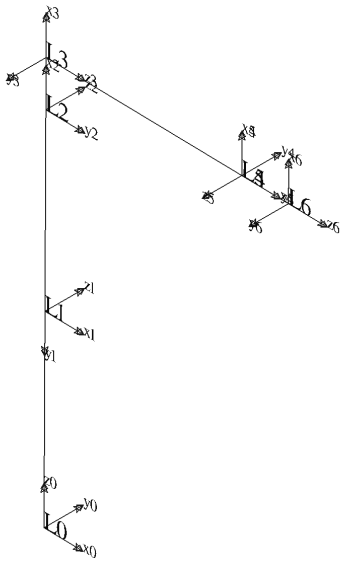


Figure 2.5: Simplified kinematic diagram of the industrial robotic manipulator.

Table 2.1: Calculated Denavit-Hartenberg parameters for the ABB IRB 120 robotic manipulator, with link lengths obtained from [40, 42].

θ **[rad]**	**d [mm]**	**a [mm]**	α **[rad]**
$\theta_1 = q_1$	$d_1 = 290$	$a_1 = 0$	$\alpha_1 = -\frac{\pi}{2}$
$\theta_2 = q_2$	$d_2 = 0$	$a_2 = 270$	$\alpha_2 = 0$
$\theta_3 = q_3$	$d_3 = 0$	$a_3 = 70$	$\alpha_3 = -\frac{\pi}{2}$
$\theta_4 = q_4$	$d_4 = 302$	$a_4 = 0$	$\alpha_4 = \frac{\pi}{2}$
$\theta_5 = q_5$	$d_5 = 0$	$a_5 = 0$	$\alpha_5 = -\frac{\pi}{2}$
$\theta_6 = q_6$	$d_6 = 72$	$a_6 = 0$	$\alpha_6 = 0$

Lagrange-Eulerov dynamics model for serial robotic manipulators

The Lagrange-Euler dynamic model of an industrial robotic manipulator is based on the Lagrange function L, which is defined as [18]:

$$L(q, \dot{q}) = T(q, \dot{q}) - U(q). \tag{2.12}$$

In other words, the model consists of the energy generated during motion (kinetic energy) T and potential energy U, which can be expressed in expanded form as [18]:

$$L(q, \dot{q}) = T(q, \dot{q}) - U(q) = \frac{\sum_{k=1}^{n} \sum_{j=1}^{n} [D_{kj}(q) \dot{q}_k \dot{q}_j]}{2} - \left(- \sum_{k=1}^{n} \sum_{j=1}^{n} g^k m_j c_j^k(q) \right). \tag{2.13}$$

In the above equation, $D_{kj}(q)$ defines the link inertia tensor between joints i and j as a function of the joint position vector q. The potential energy consists of the product of gravitational acceleration g, link mass m, and the center of mass of individual links c relative to joint positions given in q. The Lagrange-Euler model, along with the elements described above, is then defined as [18]:

$$\sum_{j=1}^{n} [D_{ij}(q) q_j] + \sum_{k=1}^{n} \sum_{j=1} C_{kj}^i(q) q_k q_j + h_i(q) + b_i(q) = \tau_i, i \leq i \leq n. \tag{2.14}$$

In the above definition, the term $\sum_{j=1}^{n}[D_{ij}(q)q_j]$ defines inertial forces and torques, while the term $\sum_{j=1} C_{kj}^{i}(q)q_k q_j$ defines Coriolis forces ($k \neq j$) or centrifugal forces ($k = j$). The terms $h_i(q)$ and $b_i(q)$ define the influence of gravity on the manipulator and friction within the manipulator's joints, respectively. The resultant value τ_i defines the torque in the actuator joint, where i represents the manipulator joint, which ranges from 1 to the total number of manipulator joints n.

The process of calculating the Lagrange-Euler model consists of three parts, two of which are iterative.

In the first part, which can be defined as the initial setup, the base transformation matrix $T_0^0 = I$, the iterator $i = 1$, and the tensor $D(q) = 0$ are set. Then, for each link of the robotic manipulator i, the coordinate system L_i, the center of mass relative to the link Δc^i, and the link inertia tensor D_i' are defined. Once all these values are determined, further calculation can begin by entering the first iterative part of the Lagrange-Euler process. This part serves to define the transformation of mass property coordinates based on the kinematic model of the industrial robotic manipulator [75].

The first step is to calculate the joint transformation vector z as:

$$z^{i-1}(q) = R_0^{i-1}(q) \times i^3. \tag{2.15}$$

Additionally, the matrix of composite homogeneous transformation for a given joint is calculated as the product of individual transformation matrices up to the current joint:

$$T_0^i(q) = T_0^{i-1}(q)T_{i-1}^i(q), \tag{2.16}$$

the coordinate of the center of mass of the current link relative to the global coordinate system (assuming the manipulator base is located at the origin of that coordinate system):

$$c^i(q) = H_1 T_0^i(q)\Delta c^i. \tag{2.17}$$

and the link inertia tensor relative to the base coordinate system:

$$D_i(q) = R_0^i(q) D_i' [R_0^i(q)]^T. \tag{2.18}$$

At the end of this section, the Jacobian matrix is defined. The Jacobian matrix allows for the connection between infinitesimal joint displacements and infinitesimal tool displacements (both angular and linear), as:

$$J^k(q) = \begin{bmatrix} A^k(q) \\ B^k(q) \end{bmatrix} = \begin{bmatrix} \frac{\partial c^k(q)}{\partial q_1} & \cdots & \frac{\partial c^k(q)}{\partial q_k} & 0 & \cdots & 0 \\ \xi_1 z^0(q) & 0 & \xi_k z^{k-1}(q) & 0 & \cdots & 0 \end{bmatrix}, \tag{2.19}$$

where ξ represents the joint type – $\xi = 1$ for a rotary joint, and $\xi = 0$ for a linear joint. Since the modeled robot, like most modern industrial robotic manipulators, uses only rotary joints, this value will mostly be set to 1. In the final step of the first part, the manipulator inertia tensor up to the current joint $D(q)$ is calculated:

$$D(q) = \sum_{k=1}^{n} [A^k(q)]^T m_k A^k(q) + [B^k(q)]^T D_k(q) B^k(q). \tag{2.20}$$

After calculating this element, the counter i is incremented by one. If the counter is less than the number of robotic manipulator joints n, the process returns to the beginning of this section, and the calculation is repeated for the remaining joints. If the counter equals n, the procedure proceeds to the second iterative process, resetting the counter value i to 1.

In the second iterative part, the first step is to calculate the velocity coupling matrix for the i-th joint:

$$C_{kj}^i = \frac{\partial D_{ij}(q)}{\partial q_k} - \frac{1}{2} \frac{\partial D_{kj}(q)}{\partial q_i}, \tag{2.21}$$

along with the gravity vector:

$$h_i(q) = \sum_{k=1}^{3} \sum_{j=1}^{n} [g^k m_j A_{ki}^j(q)]. \tag{2.22}$$

The final element of the total torque is the friction within the joint. The model used for modeling friction in a robot is the Elasto-Plastic (EP) friction model, which was selected

due to previous research into the application of friction models on IRMs, as it has shown to be the best performing amongst the other tested models [9]. The EP friction model is based on the so-called bristle model, which simulates microscopic contacts between two surfaces, where each contact is considered a flexible interaction between two elements. Assuming that each of these elements is elastic, each element acts like a spring, resisting the relative displacement of the surfaces. The EP model separates friction into two components: one where the aforementioned springs behave like elastic bodies and another where they behave like plastic bodies. The model incorporates two coefficients: μ_s, the static friction coefficient, and μ_k, the kinetic friction coefficient. Depending on the velocity v, the model is defined as [32]:

$$F_{EP} = \begin{cases} 0, & v \to 0, \\ -\mu_s \cdot v/|v|, & v < \mu_s, \\ -\mu_k \cdot \mathrm{sgn}(v), & v > \mu_s. \end{cases}$$

For simpler implementation, especially since we want to obtain a single equation we can export as a model for later use in optimization, the Elasto-Plastic model can also be expressed in a single equation as [32]:

$$F_{EP} = -\mu_k \cdot \mathrm{sgn}(v) \cdot \left(1 - \frac{e^{-|v|-\mu_s}}{\mu_k}\right).$$

The parameters of the model determining based on the existing research in the area, and the manufacturer provided approximations at $\mu_k = 0.504$ and $\mu_s = 0.302$ [40, 42] This allows the Lagrange-Euler equation to be defined as [75, 18]:

$$\tau_i = \sum_{j=1}^{n} [D_{ij}(q)\ddot{q}_j] + \sum_{k=1}^{3} \sum_{j=1}^{n} [C^i_{kj}(q)\dot{q}_k\dot{q}_j] + h_i(q) + b_i(q, \dot{q}). \tag{2.23}$$

In the above equation, the term $b_i(q, \dot{q})$ represents friction calculated according to the friction model. The specific friction models used in this work are defined later and are incorporated into the calculations. Once this value is computed, the process proceeds to the calculation for the next joint ($i \leftarrow i + 1$). If the values have been calculated for all joints, the computation is complete.

Industrial robot markup language for notation of robot properties

While the ML algorithms that will be described below can easily be modified for different IRMs through substitution of training data and re-training of models, the described LE algorithm requires adjustment of many individual variables to allow for the tuning. The algorithm, as realized into Python is given in Appendix B. It can be noted that the code for loading parameters for calculation reads those from a file, robot.irml. This is a custom XML-based format developed for this application, with the goal of storing all of the information describing the robot and its kinematic and dynamic properties in one document.

The document is structured as a tree split into two parts. First part contains direct information about the robot - such as robot name, manufacturer, reach, capacity and other elements. Each of these metrics are given between descriptive tags, as shown in the Listing 2.1.

```
<robot>
    <!--General information-->
    <name>IRB 120</name>
    <manufacturer>ABB</manufacturer>
    <carrying-capacity>3.0</carrying-capacity>
    <reach>0.6</reach>
    <configuration>Articulated</configuration>
    <ir-type>Serial</ir-type>
    <manipulator-mass>25</manipulator-mass>
    ...
</robot>
```

Listing 2.1: Base information about the robot as written in IRML.

Beyond the general information about the robot, individual information about each joint of the robotic manipulator is given. The document can handle any number of joints, and it contains the following information: joint ID – which is a unique ID, identifying joint, usually a number of the joint when counted from the base, joint type – 'T', 'R', or 'L' indicating a torsion joint (twisting motion around an axis), rotational joint (angular

motion around an axis), or linear joint (straight line movement along an axis), respectively; xi (ξ) – a numerical value indicating whether the joint is revolute (in which case the value is 1) or translational (in which case the value is 0). This is then followed by a subset of kinematics values, which contain the four kinematic parameters mentioned previously when discussing the D-H procedure – joint angle q, distances d and a (here expressed as "r" instead of "a" to avoid confusion with the following variable), and the angle α – expressed as 'a' within the IRML. This values are a minimal set of values which allow for the recalculation of kinematic transformation matrices. Finally, the properties related to dynamics of the link are given, including the mass of the link, center of mass (given as x, y and z), and tensor of inertia (as used in equation 2.18, with each individual element given separately. The example of this for the first joint of ABB IRB 120 is given in Listing 2.2

```
<joint ID="1">
    <type>T</type>
    <xi>1</xi>
    <kinematics>
        <q>q1</q>
        <d>0.29</d>
        <r>0</r>
        <a>-pi/2</a>
    </kinematics>
    <dynamics>
        <link-mass>3.06700626</link-mass>
        <center>
            <x>0.00009765</x>
            <y>0.23841163</y>
            <z>0.00011925</z>
        </center>
        <inertia>
            <xx>-0.00615871</xx>
            <xy>0.99996896</xy>
            <xz>-0.00491487</xz>
            <yx>-0.99767761</yx>
```

```
22                  <yy>-0.00647786</yy>
23                  <yz>-0.06780436</yz>
24                  <zx>-0.06783409</zx>
25                  <zy>0.00448587</zy>
26                  <zz>0.99768653</zz>
27              </inertia>
28          </dynamics>
29      </joint>
```

Listing 2.2: Information about the first joint of the ABB IRB 120 as written in IRML.

The full example of the IRML document for the ABB IRB 120 is given in Appendix C. IRML documents such as this can be created for different robots and used as an input for the calculation of dynamic properties, or other similar calculations. The format of the IRML document is such that it can be easily modified and expanded to fit different types of industrial robots, simplifying the usage and application of previously described methodology.

2.1.2 Speed interpolation

As sown in the previous chapter, the elements necessary to determine the torque of individual robot joints are the intrinsic values connected to the IRM such as link dimensions, mass, centres of mass, and tensors of inertia and the position, speed and accelerations. While the first of these are defined by the selected robot, the positions, speeds and accelerations need to be determined through a path planning technique. While there are multiple techniques that can be used to determine the movement between two points, the selected technique needs to be capable of satisfying the following conditions:

1. the internal parameters need to be easily adjustable, to allow for easier tuning of parameters using optimization algorithms,

2. the complexity of the techniques needs to be low, avoiding iterative parts, in order to not increase the algorithm complexity, as new paths will need to be calculated

within it multiple times,

3. determining the speed and acceleration from the technique needs to be simple, avoiding complex calculations to determine it from positions, and

4. it has to allow the change of the path without lengthening the time of the trajectory.

In other words, the selected interpolation technique needs to allow for easy tuning via numeric parameters and easy calculation of derivatives. Out of the common path planning techniques for industrial robots [70], the method that satisfies this is polynomial interpolation [23].
The polynomial interpolation works by calculating the positions between them using a polynomial of a general shape:

$$f(x) = a_1 \cdot x^n + a_2 \cdot x^{n-1} + \cdots + a_m \cdot x^0, \tag{2.24}$$

where the coefficients $a_i, i \in [1, m]$ are the adjustable parameters that will dictate the shape of the interpolated path. In this instance, the selected order of the polynomial for interpolating the path is five, resulting in the equation for joint position (in the joint space, expressed in radians):

$$q = a_1 \cdot t^5 + a_2 \cdot t^4 + a_3 \cdot t^3 + a_4 \cdot t^2 + a_5 \cdot t^1 + a_6 \cdot t^0, \tag{2.25}$$

where t represents the time the movement is being performed in. In the fifth-order polynomial, each coefficient plays a distinct role in shaping the behavior of the model. The leading coefficient a_1 determines the end behavior of the polynomial as $t \to \pm\infty$, with its sign dictating the direction of growth and its magnitude influencing the steepness. The coefficient a_2 affects the overall curvature of the polynomial, contributing to bending patterns and influencing inflection points. The coefficient a_3 introduces asymmetry, affecting the skewness and balance between rising and falling sections. The quadratic term a_4 adds curvature that impacts the spread and steepness near the central regions, contributing to the formation of local minima and maxima. The linear term a_5 governs the general slope or tilt of the curve, while the constant term a_6 shifts the

graph vertically and defines the y-intercept at $f(0)$. Together, these coefficients determine the polynomial's roots, turning points, inflection points, and overall shape, with higher-order terms dominating the behavior at large $|t|$ and lower-order terms refining the central and local features. The main benefit of determining path using polynomial interpolation, is that the speed can be expressed as a derivation of the equation 2.25:

$$\dot{q} = 5a_1 \cdot t^4 + 4a_2 \cdot t^3 + 3a_3 \cdot t^2 + 2a_4 \cdot t + a_5, \tag{2.26}$$

,
and the acceleration can be expressed as the second derivation:

$$\ddot{q} = 20a_1 \cdot t^3 + 12a_2 \cdot t^2 + 6a_3 \cdot t + 2a_4. \tag{2.27}$$

This allows for a simple calculation of the position, speed and acceleration during the iterative process of optimization in which this calculation will be used. As mentioned the tuning of the paths can be performed by adjusting the values. An example of different paths obtainable with interpolation is given in Figure 2.6. It can be seen that a number of different paths can be generated, with different speed and acceleration profiles. Obviously, each of these paths would have a different energy use. Still, manually determining the path would require adjusting of a lot of values. If we assume that each of the parameters can range from -10 to 10, and this range is discrete with a step of 0.1, the total number of solutions is approximately 6.4×10^{13} for six separate paths (since we are discussing a path of a six-joint IRM). This shows that a need may arise for a more advanced selection algorithm, which will be described in the following sections.

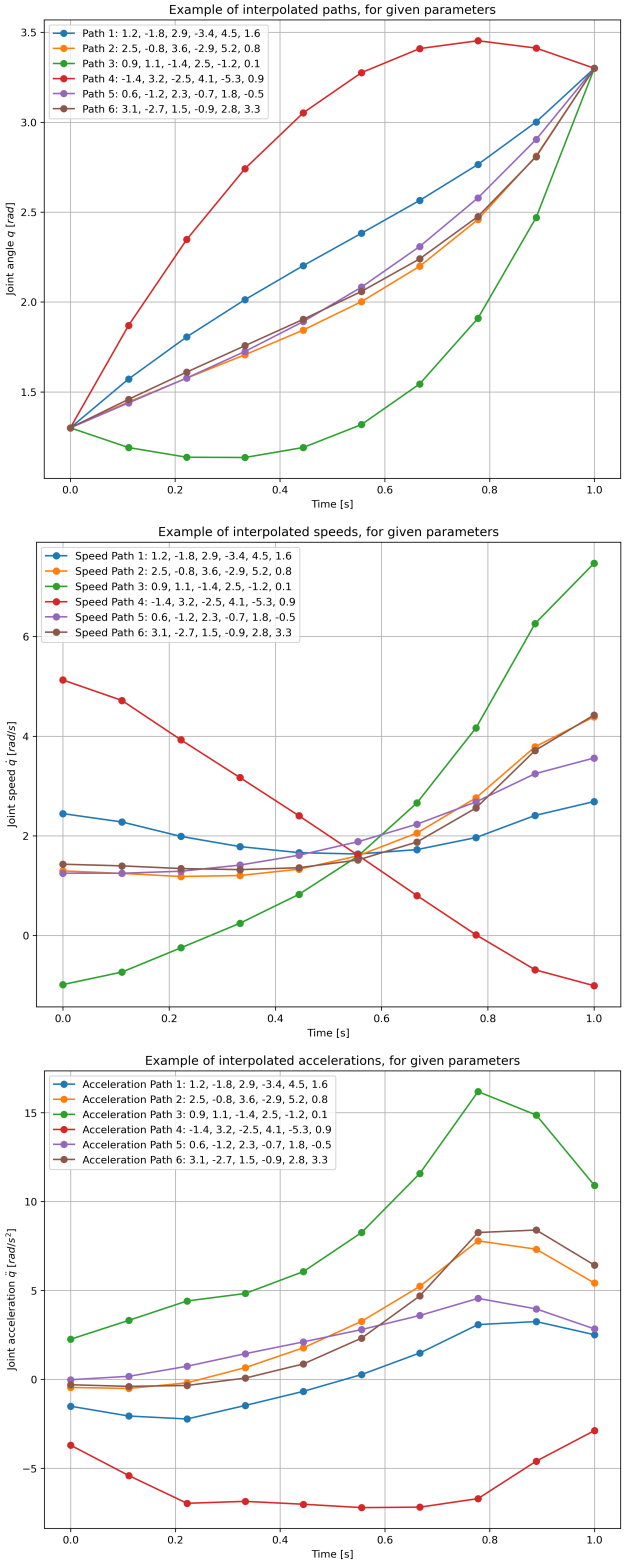


Figure 2.6: A comparison of interpolated paths, speeds and accelerations for different values of polynomial parameters.

2.2 Machine learning approach

Machine learning is one of the approaches used in modeling complex systems. This approach is data-driven. This means that the developed model is directly based on the data that describes the phenomena being modeled. Compared to the classical approach of developing mathematical models, ML algorithms have a simpler application with less room for error, they are adjusted to data, and the solutions may be less computationally expensive when compared to the conventional approach of calculating energy – depending on the approach, the ML-based model may use less memory and computational resources.

In general, the approach of ML algorithms is based on creating a model with a set of internal parameters. How these internal parameters are expressed varies on the particular model, and the descriptions for individual models used in this thesis are given going forward. The goal of ML is to adjust these model parameters, with the goal of achieving the lowest possible error [26]. For a set of data, where each data point represents a single data vector (e.g. in tabular data, as the one used in this thesis, a single data vector equals a row of data), the ML algorithm will calculate a predicted value. Then, the error, commonly referred to as a loss function, will be calculated as the difference between the predicted value and the expected value – the output which corresponds to the input vector. This process is illustrated in Figure 2.7. In the figure, y is the output representing the output associated with the data point $X_{i,:}$ – a single row of data in the dataset X. When this data point is used as an input to the model M, the model will generate an output $\hat{y}$ based on its current parameters w. These two values are used to calculate the error ϵ, also sometimes referred to as the model loss $\mathcal{L}$. Based on the type of the used model and/or optimization, this value is then used to adjust the parameters of the model. This process is iterative, and is repeated for all data points in the dataset, multiple times. The entire process of parameter adjustment is commonly referred to as the training process [33].

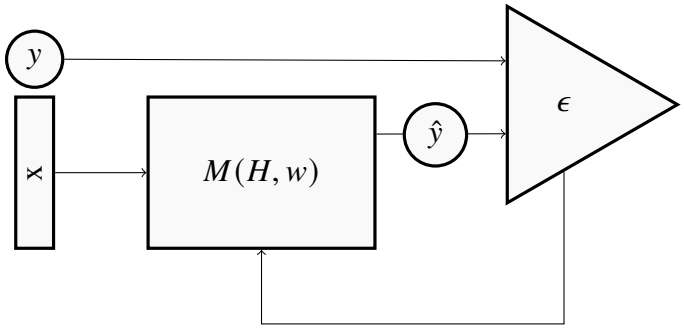


Figure 2.7: The illustration of the basic ML training process on a point of data. x – the input data point, y – the real output value associated with the data point $X_{i,:}$, from the dataset, $\hat{y}$ – the value predicted by model M – the model, defined by the parameter sets H and w.

In addition to the parameters of the model which get adjusted during the training process, each ML model also has an additional, separate, set of parameters that define its general characteristics (e.g. the number of adjustable parameters) and behavior during the training process. To differentiate between the two sets of the parameters, these additional parameters are referred to as hyperparameters [1]. These hyperparameters have a high influence on the final performance of the model – for example, defining a model that is too small (low number of parameters) can result in a model that is not complex enough to describe the information contained within the data. The proper selection of hyperparameter values is a key part of ML modeling. While some general rules exist, it is hard to pick the correct value of hyperparameters. For this reason, many hyperparameter selection techniques exist. The one selected to use in this thesis is the grid search algorithm. This algorithm works by taking a discrete set of possible values for each hyperparameter that is being adjusted, as a set of lists of possible values $H = [h_1, h_2, \cdots, h_n]$. Then, an n-dimensional grid is constructed, and the ML algorithm is trained for each intersection. In simpler terms, training is repeated for each possible combination of hyperparameters [1]. The total number of hyperparameter combinations, or the number of models N that will be trained, is calculated as:

$$N = \Pi_{i=1}^{n} |h_i|. \tag{2.28}$$

The model trained for each hyperparameter combination will be evaluated and their performance noted, together with the hyperparameter set. The possible values of hyperparameters that were adjusted in the presented study will be given in in the following sections that describe the individual algorithms used.

Model evaluation is commonly done on separate data than the one that was used for training. Detailed information on the data used for training models in this thesis is given in Chapter 3. Basically, it needs to be understood that the data is split in the part used for model training – the training set, and the data used for evaluating the model – the validation set. The validation set consists of the vectors $X^V = [X_{1,:}, X_{2,:}, \cdots, X_{n,:}]$, and the corresponding pairs of output values $Y^V = [y_1, y_2, \cdots, y_n]$. After the model that was trained on training data set X^{TR}, Y^{TR} is used on the validation data, generating the set of output values $\hat{Y^V} = [\hat{y_1}, \hat{y_2}, \cdots, \hat{y_n}]$. The model performance can be evaluated by comparing the values of the datasets Y and $\hat{Y}$. To assist in determining the performance of the models, certain metrics may be used to quantify the performance, with the two used in this thesis being the coefficient of determination R^2 and the mean absolute error MAE.

The coefficient of determination – R^2, is a statistical measure that indicates the proportion of variance in the Y dataset that is predicted within the model output set $\hat{Y}$. It provides a measure of how well the observed outcomes are replicated by the model, based on the proportion of total variation explained by the regression model. It can be expressed as the complement of the ratio between the residual sum of squares (unexplained variance) SS_{res}, and the total sum of squares (total variance in data) SS_{tot}, defined as [52]:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}{\Sigma_{i=1}^{n}(y_i - \overline{Y})^2}, \tag{2.29}$$

with $\overline{Y}$ being the mean of the target values contained in the dataset. R^2 ranges between 0 and 1, where the value closer to 1 indicates a better fit of the model (such value would indicate that all of the variance in target data has been contained in the predicted data), while a value closer to zero indicates a poor fit.

The other mentioned metric, MAE, defines the error as the mean absolute difference between the elements of actual values, and predicted values. This metric is useful because it directly illustrates the mean error, in terms of the physical quantity being modeled. According to [30], it is defined as:

$$MAE = \frac{\Sigma_{i=1}^{n} |y_i - \hat{y}_i|}{n}, \tag{2.30}$$

or in words – the sum of absolute error values divided by the number of elements in the dataset. The value of MAE is defined between 0 and ∞, with the values closer to zero indicating a better model.

A possible issue with evaluating data models is accidentally splitting the dataset in such a way that the validation set happens to score well despite actual performance being worse, causing an overestimation of its performance. There are multiple ways to avoid this issue that have been implemented. First, the data is shuffled to avoid testing on similar data. The second is the application of a cross-validation procedure. In the cross-validation procedure the dataset is split into k equal parts. The selection of data for each of these parts is done uniformly randomly – with each data point in the original dataset being equally likely to be placed into one of the five new sets. Now, with the dataset split, we can apply the cross-validation procedure. In this approach the train-test set is split in such a way that it is constructed of four dataset splits – 80% of data. In the first step, this will be the first four data splits. The remaining, fifth, data split (20%) will be used as the validation score. After the training is completed on this data configuration, the training is repeated, except this time, the validation set is changed to another part of the dataset, while the training set is made by combining the remaining four data splits. This process is repeated again, until all k dataset splits have been used as a validation set. This process is illustrated in Figure 2.8, where the dataset (illustrated with a blue square), is split into five parts. Then, the process is shown to repeat five times, with a different data split used for validation being shown in red, while the remaining dataset parts in the data fold, given in green, are combined into the set used for model training.

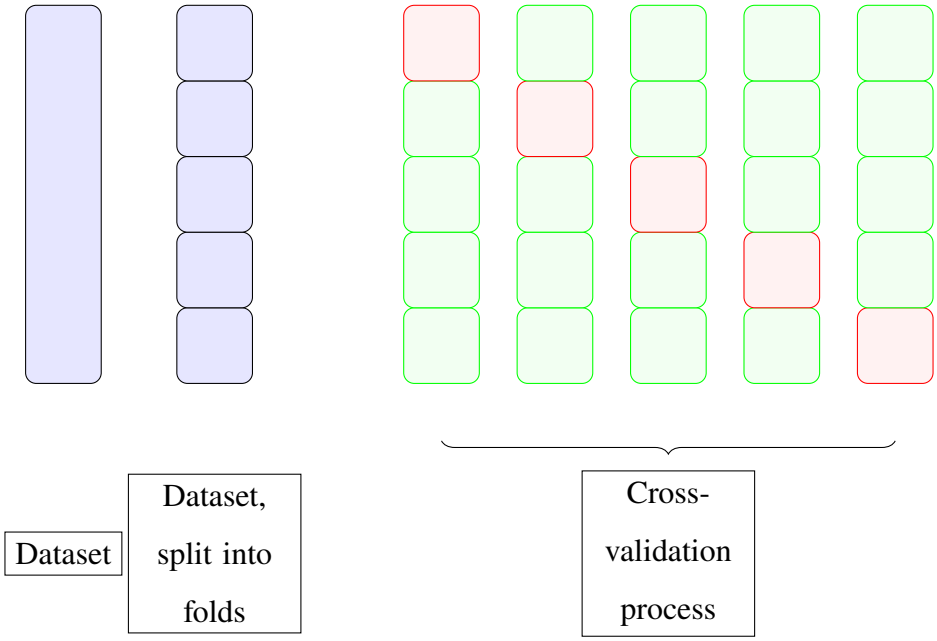


Figure 2.8: An illustration of a cross-validation process, using dataset split into five folds. The dataset parts used for training in each of the steps are indicated with green color, while the dataset parts used for validation are indicated with a red color.

Final thing to note before moving onto the algorithms used for modeling is the use of scaling on the input data. Standard scaling is a fundamental preprocessing step in ML that ensures features have comparable magnitudes, preventing any single feature from disproportionately influencing a model due to differences in scale. This is particularly important in algorithms that rely on distance metrics, such as k-nearest neighbors and support vector machines, where large-magnitude features can dominate smaller ones, leading to biased model behavior [3]. Similarly, gradient-based optimization techniques, including those used in logistic regression and neural networks, benefit from standard scaling, as it facilitates faster convergence by reducing the likelihood of features with different scales causing erratic gradient updates. Without proper scaling, optimization algorithms may struggle to reach an optimal solution efficiently, especially when features exhibit widely varying ranges.

Mathematically, standard scaling transforms a given feature $X_{:,j}$ by subtracting its mean μ_j and dividing by its standard deviation σ_j, as expressed by:

$$X'_{:,j} = \frac{X_{:,j} - \mu_j}{\sigma_j}.$$

This transformation results in features with zero mean and unit variance, where the mean of the transformed feature satisfies $\frac{1}{N}\sum_{i=1}^{N} X'_{i,j} = 0$ and the variance satisfies $\frac{1}{N}\sum_{i=1}^{N}(X'_{i,j} - 0)^2 = 1$ [3]. By centering the data at zero and normalizing its spread, standard scaling enhances numerical stability, reducing the risk of numerical overflow in computations involving exponentiation or high-dimensional matrices. Additionally, it aligns features with the assumptions of many ML algorithms, such as principal component analysis, which assumes that data is standardized to avoid features with larger variances dominating the principal components.

In the contiuation of this chapter the details of algorithms that were used to create regression models. For each algorithm, the description of how the algorithm works will be given, followed with the hyperparameters used in the grid search process.

2.2.1 Multilayer perceptron algorithm

A multilayer perceptron (MLP) is a type of feed-forward ANN characterized by its layered architecture, consisting of an input layer, one or more hidden layers, and an output layer. Each layer in an MLP is fully connected to the next, forming a dense network of weighted connections. The ability of MLPs to model complex, non-linear relationships between input and output data stems from the use of non-linear activation functions in the nodes of hidden layers. These activation functions introduce non-linearity into the model, enabling it to capture intricate patterns and dependencies in data [76]. The structure and adaptability of MLPs make them widely applicable in supervised learning tasks, such as classification and regression.

The function of an MLP can be expressed mathematically. For a given neuron j in layer l, its output $a_j^{(l)}$ is computed as:

$$a_j^{(l)} = \phi\left(\sum_{i=1}^{n} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right), \tag{2.31}$$

where $w_{ij}^{(l)}$ is the weight connecting neuron i in layer $l-1$ to neuron j in layer l, $b_j^{(l)}$ is the bias term for neuron j, $a_i^{(l-1)}$ is the output of neuron i in the previous layer, and ϕ is the activation function. Typical choices for ϕ include the rectified linear unit (ReLU), sigmoid, and hyperbolic tangent (tanh) functions, each offering different benefits depending on the application [53].

The training of an MLP involves adjusting its weights and biases to minimize the difference between its predictions and the actual target values, a process guided by a loss function. Common loss functions include mean squared error (MSE) for regression problems and cross-entropy loss for classification problems. The training process typically employs the backpropagation algorithm combined with gradient-based optimization techniques such as stochastic gradient descent (SGD) or adaptive moment estimation (Adam) [60]. The training begins with a forward pass, during which the input data propagate through the network. The network's output is computed layer by layer by applying the weighted sums and activation functions until the final output is produced. The computed output is then compared to the target values using the loss function. For instance, in a binary classification task with cross-entropy loss, the loss $\mathcal{L}$ is given by:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right], \tag{2.32}$$

where N is the number of data points, y_i represents the true label, and $\hat{y}_i$ is the predicted output for the i-th sample.

After the forward pass, the backpropagation algorithm computes the gradients of the loss function with respect to all trainable parameters in the network. Using the chain rule of differentiation, the algorithm calculates these gradients layer by layer, starting from the output layer and propagating backward to the input layer. The gradient for a weight $w_{ij}^{(l)}$ connecting neuron i in layer $l-1$ to neuron j in layer l is computed as:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}, \tag{2.33}$$

where $\delta_j^{(l)}$ represents the error term for neuron j in layer l. This error term captures

how much the loss would change if the output of that neuron were altered, and it is given by:

$$\delta_j^{(l)} = \phi'(z_j^{(l)}) \sum_{k=1}^{m} \delta_k^{(l+1)} w_{jk}^{(l+1)}, \tag{2.34}$$

where $\phi'(z_j^{(l)})$ is the derivative of the activation function with respect to the weighted input $z_j^{(l)}$, and $\delta_k^{(l+1)}$ is the error term for neuron k in the subsequent layer.

Once the gradients are calculated, the optimization algorithm updates the weights and biases to reduce the loss. For example, in stochastic gradient descent, the weight update rule is:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}, \tag{2.35}$$

where η is the learning rate, a hyperparameter controlling the step size of the updates. The biases are updated similarly. Modern optimization algorithms such as Adam improve upon basic SGD by adapting the learning rate for each parameter based on past gradient information, often leading to faster convergence. Training is an iterative process that continues for multiple epochs, where each epoch involves a complete pass over the training dataset. The goal is to reduce the loss function as much as possible, enabling the MLP to learn a mapping that generalizes well to unseen data [2].

As mentioned previously, hyperparameters can have a high rate of the influence on the performance of the model, as they dictate not only the number of internal parameters, but also the way in which the learning is performed. The hyperparameters that are tuned using GS for MLP are the size of hidden layers, activation, solver, alpha, learning rate and initial learning rate value (given as η previously). The hidden layer sizes hyperparameter specifies the architecture of the network by determining the number of neurons in each hidden layer, which directly influences the model's capacity to capture complex patterns. The activation hyperparameter defines the function applied to the outputs of the neurons in the hidden layers, introducing non-linearity to the model and enabling it to learn complex relationships in the data. The solver hyperparameter determines the optimization algorithm used to adjust the model's weights during training,

influencing the speed and convergence of the learning process. The alpha parameter regulates the complexity of the model by penalizing large weight magnitudes, which helps to improve generalization and prevent overfitting. The learning rate parameter controls how the step size for weight updates changes over time, affecting the stability and convergence of training. Finally, the initial learning rate parameter sets the initial value of the step size, impacting the scale of updates to the model's weights at the start of the training process [60]. The possible values of these hyperparameters, which were used in the tuning process, are given in Table 2.2. The hyperparameter values in question were selected to provide a comprehensive yet computationally feasible exploration of MLP performance across different configurations. Hidden layer sizes were chosen to evaluate the effect of both network depth and width on learning capacity, with values ranging from shallow and narrow architectures such as (10) to deeper and wider networks like (100, 100, 100, 100), allowing for assessment of underfitting and overfitting tendencies. Activation functions included identity, logistic, tanh, and relu to compare linear and non-linear activation behaviors, with relu being widely used for its efficiency and logistic and tanh for their suitability in smaller networks or for capturing smoother transitions. The solvers lbfgs and adam were selected based on their distinct optimization approaches: lbfgs is effective for smaller networks due to its quasi-Newton method, while adam is adaptive and well-suited for larger datasets with potentially noisy gradients. The alpha values of 0.0001, 0.001, 0.01, and 0.1 span several orders of magnitude to examine how different regularization strengths influence the trade-off between model complexity and generalization, with smaller values allowing more flexibility in weight magnitudes and larger values enforcing stricter regularization. Both constant and adaptive learning rate schedules were tested to observe the impact of static versus responsive learning dynamics, while initial learning rates of 0.01, 0.1, and 1 were included to assess training stability and convergence speed, with $1 \cdot 10^{-2}$ serving as a conservative baseline and 1 testing the limits of fast learning. Overall, the selected values reflect commonly used ranges present in previous research by the author.

Table 2.2: Hyperparameter values tested in the GS process for MLP.

Hyperparameter	Values
Hidden Layer Sizes	(10), (50), (100), (10, 10), (50, 50), (100, 100), (10, 10, 10), (50, 50, 50), (100, 100, 100), (10, 10, 10, 10), (50, 50, 50, 50), (100, 100, 100, 100)
Activation function	`identity`, `logistic`, `tanh`, `relu`
Solver	`lbfgs`, `adam`
Alpha	0.0001, 0.001, 0.01, 0.1
Learning rate type	`constant`, `adaptive`, `invscaling`
Initial Learning rate	0.01, 0.1, 1

As the table shows, in addition to the learning rate remaining constant, it can also be adjusted using two different strategies. In the adaptive schedule, the learning rate remains constant unless the training loss does not improve for a specified number of iterations (patience). When this occurs, the learning rate is typically reduced by a factor (e.g., halved). Although this behavior varies by implementation, a simplified form is:

$$\eta_{t+1} = \begin{cases} \eta_t, & \text{if improvement} \\ \gamma \cdot \eta_t, & \text{otherwise} \end{cases}$$

where η_t is the learning rate at iteration t, $\gamma \in (0, 1)$ is the reduction factor (e.g., $\gamma = 0.5$). In the inverse scaling (given in code as `invscaling`) schedule, the learning rate decreases smoothly over time based on the inverse of a scaling factor:

$$\eta_t = \eta_0 \cdot t^{-\gamma}$$

where: - η_0 is the initial learning rate, - t is the current iteration number, - $\gamma \in (0, 1]$ is the power of the inverse scaling.

This schedule gradually slows down learning as training progresses, which can help fine-tune the model and avoid overshooting minima. Both methods aim to balance fast initial learning with stable convergence, but adaptive learning adjusts based on

performance, while inverse scaling depends solely on iteration count [60]. The regularization parameter alpha controls the strength of L2 regularization, which penalizes large weights to help prevent overfitting. The regularization term is added to the loss function, modifying it as follows:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \alpha \sum_i w_i^2$$

where $\mathcal{L}$ is the original loss (e.g., mean squared error or cross-entropy), w_i are the model weights, and α is the regularization strength. A larger alpha increases the penalty on large weights, encouraging the model to keep weights smaller, which typically improves generalization but may limit the model's ability to fit complex patterns. Conversely, a smaller alpha allows the model more flexibility to fit the training data, at the risk of overfitting. The selection of alpha balances model complexity and generalization, with typical values chosen across several orders of magnitude (e.g., 0.0001 to 0.1) to explore this trade-off.

2.2.2 Support vector regressor algorithm

Support Vector Regressor (SVR) is a supervised learning algorithm that extends the principles of Support Vector Machines (SVMs) to regression tasks. The main idea behind SVR is to find a function $f(X_{i,:})$ that approximates the relationship between the input data $X_{i,:}$ and the target values y within a specified margin of tolerance ϵ. Unlike other regression methods that minimize the prediction error directly, SVR aims to achieve a balance between model complexity and prediction accuracy by penalizing deviations beyond the ϵ-margin while keeping the model's parameters as simple as possible [74].

In SVR, the optimization objective is to minimize the following function:

$$\frac{1}{2}\|w\|^2 + C \sum_{i=1}^{N} (\xi_i + \xi_i^*), \tag{2.36}$$

where w represents the weight vector defining the model, ξ_i and ξ_i^* are slack variables

that measure the extent of deviation from the ϵ-margin, C is the regularization parameter that controls the trade-off between margin violations and model complexity, and N is the number of training data points. The term $\frac{1}{2}\|w\|^2$ ensures that the model complexity is minimized, encouraging a simpler solution.

The function $f(X_{i,:})$ that SVR learns is expressed as:

$$f(X_{i,:}) = 2^\top \phi(X_{i,:}) + b, \tag{2.37}$$

where $\phi((X_{i,:}))$ is a non-linear transformation mapping the input data to a higher-dimensional feature space, enabling the algorithm to model non-linear relationships. The bias term b shifts the decision boundary, and both w and b are determined during training. The kernel trick allows computation in the original feature space without explicitly transforming $X_{i,:}$, using a kernel function $K(X_{i,j}) = \phi((X_{i,:}))^\top \phi(X_{:,j})$ [74].x

The optimization is subject to the following constraints for each data point $(X_{i,:}, y_i)$:

$$\begin{aligned} y_i - f(X_{i,:}) &\leq \epsilon + \xi_i, \\ f(X_{i,:}) - y_i &\leq \epsilon + \xi_i^*, \\ \xi_i, \xi_i^* &\geq 0. \end{aligned} \tag{2.38}$$

Here, ϵ defines the margin of tolerance around the true value y_i, within which predictions are not penalized. The slack variables ξ_i and ξ_i^* represent the deviations from the ϵ-margin for over-predictions and under-predictions, respectively. These constraints ensure that the model focuses only on errors outside the margin, while ignoring smaller deviations [60].

To solve this constrained optimization problem, SVR employs Lagrange multipliers, a mathematical tool used to optimize a function subject to equality or inequality constraints. The dual formulation of the problem allows for efficient computation, focusing on support vectors, which are the data points lying on or outside the ϵ-margin. These support vectors directly influence the model's predictions, while other data points do not contribute to the solution [15].

The regression function in the dual formulation is expressed as:

$$f(X_{i,:}) = \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) K(X_{i,j}) + b, \tag{2.39}$$

where α_i and α_i^* are the Lagrange multipliers corresponding to the slack variables ξ_i and ξ_i^*. The sparsity of the solution arises because many α_i and α_i^* are zero, leaving only the support vectors to define the regression function [15].

Hyperparameters such as C, ϵ, and the kernel parameters significantly influence the SVR's performance. The regularization parameter C determines the trade-off between the flatness of the regression function and the tolerance for margin violations. A smaller ϵ creates a narrower margin, focusing the model on more accurate predictions but potentially overfitting the data. The choice of the kernel function, whether linear, polynomial, or radial basis function (RBF), determines the nature of the mapping $\phi((X_{i,:}))$ and hence the types of relationships the model can capture [60].

By training on a dataset, SVR learns to approximate the target function with minimal complexity and deviations beyond the tolerance margin. Its formulation makes it particularly effective for regression problems where robustness to outliers and generalization to unseen data are critical. With proper tuning of its hyperparameters, SVR can provide accurate and interpretable solutions for complex regression tasks.

When it comes to the tuned hyperparameters for the SVM regressor, they can be seen in Table 2.3. The kernel hyperparameter specifies the type of kernel function used to map input data into a higher-dimensional space, influencing the model's ability to capture complex relationships. The degree parameter is relevant for the polynomial kernel, defining the degree of the polynomial and controlling the flexibility of the decision boundary, with higher values allowing for more complex patterns. The γ parameter determines the influence of a single training example, effectively controlling how far the impact of a single support vector extends: a higher γ focuses on closer points, while a lower γ considers a broader range. The C hyperparameter represents the regularization parameter, balancing the trade-off between achieving a low error on the training data and maintaining a smooth model. Finally, the ε parameter defines a margin of tolerance around the predicted values within which no penalty is given for errors, allowing

the model to focus on capturing significant patterns and ignore small fluctuations. Together, these hyperparameters allow fine-tuning of the SVM regressor for performance and generalization [60]. The hyperparameter values were selected to enable an exploration of SVM regression behavior. The kernel choices—linear, polynomial, RBF, and sigmoid – represent a range of transformations from simple linear mappings to complex non-linear functions, allowing the model to adapt to various data patterns. The degree parameter, relevant for the polynomial kernel, was varied between 2 and 4 to control the flexibility of the decision boundary, with higher degrees enabling more complex relationships. The γ parameter, which influences the shape of the decision boundary in RBF, poly, and sigmoid kernels, was tested using the scale and auto options: scale sets γ to $1/(n_{\text{features}} \cdot \text{Var}(X))$, while auto sets γ to $1/n_{\text{features}}$, both of which adapt to data characteristics to avoid manual tuning. The regression parameter C was varied from 0.1 to 10 to control the trade-off between minimizing the training error and maintaining a smooth regression function, with smaller values allowing more tolerance for error and larger values enforcing stricter fitting. Finally, the epsilon parameter ε defines the width of the epsilon-insensitive tube within which no penalty is given for errors; values ranging from 0.1 to 1 were tested to examine how tolerant the model is to small deviations from the target, with smaller values aiming for higher precision and larger values promoting sparsity in the support vectors.

Table 2.3: Hyperparameters and their values for SVM Regressor.

Hyperparameter	Values
Kernel	`linear`, `poly`, `rbf`, `sigmoid`
Degree	2, 3, 4
γ	`scale`, `auto`
C	0.1, 0.5, 1, 10
ε	0.1, 0.2, 0.5, 1

2.2.3 Passive-aggressive regressor

Passive Aggressive Regressor (PAR) is an efficient and robust algorithm designed for regression tasks, particularly in online learning scenarios where data arrives sequentially. Unlike traditional regression methods that minimize a loss function over an entire dataset, PAR updates its model parameters incrementally by processing one data point at a time. The term "passive-aggressive" reflects the algorithm's behavior: it remains passive if the current prediction is within an acceptable error margin and becomes aggressive when the prediction error exceeds a predefined threshold [29].

The goal of PAR is to minimize a hinge-loss-based objective function while keeping the model's updates constrained. Given a single data point $(X_{i,:}, y_i)$, the optimization objective for PAR can be expressed as:

$$\min_{w} \frac{1}{2}\|w - w_{\text{prev}}\|^2 \quad \text{subject to } |y_i - w^\top (X_{i,:})| \leq \epsilon, \tag{2.40}$$

where w represents the current weight vector, w_{prev} is the weight vector from the previous step, y_i is the true target value, $(X_{i,:})$ is the input feature vector, and ϵ is the margin of tolerance for the prediction error. The term $\frac{1}{2}\|w - w_{\text{prev}}\|^2$ ensures that updates to the model are minimal, preserving stability while adapting to new data [29].

When the prediction error exceeds the margin ϵ, the algorithm updates w to correct the prediction. The update rule for the weight vector is derived by introducing a Lagrange multiplier τ to enforce the margin constraint, leading to the following closed-form solution:

$$w \leftarrow w_{\text{prev}} + \tau \cdot X_{i,:}, \tag{2.41}$$

where τ is the step size, calculated as:

$$\tau = \frac{\max(0, |y_i - w^\top X_{i,:}| - \epsilon)}{\|X_{i,:}\|^2 + \frac{1}{C}}, \tag{2.42}$$

and C is a regularization parameter that controls the aggressiveness of updates. Larger values of C result in more significant updates, while smaller values encourage conser-

vative adjustments. The denominator $\|X_{i,:}\|^2 + \frac{1}{C}$ ensures that updates are proportional to the magnitude of the input features, preventing overly large weight changes.

The hinge-loss-based constraint ensures that the model focuses only on instances where the prediction error is large, making it robust to outliers and noise in the data. The use of ϵ defines a tolerance margin, within which predictions are considered satisfactory. This mechanism prevents unnecessary updates for minor deviations, contributing to the algorithm's computational efficiency [71].

While PAR does not explicitly involve kernel methods or transformations, its linear formulation can be extended to non-linear regression tasks through the application of the kernel trick. This extension involves substituting the dot product $w^\top X_{i,:}$ with a kernel function $K(X_{i,j})$, enabling the algorithm to capture non-linear relationships while maintaining its online learning capabilities.

Again, the table for possible hyperparameters is given for PAR as well, in Table 2.4. The C hyperparameter represents the regularization strength, balancing the trade-off between minimizing training error and controlling model complexity. Smaller values of C encourage stronger regularization, which can improve generalization but may lead to higher training error, while larger values reduce regularization, allowing the model to fit the training data more closely. The fit_intercept hyperparameter in the PAR determines whether an intercept term is included in the model. If set to True, the model learns an additional bias term that shifts the decision boundary or prediction function, which can be crucial when the features are not centered or the target values are offset. For example, if the input data has a non-zero mean, enabling the intercept allows the model to align predictions better with the target values by accounting for this offset. Conversely, if set to False, the model assumes that the data is already centered, and no bias term is added, which may simplify the model but risks introducing systematic errors if the data requires centering. This setting is particularly relevant for datasets that have been preprocessed or standardized, where excluding the intercept can improve computational efficiency without sacrificing accuracy. The tolerance hyperparameter specifies the tolerance for stopping criteria, defining how small the changes in the optimization

objective must be to stop further iterations, with lower values leading to potentially more accurate but computationally intensive solutions. The loss parameter determines the type of loss function used to measure prediction errors, such as epsilon-insensitive loss or squared epsilon-insensitive loss, which affect how deviations from the true target within the epsilon margin are penalized. Finally, the epsilon hyperparameter defines the margin of tolerance, where predictions falling within this range of the true values are not penalized. This allows the model to focus on capturing significant trends while ignoring minor deviations, which is particularly useful when dealing with noisy data [60]. The hyperparameter values were selected to explore the behavior of PAR under different settings. The parameter C, ranging from 0.1 to 10, controls the trade-off between staying close to the current model (passive) and aggressively updating it when errors occur; smaller values allow more tolerance for deviations, while larger values enforce stricter corrections. The tolerance parameter, tested at 10^{-3}, 10^{-4}, and 10^{-5}, defines the stopping criterion for the optimization process, with smaller values enabling more precise convergence at the cost of longer computation time. Two loss functions were included: epsilon_insensitive, which ignores errors within a margin of ε, and squared_epsilon_insensitive, which squares errors outside the margin, penalizing larger deviations more heavily. The epsilon parameter ε, varied from 0.1 to 1, specifies the width of the epsilon-insensitive zone where errors incur no penalty; smaller values demand more precise predictions, while larger values tolerate more deviation from the target.

Table 2.4: Hyperparameters and their values for PAR.

Hyperparameter	Values
C	0.1, 0.5, 1, 10
`fit_intercept`	`True`, `False`
Tolerance	10^{-3}, 10^{-4}, 10^{-5}
Loss	`epsilon_insensitive`, `squared_epsilon_insensitive`
ε	0.1, 0.2, 0.5, 1

2.2.4 Gradient boosted trees

Gradient Boosted Trees (GBT) are an ensemble learning technique designed for regression and classification tasks, combining the predictions of multiple weak learners, typically decision trees, to produce a strong predictive model. The fundamental idea behind GBT is to iteratively build new trees that minimize the residual errors of the ensemble model. XGBoost (Extreme Gradient Boosting) is a widely used implementation of GBT that incorporates advanced optimization techniques and system-level improvements for enhanced performance and computational efficiency [24].

In GBT, the model is constructed iteratively, starting with a simple base model, often a constant prediction. At each iteration t, a new decision tree $f_t(X_{i,:})$ is added to the ensemble to reduce the residual errors from the previous predictions. The model at iteration t can be expressed as:

$$\hat{y}^{(t)} = \hat{y}^{(t-1)} + \eta^{f_t} f_t(X_{i,:}), \tag{2.43}$$

where $\hat{y}^{(t)}$ is the prediction of the ensemble at iteration t, $\hat{y}^{(t-1)}$ is the prediction from the previous iteration, $f_t(X_{i,:})$ is the newly added tree, and η^{f_t} is the learning rate that controls the contribution of each individual tree [48]. The objective is to minimize a loss function $\mathcal{L}(y, \hat{y})$, such as MSE for regression or log-loss for classification.

In XGBoost, the optimization process involves a regularized objective function to improve generalization and prevent overfitting. The objective at iteration t is given by:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)} + f_t((X_{i,:}))) + \Omega(f_t), \tag{2.44}$$

where $l(y_i, \hat{y}_i)$ is the loss function measuring the difference between the true value y_i and the predicted value $\hat{y}_i$, and $\Omega(f_t)$ is a regularization term penalizing the complexity of the tree $f_t(X_{i,:})$. The regularization term is expressed as:

$$\Omega(f_t(X_{i,:})) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2, \tag{2.45}$$

where T is the number of leaves in the tree, w_j is the weight of the j-th leaf, γ controls the penalty for the number of leaves, and λ penalizes the magnitude of leaf weights [48]. These regularization parameters improve the model's robustness by controlling overfitting.

To optimize the objective, XGBoost uses a second-order Taylor expansion of the loss function. For a given tree structure, the objective can be approximated as:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{N} \left[g_i f_t(X_{i,:}) + \frac{1}{2} h_i f_t^2((X_{i,:})) \right] + \Omega(f_t), \tag{2.46}$$

where $g_i = \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i}$ and $h_i = \frac{\partial^2 l(y_i, \hat{y}_i)}{\partial \hat{y}_i^2}$ are the first and second derivatives of the loss function with respect to the predictions, often referred to as the gradient and Hessian, respectively. These derivatives guide the optimization by quantifying the direction and magnitude of the loss change [14].

The optimal weight w_j for each leaf j in the tree is determined by solving:

$$w_j = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}, \tag{2.47}$$

where I_j is the set of data points assigned to leaf j. The corresponding optimal loss reduction for a split is:

$$\mathcal{R} = \frac{1}{2} \left(\frac{\left(\sum_{i \in I_L} g_i\right)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{\left(\sum_{i \in I_R} g_i\right)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{\left(\sum_{i \in I} g_i\right)^2}{\sum_{i \in I} h_i + \lambda} \right) - \gamma, \tag{2.48}$$

where I_L and I_R are the sets of data points in the left and right child nodes, and I is the parent node. This formula determines the quality of a split and is used to grow the tree in a greedy manner, selecting splits that maximize the reduction in loss [14].

Finally, the hyperparameters which were adjusted for GBT, along with their values are given in Table 2.5. In the context of gradient boosting models, the n_estimators hyperparameter specifies the number of boosting iterations or decision trees used in the ensemble, controlling the model's capacity and complexity. The max_depth parameter limits the maximum depth of each individual tree, helping to balance underfitting (shallow trees) and overfitting (deep trees). The learning_rate controls the contribution

of each tree to the final prediction, with smaller values requiring more trees for convergence but potentially improving generalization. The subdata point parameter determines the fraction of the training data sampled to grow each tree, which can reduce overfitting and increase robustness by introducing randomness. The colsample_bytree hyperparameter specifies the fraction of features considered for splitting at each tree level, while colsample_bylevel and colsample_bynode control the fraction of features used at each level of the tree and for each split, respectively, allowing fine-grained control over feature sampling. The reg_alpha parameter adds L1 regularization to the loss function, encouraging sparsity in the model and reducing overfitting by shrinking less important weights. Similarly, reg_lambda applies L2 regularization, penalizing large weights to enhance generalization and stability. Together, these hyperparameters provide flexibility to tune the gradient boosting model for various datasets and tasks [60]. The hyperparameter values were selected to provide meaningful control over model complexity, regularization, and feature sampling, while remaining within ranges that are commonly effective in practice. The values for n_estimators—10, 50, and 100—were chosen to explore the impact of ensemble size on performance and training time. A small value like 10 allows quick training and evaluation, useful for rapid prototyping, while 100 provides a more thorough boosting process capable of reducing bias. The max_depth values from 3 to 7 were selected because shallow trees (depth 3 — 4) help prevent overfitting, especially on noisy or small datasets, while deeper trees (up to 7) allow modeling of more intricate patterns in the data. The learning_rate values of 0.01, 0.1, and 1 span from conservative to aggressive learning. A rate of $1 \cdot 10^{-2}$ typically requires more estimators but leads to better generalization, 0.1 is a standard default offering a balance of speed and accuracy, and 1 is included to test the limits of rapid convergence, although it can risk instability. For subsample, values of 0.5, 0.75, and 1 were selected to explore how reducing the training sample size per tree affects variance and robustness. Lower values introduce stochasticity that can reduce overfitting, while 1 uses the full data, maximizing information per iteration. Similarly, colsample_bytree, colsample_bylevel, and colsample_bynode—each set to 0.5, 0.75, and 1—control fea-

ture subsampling at different granularities. These values allow examination of how limiting feature usage influences model diversity and generalization. Lower values help reduce correlation between trees and often improve performance on high-dimensional data. Lastly, the regularization parameters reg_alpha and reg_lambda were tested at 0, 0.1, 0.5, and 1 to understand how L1 and L2 penalties influence overfitting. A value of 0 implies no regularization, while increasing values progressively enforce sparsity (via L1) and weight shrinkage (via L2), encouraging simpler models. These specific values span typical operational ranges and were selected based on empirical evidence from practical applications to ensure both efficiency and effectiveness during hyperparameter tuning.

Table 2.5: Hyperparameters and their values for GBT.

Hyperparameter	Values
n_estimators	10, 50, 100
max_depth	3, 4, 5, 6, 7
learning_rate	0.01, 0.1, 1
subsample	0.5, 0.75, 1
colsample_bytree	0.5, 0.75, 1
colsample_bylevel	0.5, 0.75, 1
colsample_bynode	0.5, 0.75, 1
reg_alpha	0, 0.1, 0.5, 1
reg_lambda	0, 0.1, 0.5, 1

2.3 Optimization

2.3.1 Defining a fitness function and selection

As the ML models had certain elements in common, so do the evolutionary algorithms that will be used for the optimization process. The key elements that will be covered in this section are the terms of population, generations, solutions space, fitness function

and element selection.

The evolutionary algorithms operate using the process of natural selection from the solution space. The solution space is an n-dimensional space containing all possible solutions that can be present. In the presented study, we know that each joint has its movement defined with a fifth-order polynomial. Each of these polynomials will have six coefficients, according to the equation 2.25. As the robot has six joints, this means that a single path can be defined with a 6×6 matrix of coefficient values, as each joint of the IRM can have a different path:

$$\begin{bmatrix} a_1^1 & a_2^1 & a_3^1 & a_4^1 & a_5^1 & a_6^1 \\ a_1^2 & a_2^2 & a_3^2 & a_4^2 & a_5^2 & a_6^2 \\ a_1^3 & a_2^3 & a_3^3 & a_4^3 & a_5^3 & a_6^3 \\ a_1^4 & a_2^4 & a_3^4 & a_4^4 & a_5^4 & a_6^4 \\ a_1^5 & a_2^5 & a_3^5 & a_4^5 & a_5^5 & a_6^5 \\ a_1^6 & a_2^6 & a_3^6 & a_4^6 & a_5^6 & a_6^6 \end{bmatrix}, \tag{2.49}$$

with each individual element having a form o a_i^j, where i is the order of the coefficient, and j is the robot joint the path of which the coefficient is associated with. If this is a form of one solution (set of paths in the joint space), expressed as a mathematical form (also known as a chromosome in the evolutionary computing), we can approximate the number of possible solutions. The possible range of each coefficient is $< -10.0, 10.0 >$. This range needs to be discretized. The discretization step between solutions was selected as 0.05. The step has been selected so that the difference between two possible paths is visible, but not so significantly different that it would be possible to miss a good solution. This is demonstrated in Figure 2.9, which shows how modifying the parameters a_1 to a_6 of the path, by a certain value, influences a certain path. The leftmost subfigure, for the discretization parameter value of 0.01, shows that the modified paths barely show any difference, even in detailed, zoomed in views of certain areas. On the other hand, the right-most subfigure, for the value of 0.1, shows visible empty space between the lines (apparent in the detail of the figure). The central subfigure does not show any of those gaps, while still demonstrating that most paths are at least visibly

different, which is why it was selected as the used discretization value.

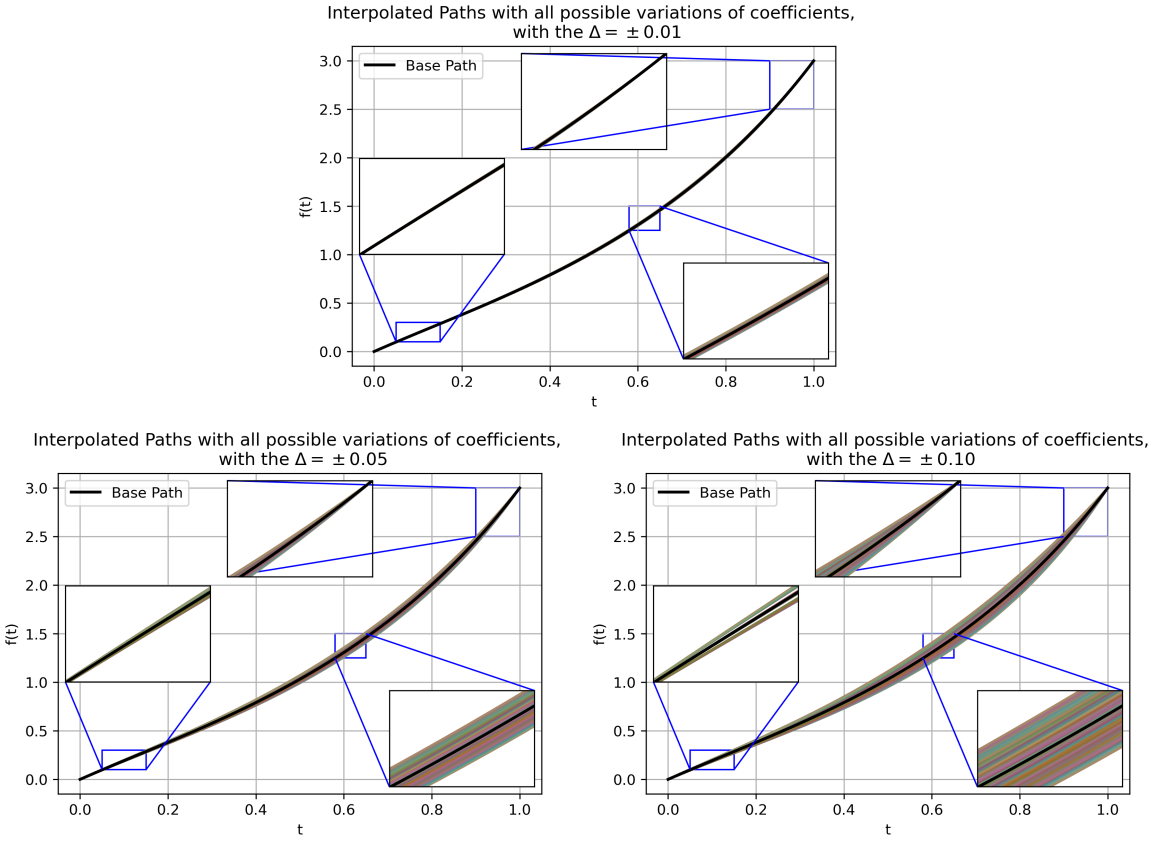


Figure 2.9: A comparison of the influence of different discretization steps on the paths.

With this, we can calculate that each coefficient in the solution can have 400 possible discrete values. Considering that the total number of coefficients is 36, the number of possible solutions is 400^{36} or approximately 4.72×10^{93}. The sheer number of solutions means that it would not be possible to perform an exhaustive search of all possible solutions, indicating a need for a different search algorithm.

Evolutionary algorithms perform the optimization by taking a number of solutions, with this set of candidate solutions being called a population. Then, they create new solutions by combining existing solutions and generating new ones. Each replacement of population with new solutions indicates a new iteration of the algorithm – called a generation (t) [37]. While the generation of new solutions may differ between algorithms, two more elements are common and key to the improvement, first being the fitness

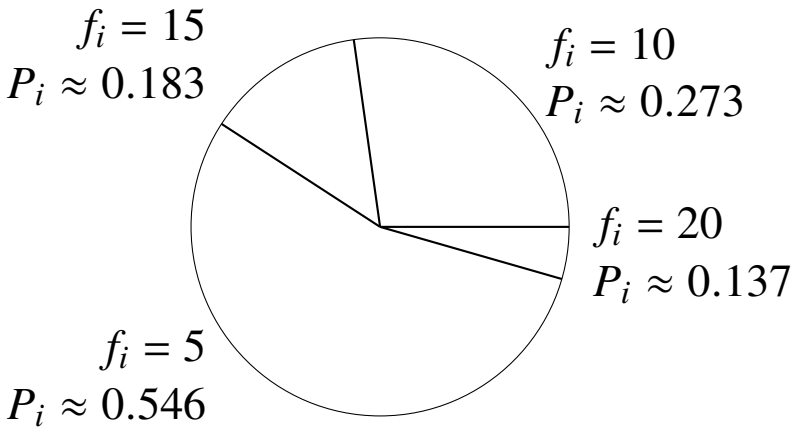


Figure 2.10: RWS illustration, with values of fitness and probability as given in Table 2.6.

function, and the other being the fitness proportional selection.

Fitness function, also known as the criterion function, is a function used to evaluate how well does a candidate solution fit the given problem. In other words, it is the function that the algorithm is optimizing against, looking to either increase or decrease the value. In the presented research this function is the total energy of certain joint path. The goal of the application of the algorithm is lowering the output value of this function. The improvement in fitness function is the main goal, and the key step in achieving this is the fitness proportional selection. By selecting the individual with a better (in the presented case, smaller) fitness function value to be used as operators in the evolutionary calculations used to generate new populations of candidate solutions, the entirety of the system should tend towards a better solution.

There are numerous ways of achieving this, such as elitism, ranking, and tournament selection [37], but the one selected for the algorithm at hand is the roulette wheel selection. The roulette wheel selection (RWS) works by creating assigning a likelihood of selection proportional to the individual candidate solutions fitness, with the better fitting solutions having a higher likelihood of being selected. In other words, it is equivalent of creating a wheel, sectioned in the parts equal to the number of elements in the population, and then adjusting the width of each of this sections to be proportional to their fitness, as shown in Figure 2.10.

Mathematically and programatically this is achieved by taking the total sum of adjusted fitness values (F_{TOTAL}) per:

$$F_{TOTAL} = \Sigma_{i=1}^{N} f_i', \tag{2.50}$$

where f_i' is the adjusted fitness value. This value is used because our real fitness value, the energy of the robot movement, is being minimized, so in reality we are creating the inverse of the standard RWS, since the likelihood of selection is inversely proportional to the fitness function. So, the adjusted fitness value of the i-th population member is:

$$f_i' = \frac{1}{f_i} \tag{2.51}$$

The above equation will commonly have the term ε, representing an extremely small value, added to the denominator, to avoid division by zero errors. In the presented research this is not necessary, as the energy of any joint path cannot be equal to zero. The probability of a i-th solution being selected is then calculated as:

$$P_i = \frac{f_i'}{F_{TOTAL}}. \tag{2.52}$$

To perform the actual selection on the "wheel", the cumulative probability of the i-th individual is calculated as the sum of the individual solutions' fitness values:

$$C_i = \Sigma_{j=1}^{i} P_j. \tag{2.53}$$

Then, a random value r is generated uniformly randomly on the range $[0, 1]$, and the selected individual is the one satisfying the condition of being the first element that has the value C_i equal or greater than the value r. An example of how these values are calculated is shown in Table 2.6, which shows how the different values of fitness change the probability of the solution being selected in an algorithm whose goal is the minimization of the fitness value.

Table 2.6: Example of Fitness Proportional Selection for Minimization.

Individual	(f_i)	(f_i')	(P_i)	(C_i)
1	10	$\frac{1}{10} = 0.1$	$\frac{0.1}{0.366} \approx 0.273$	0.273
2	20	$\frac{1}{20} = 0.05$	$\frac{0.05}{0.366} \approx 0.137$	0.410
3	5	$\frac{1}{5} = 0.2$	$\frac{0.2}{0.366} \approx 0.546$	0.956
4	15	$\frac{1}{15} \approx 0.067$	$\frac{0.067}{0.366} \approx 0.183$	1.000

In the presented case, fitness can be described as function of a single value ($\mathcal{F}(\S)$), or multiple ones ($\mathcal{F}(\S_\infty, \S_\in, \cdots, \S_\setminus)$). When the system is optimized against a single value, this process is referred to as as a single-objective optimization, while the latter is referred to as multi-objective optimization. Both approaches are tested in the presented research, with single-objective using a single function for the total energy, while the multi objective uses multiple optimization functions (one for each joint's energy) and sorts the candidate solutions according to the values.

2.3.2 Genetic algorithm

GA is a search and optimization technique inspired by the principles of natural selection and genetics. It operates on a population of candidate solutions, evolving them over generations to optimize a given objective. The fundamental components of GA include selection, crossover, and mutation, which mimic biological processes to explore and exploit the search space efficiently [37].

The algorithm begins by initializing a population $\mathcal{P}$ of size N, where each individual represents a potential solution encoded as a chromosome. At each generation t, the fitness of each individual in the population is evaluated to determine its quality concerning the objective. A selection mechanism is then applied to choose individuals for reproduction based on their fitness. Common selection methods include roulette wheel selection, tournament selection, and rank-based selection, each ensuring that individuals with higher fitness have a greater likelihood of being chosen while maintaining diversity in the population [37].

Once the parents are selected, the crossover operation is performed to generate offspring by combining the genetic material of the parents. For two parents $\mathbf{p}_1$ and $\mathbf{p}_2$, the offspring $\mathbf{o}_1$ is created via a crossover operation:

$$\mathbf{o}_1 = C(\mathbf{p}_1 + \mathbf{p}_2) \tag{2.54}$$

Mutation introduces variability into the population by randomly altering some genes in the offspring. For a gene g in the offspring chromosome, its mutated value g' can be expressed as:

$$g' = g + \delta, \tag{2.55}$$

where δ is a small random perturbation drawn from a predefined distribution, such as a Gaussian or uniform distribution. Mutation helps the algorithm escape local optima and explore unexplored regions of the search space [4].

After generating the offspring, the population for the next generation $\mathcal{P}^{(t+1)}$ is formed by combining the offspring and selected parents. This process can follow different strategies, such as generational replacement, where the entire population is replaced, or elitist selection, where the top-performing individuals are preserved to ensure convergence [4].

The algorithm iterates over generations until a stopping criterion is met, such as reaching a maximum number of generations or achieving a target fitness level. The final output is the best individual in the population, which represents the optimal or near-optimal solution to the problem. This process is illustrated in Figure 2.11. As shown, the initial population $\mathcal{P}_0$ is generated. As this population is generated randomly, we do now know how good of a solution is each candidate solution in it. So, for each $\mathbf{p}_i^0$, where 0 indicates the order of population – 0 for the starting population, is calculated using the fitness function $\mathcal{F}$. After this, the generational operation starts, with the number of generations equal to 0. Then, we iterate over the population. Then we randomly select one operation to perform – reproduction R copies an existing solution in the population, with the goal of keeping some of the well-performing solutions in the

population going forward, assuring that different solutions are kept. Crossover operation C combines two solutions. Two different strategies were tested – the average and random recombination.

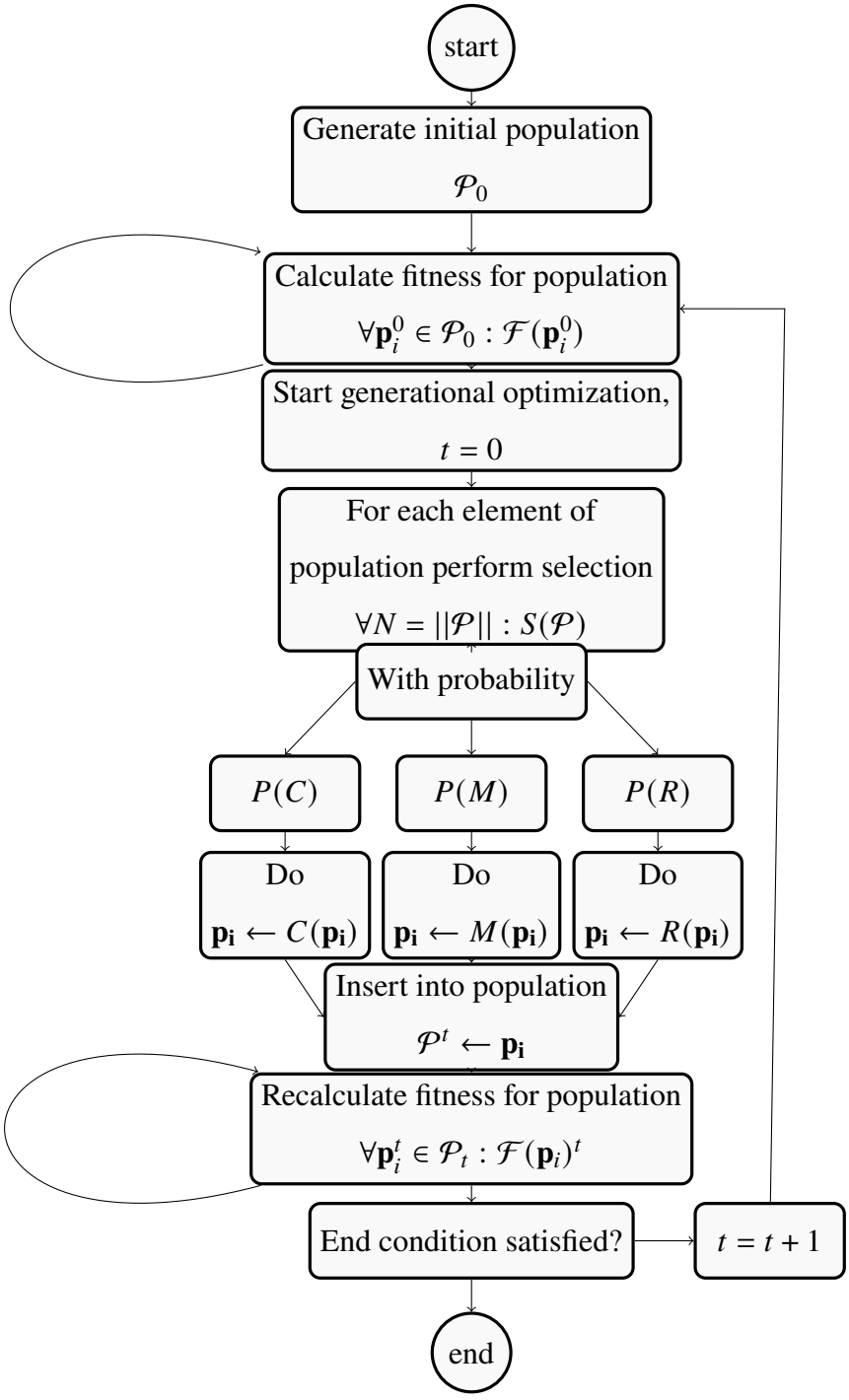


Figure 2.11: The illustration of the GA algorithm.

Let us assume that we have selected two candidate solutions ($\mathbf{p}_A$ and $\mathbf{p}_B$) whose chromosomes are shaped according to Equation 2.49 A and B:

$$A = \begin{bmatrix} a_1^1 & \cdots & a_6^1 \\ \vdots & \ddots & \vdots \\ a_1^6 & \cdots & a_6^6 \end{bmatrix}, B = \begin{bmatrix} b_1^1 & \cdots & b_6^1 \\ \vdots & \ddots & \vdots \\ b_1^6 & \cdots & b_6^6 \end{bmatrix}. \tag{2.56}$$

With these two, the random recombination can be described as iterating over each of the possible values and selecting the value from the matrix A or B with the equal corresponding indexes i and j to insert in the new equation. This means that the output solution Y is selected as:

$$Y = \begin{bmatrix} y_1^1 \in \{a_1^1, b_1^1\} & \cdots & y_6^1 \in \{a_6^1, b_6^1\} \\ \vdots & \ddots & \vdots \\ y_1^6 \in \{a_1^6, b_1^6\} & \cdots & y_6^6 \in \{a_6^6, b_6^6\} \end{bmatrix}, \tag{2.57}$$

with $\{a, b\}$ indicating a selection between two possible values a and b with equal likelihood. This results in a new candidate solution that is the mix of the parameters, offering a wider search range. The other tested recombination manner is the average recombination. Instead of recombining by randomly selecting existing parameters, the new parameters are calculated as the average between the two selected solutions, allowing for a more direct convergence to a solution between them. This can be expressed as:

$$Y = \begin{bmatrix} y_1^1 = \frac{a_1^1 + b_1^1}{2} & \cdots & y_6^1 = \frac{a_6^1 + b_6^1}{2} \\ \vdots & \ddots & \vdots \\ y_1^6 = \frac{a_1^6 + b_1^6}{2} & \cdots & y_6^6 = \frac{a_6^6 + b_6^6}{2} \end{bmatrix}. \tag{2.58}$$

The crossover is performed with the probabilities of $P(C) = [90\%, 95\%]$, and mutation with the probability $P(M) = [1\%, 5\%]$. In the remaining cases, reproduction is performed. These values have been selected based on the previous research on the similar problem [8]. If the obtained solutions show to be better than the current population element according to the fitness function, the current element is replaced. We then sort the solutions to find the best current solution and note it. The end condition in

the presented case is given as the lack of change in the last ten generations or more than 200 generations passing in total. This is also given as algorithm 1.

Algorithm 1 Genetic Algorithm.

Require: Population size N, Crossover probability p_c, Mutation probability p_m, Maximum generations G

Ensure: Best solution found

1: Initialize population P of size N randomly
2: Evaluate fitness of each individual in P
3: **for** $g = 1$ to G **do**
4: Select parent individuals from P based on fitness
5: Create offspring population $P_{\text{offspring}}$ using crossover with probability p_c
6: Mutate individuals in $P_{\text{offspring}}$ with probability p_m
7: Evaluate fitness of each individual in $P_{\text{offspring}}$
8: Select next generation population P from P and $P_{\text{offspring}}$
9: **end for**
10: **return** Best individual from the final population P

2.3.3 Differential evolution

DE is a variant of the evolutionary algorithms, similar to the GA. It functions very similarly to GA, with the same process of creating a population and improving it over generations. The difference between the two algorithms is in the way that the evolutionary part of the algorithm, the recombination of the candidate solutions, is realized. Instead of basing the new solution on two candidate solutions, the DE algorithm bases it on three – a, b and c. Unlike the GA, where the manner in which the selected candidate solutions can be recombined in various ways, in addition to being reproduced and mutated, the only operation the DE performs is the singular recalculation based on the three selected values [61]:

$$y' = a + F \cdot (b - c), \tag{2.59}$$

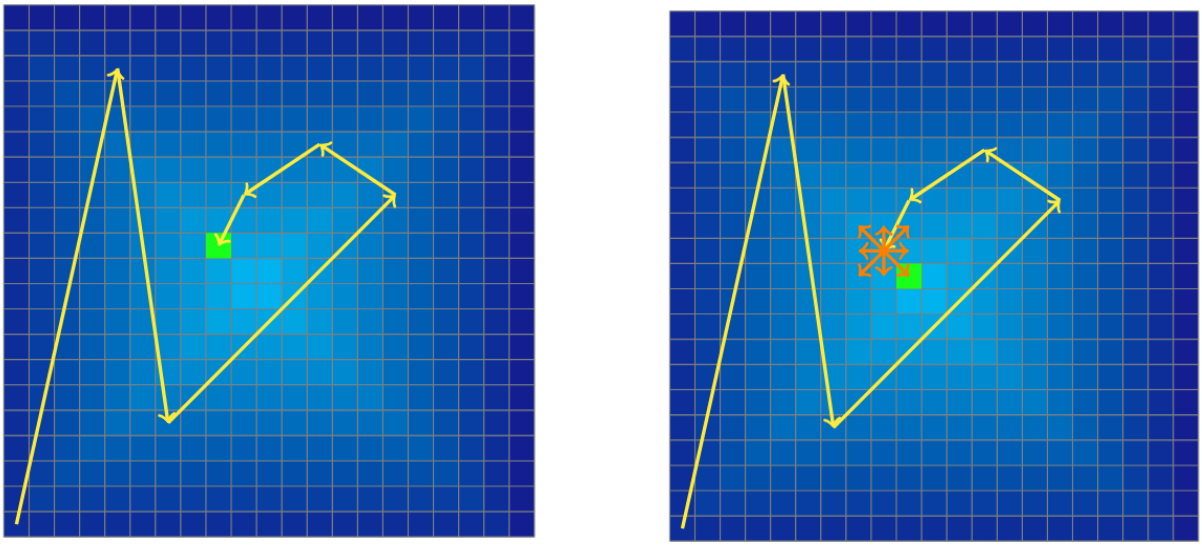
which is performed with the rate equal to the crossover rate in the GA, with the reproduction (propagation of the currently selected solution into the next population) happening otherwise. If the newly formed solution has a better fitness than the current solution the algorithm is iterating over, it is inserted into the population. There is no separate mutation mechanic, as the same function mutation achieves is achieved with the mutation factor F, that multiplies the difference between b and c. It plays a crucial role in controlling the step size of the perturbations applied to candidate solutions, directly influencing the balance between exploration and exploitation. A higher F value (typically in the range $(0, 2]$) increases the magnitude of the differential mutation, allowing the algorithm to explore a broader search space and escape local optima. However, excessively large values can lead to instability and divergence, as solutions may be perturbed too aggressively. Conversely, a smaller F encourages fine-tuned exploitation by making smaller adjustments to candidate solutions, improving convergence stability but increasing the risk of premature stagnation in suboptimal regions. Empirical studies suggest that an adaptive or self-adaptive F, which dynamically adjusts based on the progress of the optimization, can enhance DE's performance by maintaining an optimal balance throughout the search process [61].

2.3.4 Memetic algorithm

MA is an advanced optimization technique that combines the global search capabilities of evolutionary algorithms with local refinement methods to enhance solution quality and convergence speed. This hybrid nature allows MA to address both exploration and exploitation challenges more effectively than traditional methods. By leveraging evolutionary mechanisms to explore the search space and incorporating local search to exploit promising regions, MA achieves a level of efficiency and precision that makes it particularly suitable for solving complex and large-scale optimization problems. The flexibility of MA lies in its ability to integrate domain-specific knowledge and heuristics into its framework, tailoring the algorithm to the characteristics of specific problem domains [47].

The algorithm begins with an initial population $\mathcal{P}$ of candidate solutions, where each individual represents a potential solution to the optimization problem. Similar to the GA, the population is initialized either randomly or through a heuristic process to ensure diversity. At each generation, the population undergoes evolutionary operations, including selection, crossover, and mutation, to generate offspring. However, what distinguishes MA from GA is the incorporation of a local search phase. After the genetic operators are applied, each individual undergoes a refinement process guided by problem-specific information. This local search phase fine-tunes the solutions by iteratively improving them based on a neighborhood exploration mechanism or gradient-based methods, effectively reducing the likelihood of premature convergence and increasing the algorithm's ability to find near-optimal solutions [13].

A defining characteristic of MA is its ability to balance the trade-off between global and local search. The global search component, governed by evolutionary operators, ensures that the algorithm explores a wide area of the solution space, avoiding entrapment in local optima. Conversely, the local search phase targets specific regions of interest, refining solutions within these areas to achieve higher rate of optimization. This dual approach is particularly advantageous in multi-modal optimization problems where the search space contains multiple peaks. The effectiveness of MA hinges on the seamless integration of these two components, as well as the adaptability of the local search strategy to the problem domain. The comparison between how a classic evolutionary algorithm may work (e.g. GA) is shown in Subfigure 2.12a. In the figure, a two-dimensional space is given as a solution space, with each square indicating a possible solution, and lighter squares indicating a better solution, with the best solutions given in the center four squares. The GA performs a wide search, with the example of the best found solution in each generation indicated with yellow arrows, and the best solution found at the end of the algorithm presented with a green square. As shown, GA does not guarantee that the best solution found will be the optimal solution, but merely close to it, indicating that further improvement may be possible. Due to this, when MA is applied to the best possible solution, as shown in 2.12b, the solution found



(a) GA example on two-dimensional search space.

(b) MA example on two-dimensional search space.

Figure 2.12: Illustration of operation between GA and MA, lighter color indicating a better solution, the optimal solution found by algorithm shown with green.

has a local search applied in its surroundings, finding a better solution nearby. Please note that the local search can be applied as shown in the figure – at the end of the algorithm run, or in each step. While slower, applying it in each step can lead to finding of better solutions in each step which will significantly differentiate the path of the algorithm through the search space, and may be beneficial in some cases.

Mathematically, the local search phase can be formalized as follows. Let $\mathbf{p}_i \in \mathcal{P}$ denote an individual in the population. The local search operator, denoted as LocalSearch$(\cdot)$, is applied to $\mathbf{p}_i$ to produce a refined solution $\mathbf{p}'_i$ [47]:

$$\mathbf{p}'_i = \text{LocalSearch}(\mathbf{p}_i), \tag{2.60}$$

where LocalSearch$(\cdot)$ is a function that maximizes a fitness objective $f(\mathbf{p})$ within a defined neighborhood $\mathcal{N}(\mathbf{p}_i)$. Formally, this can be expressed as:

$$\mathbf{p}'_i = \underset{\mathbf{q} \in \mathcal{N}(\mathbf{p}_i)}{\arg\max}\, f(\mathbf{q}), \tag{2.61}$$

where $\mathcal{N}(\mathbf{p}_i)$ is the set of all potential candidates in the neighborhood of $\mathbf{p}_i$. This en-

sures that the refined solution $\mathbf{p}_i'$ is the most fit among the considered neighbors. The different ways in which a local search can be performed will be analyzed in the following subsections. The versatility of MAs makes them applicable to a wide range of problem domains, including different problem domains. Their ability to adapt the local search component to the specific characteristics of the problem at hand allows for significant performance improvements compared to traditional evolutionary algorithms. Furthermore, the incorporation of domain knowledge within the local search phase adds an extra layer of customization, making MAs a highly effective tool for tackling complex optimization tasks. As mentioned, MA is just an upgraded version of GA, with local search added as a step after a candidate solution is found. Besides that, all of the values are equal to the GA, including its hyperparameters. The details on the ways the local search was performed in this research, using exhaustive search and knowledge-based search, are given in the following sections. It has to be noted that while most of the above explanation described MA as an extension of GA, any evolutionary algorithm, including DE can be used as the first, stochastic, part of the algorithm.

Random local search

Ideally, the local search part of the MA would search all of the possible values in the solution space neighboring the candidate solution found by the evolutionary part of the algorithm. While this is possible for simpler solution spaces, the solution space in the presented research is too complex for this. For example, with the discretization step of 0.05, if we wanted to search just five steps around the found solutions space in positive and negative direction this would yield n^{ij}, possible combinations – where n is the number of possible steps, in this case 11 ($I = [\pm 0.25, \pm 0.20, \pm 0.15, \pm 0.10, \pm 0.05, 0]$), and ij is the total dimensionality of the gene chromosome – 36 in the presented case. In other words, exhaustively searching just the local area would require the search of 11^{36}, which is approximately 3×10^{37} solutions. As this is impossible to accomplish in a realistic span of time, a randomized local search is applied. In this case, a randomized sampling is performed according to:

$$Y' = \begin{bmatrix} y_1^1 \pm U(I) & \cdots & y_6^1 \pm U(I) \\ \vdots & \ddots & \vdots \\ y_1^6 \pm U(I) & \cdots & y_6^6 \pm U(I)2 \end{bmatrix}, \tag{2.62}$$

where $U(I)$ represents the random selection of an element of I with uniform probability. This can be performed N times, to search more possible solutions in the neighborhood, with N being set to 1,000 for the presented research.

Informed local search

Unlike random local search which aims at searching the immediate neighborhood of the found candidate solution, informed search works by targeting the elements which may be of significant importance. In the presented research, this means that we will observe how individual paths influence the energy use. This will be done by determining which of the input variables have the highest influence, and then the adjustment will be made to a specific coefficients that may have a direct influence on it. For example, if the coefficients show that the position and speed of the fourth joint have the highest influence on the energy use of the IRM, then the coefficients of that specific joints path may be targeted, lowering them, in order to lower the influence of that particular joint, with the goal of lowering the overall energy use. Four methods will be used to determine the feature importance, and design the cultural algorithm selection – Pearson's, Spearman's and Kendall's correlation, and the Random forests (RF) algorithm. The first three are used to determine the linear, monotonic and non-linear correlation between the variables, while the RF algorithm is used to determine the importance of the variables in the dataset in regards to the targeted output. Pearson's coefficient of correlation is used to determine the linear correlation between an input variable and the energy consumption of the IRM. It is defined on the range of $[-1, 1]$, where -1 indicates a perfect negative linear correlation, 1 indicates a perfect positive linear correlation, and 0 indicates no linear correlation. The formula for the Pearson's coefficient of correlation is given as:

$$\rho^{P}_{X^d_{:,j},Y^d} = \frac{\sum_{i=1}^{n}(x_i - \mu_{X^d_{:,j}})(y_i - \mu_{Y^d})}{\sqrt{\sum_{i=1}^{n}(x_i - \mu_{X^d_{:,j}})^2}\sqrt{\sum_{i=1}^{n}(y_i - \mu_{Y^d})^2}}, \tag{2.63}$$

where $Y^d = [y_1, y_2, \cdots, y_n]$ represents the output variable of the dataset d, with the length n equal to the dimension of j-th input variable $X^d_{:,j}$. To determine the monotonic correlation between the variables, the Spearman's coefficient of correlation is used. It is defined on the same range, and in the same manner, as the Pearson's coefficient. Spearmans's coefficient of correlation is defined as:

$$\rho^{S}_{X^d_{:,j},Y^d} = \frac{\sum_{i=1}^{n}(\varrho(x_i) - \varrho(\mu_{X^d_{:,j}}))(\varrho(y_i) - \varrho(\mu_{Y^d}))}{\sqrt{\sum_{i=1}^{n}(\varrho(x_i) - \varrho(\mu_{X^d_{:,j}}))^2}\sqrt{\sum_{i=1}^{n}(\varrho(y_i) - \varrho(\mu_{Y^d}))^2}}, \tag{2.64}$$

where $R(x_i)$ and $R(y_i)$ are the ranks of the variables within $X^d_{:,j}$ and Y^d, and $\varrho(\mu_{X^d_{:,j}})$ and $\varrho(\mu^d_Y)$ are the mean ranks of the variables. The rank of the variable is the position of the variable in the sorted vector, and the mean rank is the mean value of the ranks.

Finally, the Kendall's coefficient of correlation is used to determine the non-linear correlation between the variables. It is also defined on the range of $[-1, 1]$, as:

$$\rho^{K}_{X^d_{:,j},Y^d} = \frac{2}{n(n-1)}\sum_{i=1}^{n-1}\sum_{k=k+1}^{n}\text{sgn}((x_i - x_k)(y_i - y_k)), \tag{2.65}$$

where sgn is the sign function, which returns 1 if the argument is positive, -1 if the argument is negative, and 0 if the argument is zero.

RF regression is an ensemble learning technique that combines predictions from multiple decision trees to produce a robust and accurate model for continuous target variables. It builds each tree using a random bootstrap sample from the training data, where each sample is drawn with replacement. Additionally, at each split in a tree, a random subset of features is considered, ensuring diversity in the structure of the trees. This randomness, coupled with the ensemble averaging, reduces overfitting and improves the generalization capability of the model [65].

The prediction of a RF regression model is obtained by aggregating the predictions from all the individual trees in the forest. For a single input $(X_{i,:})$, the prediction from

the t-th tree is denoted as $f_t(X_{i,:})$. The overall prediction of the RF for $(X_{i,:})$, denoted as $\hat{y}$, is given by:

$$\hat{y} = \frac{1}{T} \sum_{t=1}^{T} f_t(X_{i,:}), \tag{2.66}$$

where T is the total number of trees in the forest. This averaging mechanism reduces variance, as the ensemble tends to smooth out the errors of individual trees, leading to more stable predictions [65].

Each tree in the forest is constructed using a random bootstrap data point of the data. For the i-th bootstrap sample, let D_i denote the subset of data used to train tree t. At each split in the tree, a random subset of features $F_{split} \subseteq F$ is selected, where F is the set of all features. The split is chosen to minimize the impurity of the resulting child nodes, measured using the variance of the target variable. For a split at node v with left child L and right child R, the reduction in impurity ΔI_m is calculated as:

$$\Delta I_v = I_v - \left(\frac{|L|}{|v|} I_L + \frac{|R|}{|v|} I_R \right), \tag{2.67}$$

where I_v, I_L, and I_R are the variances of the target variable in the parent node v, left child L, and right child R, respectively, and $|v|$, $|L|$, and $|R|$ denote the number of data points in these nodes. The split that maximizes ΔI_v is selected [27].

To ensure robust performance, RF regression uses Out-of-Bag (OOB) data points to evaluate model performance during training. OOB data points are the data points not included in the bootstrap data point for a specific tree. The prediction for an OOB data point is obtained by aggregating the predictions from all trees that did not use it in training. The OOB error is then computed as the mean squared error between the true target values and the OOB predictions, providing an unbiased estimate of the model's performance.

One of the significant advantages of RF is its ability to estimate feature importance, providing insights into which features contribute most to the predictive model. Two widely used methods for computing feature importance in RF are the Mean Decrease in Impurity (MDI) and Feature Permutation Importance.

MDI leverages the reduction in impurity achieved by splits involving a particular feature. Impurity measures, such as Gini impurity for classification tasks or variance for regression tasks, quantify the homogeneity of the target variable within a node. For a given feature $X_{:,j}$, the importance score $I_{MDI}(X_{:,j})$ is computed as:

$$I_{MDI}(X_{:,j}) = \sum_{t=1}^{T} \sum_{n \in \mathcal{N}_t} \Delta I_n \cdot \mathbb{1}(n \text{ splits on } X_{:,j}), \tag{2.68}$$

where T is the total number of trees in the forest, $\mathcal{N}_t$ is the set of nodes in tree t, ΔI_n represents the impurity reduction achieved by node n, and $\mathbb{1}$ is an indicator function that equals 1 if n splits on $X_{:,j}$, and 0 otherwise [27]. The computed importance scores are normalized to facilitate comparison among features.

Feature Permutation Importance is another method that evaluates the impact of each feature on the model's predictive performance. It permutes the values of a feature $X_{:,j}$ randomly, breaking the association between $X_{:,j}$ and the target variable, and measure the resulting decrease in model accuracy. The importance of $X_{:,j}$, denoted $I_{perm}(X_{:,j})$, is calculated as:

$$I_{perm}(X_{:,j}) = \frac{1}{T} \sum_{t=1}^{T} \left(A_{orig}^{(t)} - A_{perm}^{(t)} \right), \tag{2.69}$$

where $A_{orig}^{(t)}$ is the accuracy (or another performance metric) of tree t on the original data, and $A_{perm}^{(t)}$ is the accuracy after permuting $X_{:,j}$. The averaged difference across all trees quantifies the decrease in performance caused by disrupting $X_{:,j}$, indicating its contribution to the model [27].

While MDI provides a computationally efficient measure of feature importance by examining splits during training, it can sometimes be biased toward features with more categories or higher variability. Feature Permutation Importance, on the other hand, directly assesses the impact of a feature on the model's performance and is model-agnostic, but it requires additional computations involving out-of-bag data points or a separate validation dataset.

Both methods provide valuable insights into feature importance, helping practitioners interpret the model and identify the most influential features for decision-making or

further analysis. By comparing the results of these methods, one can gain a comprehensive understanding of the role each feature plays in the predictive model.
The code implementing the MA, with all the parts involving the individual GA parts and the local searches are given in Appendix F.

2.3.5 Summary of presented methods

This chapter served to discuss two main parts of the methodology – the MA optimization approach, together with the supporting approaches for determining the influence of individual features on the output. In addition, multiple possible approaches for determining the fitness function have been discussed – including four different ML approaches for developing a data-driven model, and the LE approach for developing a numerical model. The full realization of the scientific contributions of these described approaches will be given in the "Results and discussion" chapter.

CHAPTER 3

Dataset collection and analysis

This chapter will describe the process in which the experiment was setup for the collection of data within a virtual simulation environment and a real-world, laboratory environment – along with the description of the used equipment. Following that, the process of generating synthetic data will be shown and described. Finally, the statistical analysis will be performed, with a detailed description of used evaluation metrics, and the results will serve to compare the datasets which will be used in the research going forward.

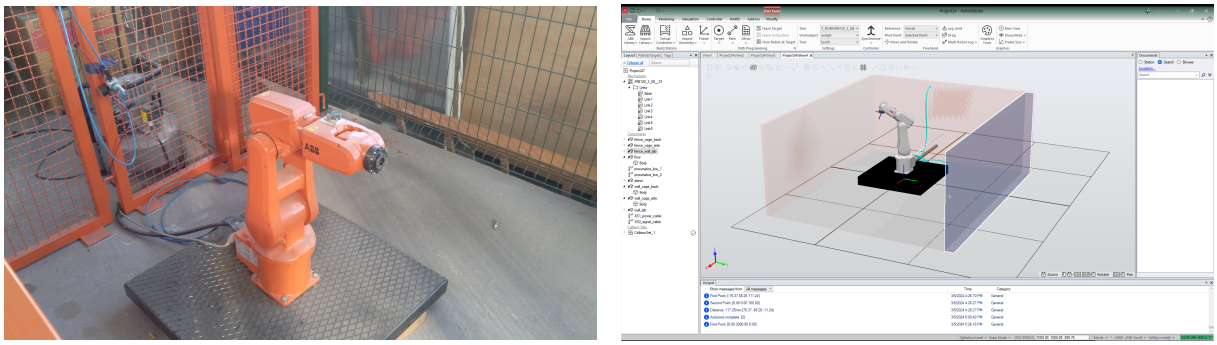
3.1 Simulation setup

The initial setup for the experimental data collection and the data collection within the simulation are the same, which is why they will be described together. For both of the data collection schemes, the setup is performed within the RobotStudio software. The robot used is a ABB IRB 120 industrial robotic manipulator. The IRM in question is a small IRM, with a maximum reach of 0.58 meters, and a maximal handling capacity of 3 kilograms. It is designed as an articulated robot with six axis, and six degrees of freedom. This is achieved by the following joint configuration, where T denotes a T-Type joint (joint which rotates the wrist about the arm axis) and R denoting an R-type joint (joint which rotates perpendicularly to the arm axis): T-R-R-T-R-T – which is a standard configuration of articulated IRMs, when observed from the base to the end-effector. The IRB 120 IRM is powered by electricity, with each of the joints having an individual electric motor. The nominal power consumption of the robot is 0.24 kW [41]. The robot is controlled with an IRC5 controller unit. The controller is connected with two user input devices – a laptop connected over the local network and a FlexPendant

controller.

The first step of the simulation setup process is the creation of a virtual environment for the used IRM, equal to the laboratory setup. This process matters to assure no impacts happen during the simulation. Despite the limits of the of the robot allowing a wider movement, the reality of the laboratory setup and the robot configuration as given limits certain motions of the joints, lowering their realistic range. To simulate the laboratory environment the robot is placed on the 800.00 mm by 800.00 mm base that is 100 mm tall. Below the base, a 2000,00 mm by 2400,00 mm floor object is added. This object has no height, and serves only for collision detection. Then, three fences are added surrounding the base of the IRM, with the height of 1000,00 mm. The IRM sits on the middle pedestal which is removed from the left wall 416 mm, the right wall by 786 mm and the back wall by 378 mm. The floor plan of the laboratory is given in Appendix A.

The described environment is designed in the RobotStudio software package [43] which will be used for the data collection process – both within the simulation and the laboratory environment. The comparison between the real and simulated laboratory environements are shown in Figure 3.1, with the laboratory environment as it appeared during the data collection procedure (March 2024) shown in subfigure 3.1a, and the created simulated environment shown in subfigure 3.1b. The figure shows that the specific elements and textures weren't replicated perfectly. This was not the goal of environment modeling. The goal was to simply set the objects in the same position as the real environment, as in this way they can serve as collision detection. This allows for testing of various positions of the used IRM, in order to determine the real ranges in which the movement can be performed without any danger of collision between the robot and the environement.



(a) Laboratory environment

(b) Simulated environment

Figure 3.1: Comparison between the laboratory and the simulated environment.

In addition to the environment elements shown in the floor plan, Subfigure 3.1a shows that there are power and signal cables, as well as pneumatic lines connected to the used IRM. These elements were also added to the simulation, in order to avoid and potential snags due to their position in the environment. Then, the IRM was moved in the simulation environment, by each individual joint, testing a multitude of positions to determine which ranges of movement will not results in any possible collision or a near miss between the IRM and other elements. The determined ranges are given in Table 3.1.

Table 3.1: Comparison of the nominal joint range, as given by the manufacturer, and the joint range that was determined for use in simulation and laboratory data collection. Joints numbered from base towards the end-effector in ascending order.

Joint	Nominal Joint Range		Actual Joint Range	
	Minimum [°]	Maximum [°]	Minimum [°]	Maximum [°]
1	-165	+165	-90	90
2	-110	110	-10	45
3	-110	70	-100	40
4	-160	160	-160	160
5	-120	120	-120	120
6	-400	400	-400	400

3.2 Data collection program in RAPID

The data collection procedure was realised within the RAPID programming language – a proprietary functional programming language created by ABB for the purpose of automating robotic tasks [44]. RAPID programs are written as modules. While no parallelization of features is allowed within a module, multiple modules can be added to the controller, and ran concurrently – if the virtual controller supports parallel execution. To allow for the data collection process to be executed while the robot the data is being collected from is in motion, this parallelization feature is used. As shown in the Figure 3.2, the robot controller (virtual – within the simulation or a real one) controls the execution of the robot. It is directly controlling the robot through the instructions generated within the control module. Meanwhile, the relevant measurements from the robot are collected within the measurement module. These values are stored within a comma-separated values (CSV) file.

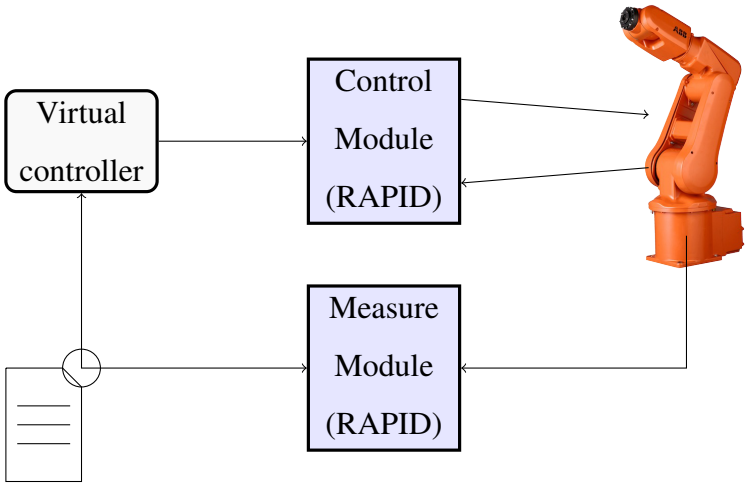


Figure 3.2: The two modules used for data collection.

First, the control module needs to be discussed, in order to understand how the operation of the robot is controlled in order to obtain the measurements. The goal of the code is to generate a random position in the joint space of the industrial robotic manipulator, or in other words, get six random values within the ranges provided in Table 3.1,

and then move the robot from the current position to the other. This process should be repeated 1,000 times. In other words, the algorithm for obtaining the dataset can be defined as:

Algorithm 2 Pseudocode of the algorithm for generating a random value of path.

$n \leftarrow 1$

$i \leftarrow 1$

$joint_{higher}, joint_{lower} \leftarrow$ array of joint limits

$RAND\_MAX \leftarrow 32767$

$V \leftarrow$ array of possible speeds, $N_{speeds} \leftarrow$ length of speed array

$Z \leftarrow$ array of possible zones, $N_{zones} \leftarrow$ length of zone array

while $n \leq 1,000$ **do**

  for $i \leq 6, i++$ **do**

    $R_i \leftarrow Random()/RAND\_MAX * (joint_{higher}[i] - joint_{lower}[i]) + joint_{lower}[i]$

  end for

  $J = [R_1 \cdots R_6, V[R/RAND\_MAX * N_{speeds}], Z[R/RAND\_MAX * N_{zones}]]$

  Move Robot to J

  n+=1

end while

Regarding the code itself, first, the constants need to be defined. These variables include the joint ranges (as already noted, given in Table 3.1) – an example of this is given in the Listing 3.1, as well as the possible values of speed and zoning. The '...' symbol in the listing is used to skip through similar, repetitive code.

```
1  ! Define the upper and lower bounds of joints,
2  ! grabbed by observing kinematics
3  CONST num J1_LO := -120;
4  CONST num J1_HI := 120;
5  ...
6  CONST num J6_LO := -400;
7  CONST num J6_HI := 400;
```

Listing 3.1: Setting joint value limits

The parameter speed controls the speed of the tool center point (TCP) along the defined path. The speed is given as vX, where X represents the desired speed, in mm/s. For example, '*v150*' means that the target speed is 150 mm/s; while '*v3000*' means that the target speed is 3000 mm/s or 3 m/s. The actual TCP speed may vary, due to the path planning considerations [43]. The zone parameter refers to the precision, or how close should the TCP path follow the pre-calculated path between the assigned points. Smaller zone value means that the robot will follow the path more precisely, but will slow down – especially in corners, leading to slower execution. A larger zone allows for smoother, faster motion by rounding corners or blending paths. In addition to the value definition similar to speed with zX, it is possible to assign the value of '*fine*'. If positioning is given as 'fine' the robot stops precisely at the programmed point, ensuring maximum accuracy – no blending occurs between this and the following point. If the value is provided with zX then X indicates the amount of deviation (in millimeters) allowed to be used for path smoothing. In the provided simulation, random variables of these two parameters are selected in order to simulate as many different applications and path programming approaches as possible. To achieve this, the possible values are stored in an array. The array of the values needs to have the number of values defined at the start. This can be seen in the Listing 3.2, which shows the array containing the possible values of the zone parameter. Do note that the '*ZONE_NUMBER*' could be given explicitly as a number of values when defining the array, but in the given example it was defined separately. In addition to that, note that the specific variable type for zone is '*zonedata*', which is an internally defined RAPID variable type for zone information – speed information has the equivalent type defined as '*speeddata*'.

```
1  CONST num ZONE_NUMBER:=14;
2  CONST zonedata ZONE_ARR{ZONE_NUMBER}:=[fine, z0, z1, z5, z10, z20, z30,
3                                         z40, z50, z60, z80, z100, z150,
   z200];
```

Listing 3.2: Example of defined array of zones

In addition to the defined constant values, the variables need to be defined. The defined

variables are the target joint positions (*J1*, *J2*, *J3*, *J4*, *J5*, and *J6*), the speeddata V and zonedata Z as holders for the appropriate values, along with some helper string variables for storing the execution times, as shown in Listing 3.3.

```
1  !Define variables for storing generated random values
2  VAR num J1;
3  ...
4  VAR num J6;
5  VAR speeddata V;
6  VAR zonedata Z;
7
8  VAR string starttime;
9  VAR string endtime;
```

Listing 3.3: Defining variables.

The control of the robot happens going forward, within he main process (defined as '*PROC main()*') of the module. We perform the iteration for the amount of ties, defined by the constant *SIMULATION_COUNT*, using a FOR loop. We perform the random selection of the joint value within the ranges given as constants (defined as shown in Listing 3.1), according to Algorithm 1. This is done using the '*RAND*' command. This command uniformly randomly selects the value between 0 and 32767 (2^{15}). To use this to get a real value, first the value is defined with the maximum possible value, normalizing it to the range of $[0, 1]$. This value is multiplied with the posible range of the joint value J. To move the randomly selected value into the range of the joint, the lower value of the joint is added to it, per:

$$j \sim \min(J) + \frac{U(0, 32767)}{32767} \cdot [\max(J) - \min(J)], \tag{3.1}$$

where U indicates a uniform selection.

The same process is done for the selection of values from '*SPEED_ARR*' and '*ZONE_ARR*' arrays of values. Then, these values are used with the command '*MoveAbsJ*'. The command '*MoveAbsJ*' takes four parameters. The second and third parameter are the previously defined values of speed and zone, randomly selected from the variable ar-

ray containing possible values. It should be noted that the values could be selected directly here, using the defined command – but the variables '*V*' and '*Z*' are used for readability. The tool parameter (given as '*tool0*') allows the command to calculate the TCP, which is necessary to achieve the target speed and zone defined with the previous two parameters. In other words, this is a parameter used in internal path planning used for moving the joints into the position given as the first parameter. This position is defined with a two dimensional array. The first sub-array of six values contains the target values of robot joints J1 through J6. The second sub-array is used to control the external axes (e.g. tilt tables, conveyor axes, gripper axes, etc.). Up to six of these axes can controlled. In case an axis is not present, the convention of the RAPID language dictates that its value should be set to 9×10^9 ('*9E9*' in RAPID), with this value indicating that no axis is present. This loop is shown in the Listing 3.4

```
1  FOR i FROM 0 TO SIMULATION_COUNT DO
2
3  !Randomly select joint 1--6, speed and zone
4  J1 := ((RAND()/RAND_MAX)*(J1_HI-J1_LO))+J1_LO;
5  ...
6  J6 := ((RAND()/RAND_MAX)*(J6_HI-J6_LO))+J6_LO;
7
8  V := SPEED_ARR{1+ROUND((RAND()/RAND_MAX)*(SPEED_NUMBER-1))};
9  Z := ZONE_ARR{1+ROUND((RAND()/RAND_MAX)*(ZONE_NUMBER-1))};
10
11 MoveAbsJ [[J1,J2,J3,J4,J5,J6],
12             [9E9,9E9,9E9,9E9,9E9,9E9]],V,Z,tool0;
13
14 ENDFOR
```

Listing 3.4: Main loop for moving the robot in the randomly selected position.

The second module deals with obtaining the values from the industrial robot unit. These values include: position (in degrees), speed (in /s), and joint torque (in Nm) for joints 1 through six; and x, y and z position (in mm), the angle defined via quarterions q_1, q_2, q_3, and q_4, and the angle defined via Euler angles ψ, θ, ϕ for TCP. Quaternions and

Euler angles are two mathematical representations used to describe rotations in three-dimensional space. While Euler angles express rotations through a sequence of angles about coordinate axes, quaternions provide a compact and computationally efficient representation that avoids singularities and ambiguities inherent in Euler angles [10]. A quaternion is a four-dimensional extension of complex numbers and is represented as:

$$q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}, \tag{3.2}$$

where w is the scalar part, x, y, z are the vector components, and $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are the imaginary unit vectors satisfying:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1. \tag{3.3}$$

For unit quaternions, which are often used to represent rotations, the constraint is:

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1. \tag{3.4}$$

A rotation in three-dimensional space can be expressed using a quaternion q derived from the axis-angle representation. For a rotation by an angle θ about a unit axis $\mathbf{u} = (u_x, u_y, u_z)$, the quaternion is given by:

$$q = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)(u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}). \tag{3.5}$$

Euler angles, on the other hand, represent rotations using three angles (ϕ, θ, ψ), corresponding to rotations about fixed axes. A common convention is the ZYX order, where the angles correspond to rotations about the z-axis, y-axis, and x-axis, respectively. The composite rotation matrix R for Euler angles is given by:

$$R = R_z(\phi)R_y(\theta)R_x(\psi), \tag{3.6}$$

where $R_z(\phi)$, $R_y(\theta)$, and $R_x(\psi)$ are the rotation matrices for individual axes:

$$R_z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \quad R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix}. \tag{3.7}$$

Quaternions can be converted to Euler angles and vice versa. For a quaternion $q = (w, x, y, z)$, the Euler angles (ϕ, θ, ψ) in the ZYX convention are computed as:

$$\phi = \arctan 2(2(wx + yz), 1 - 2(x^2 + y^2)), \tag{3.8}$$

$$\theta = \arcsin(2(wy - zx)), \tag{3.9}$$

$$\psi = \arctan 2(2(wz + xy), 1 - 2(y^2 + z^2)). \tag{3.10}$$

Conversely, a quaternion can be derived from Euler angles using:

$$q = \cos\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right)\mathbf{i} + \ldots, \tag{3.11}$$

with additional terms for the **j** and **k** components following a similar pattern. While Euler angles are intuitive and easy to interpret, they suffer from gimbal lock, a condition where the rotation axes become degenerate. Quaternions, being free from such singularities, are widely used in applications requiring smooth and continuous rotations [10]. In addition to numerical variables used to store the joint positions, speeds and torques, some additional variables, and variable types, are necessary, as shown in Listing 3.5.

```
1  VAR NUM POS1;
2  VAR NUM SPD1;
3  VAR NUM TOR1;
4  ...
5  VAR NUM POS6;
6  VAR NUM SPD6;
7  VAR NUM TOR6;
8
9  VAR IODEV LOGFILE;
10
11 VAR robtarget TCP;
12 VAR clock sim_clock;
13 VAR num time;
```

Listing 3.5: Example of writing the values.

As the code snippet shows, the file is defined via the variable type '*IODEV*'. The desired information about the position and orientation of the TCP is stored in the variable

type '*robtarget*'. The clock variable, with the type of the same name is needed as the simulation time may not equal real time. The simulation wthin the RobotStudio package can be sped up by a certain factor provided that the hardware of the workstation the simulation is executed on allows this. The sped up simulation will physically perform identically to a real-world one, with the benefit of the measurement being completed sooner. In case when this code is used on the real controller and robot, the clock variable will simply track the real time. The output value of this will be stored in the variable time. In the start of the main process the logfile location and name is defined with command '*Open*'. The file name is formatted as:

< Date of measurement > _ < Hour > _ < Minute > _ < Second > _MEASUREMENT.CSV,

with the time being the start of the simulation. In the opened logfile, using the '*Write*' command, the header containing the names of variables that will be stored in each column of the CSV file are written, per code snippet shown in Listing 3.6.

```
1 Open "HOME:" \File:=CDate()+"-"+
2     NumToStr(GetTime(\Hour),0)+"-"+
3     NumToStr(GetTime(\Min),0)+"-"+
4     NumToStr(GetTime(\Sec),0)+
5     "_MEASUREMENT.CSV", logfile \Write;
6
7 Write logfile, "t,q1,dq1,tau1,q2,dq2,tau2,"\NoNewLine;
8 ...
9 Write logfile, "x,y,z,e1,e2,e3,e4,psi,theta,phi";
```

Listing 3.6: The opening of the logfile which stores the measured data

Then, to assure that there are no mistakes or issues with the clock, it is reset and restarted, before entering the data collection loop. In this loop, the data for each of the six joints is collected and stored in the appropriate variable. In addition to this, the data for TCP position and orientation is stored in a variable along with time, per Listing 3.7

```
1 GetJointData \MechUnit:=ROB_1, 1
```

```
    \Position:=POS1
    \Speed:=SPD1
    \Torque:=TOR1;
...
GetJointData \MechUnit:=ROB_1, 6
    \Position:=POS6
    \Speed:=SPD6
    \Torque:=TOR6;

TCP := CRobT(\Tool:=tool0 \WObj:=wobj0);

time := ClkRead(sim_clock);
```

Listing 3.7: The commands for measuring the values from the robot unit

The values for position and orientation are obtained from the TCP variable at the time of writing to the logfile – the x, y, and z psitions are obtained from the TCP object '*TCP.trans.x*', while the quarternions are obtained with '*TCP.rot.q1*'. The Euler angles are calculated using the '*EulerZYX*' command, which takes two values – the desired angle (given as '*X*', '*Y*', or '*Z*'), and the '*TCP.rot*' object. All of the numerical values are converted to strings, with five decimal places of precision. These strings are joined together, separated with commas, using the '+' operator. The execution of the code is paused, until 0.025 seconds have elapsed, to assure the measurement frequency of 40Hz. This value was selected to balance the amount of collected datapoints and their variety, with the amount of time long enough to perform the measurements. The time was mostly influenced by the amount of measurements desired for further processing and model creation, as a higher frequency may result in a large number of similar datapoints, especially in those path segments that use slower speeds. The complete program codes for both modules are given in Appendix D.

As the goal is to model the energy of the individual joint, as well as the full energy used to move the whole IRM, additional values need to be calculated. These value include the output values of energy – both per each joint and the total used energy. In addition o this, some additional supporting values are calculated in order to potentially assist the

simplicity of modeling using the previously detailed techniques. Additional variables, even if they are derived from the existing ones, can have a more direct influence on the output, easing the process of model training. These values are:

- the measurement time between two data points Δt,
- acceleration of each axis $\omega_i \forall i \in [1, \cdots, 6]$,
- the change in linear position of TCP since the last measurement $\Delta x, \Delta y$, and Δz,
- the change in angle of the TCP $\Delta\phi, \Delta\theta$, and $\Delta\psi$,
- the linear components of TCP speed and acceleration $\Delta\dot{x}, \Delta\dot{y}$, and $\Delta\dot{z}$, and
- the angular speeds and accelerations $\Delta\dot{\phi}, \Delta\dot{\theta}$, and $\Delta\dot{\psi}$.

The calculation of these values is done within using Python, namely the Pandas library. The measurement time between two data points is calculated by subtracting the time of the first data point t_1, from the second data point $t_2 - \Delta t = t_2 - t_1$. The changes in position and speeds are calculated in the same manner, subtracting the previous measurement value from the current one. While this sacrifices a single, first, datapoint, considering a large amount of data points in the dataset, this does not influence the measurement in any significant way. Final thing to note is that the energy cannot be directly written for individual joints in the robot studio. What can be written is the individual joint torque $\tau_i \forall i \in [1, \cdots, 6]$, which can then be used to calculate the momentary power and the energy as a product of it with speed.

3.3 Statistical analysis and comparison between datasets

One of the key indicators of the quality of simulation or synthetic datasets is the similarity between the datasets. Since the research presented in this thesis includes both simulated synthetic and statistically generated synthetic data, comparisons of this type are extremely important. To achieve a good performance on real validation data when

the model has been trained using synthetic data, the training set should be statistically similar to the original data [28]. For this reason three analyses were performed to test the data similarity and determine if it is satisfactory – calculation and comparison of descriptive statistics between datasets, variable pair comparisons and distribution comparisons. Descriptive statistics are a set of brief coefficients and metrics that summarize a given dataset – usually describing the central tendency of the set and its spread. The central tendency refers to the point in which the main part of the distribution is centered – or, in other words, where the probability density function (PDF) is at its maximum, while the spread indicates how wide is the domain of PDF for a given set of data. The metrics used for descriptive statistics in this paper are mean, median, mode, variance, standard deviation, range (minimum and maximum values), kurtosis and skewness [31].

First, we will define symbols which will be used throughout the description. The dataset will be defined as X. Each variable within the dataset (a column), will be defined as a vector $X_{:,j}$. If $X_{:,j}$ consists of N elements per $X_{:,j} = [x_{1,j}, X_{2,j}, \cdots X_{N,j}]^T$. The number of elements N is equal for each feature $X_{:,j}$. Then, the dataset can be noted as $X = [X_{i,1}, X_{i,2}, \cdots X_{i,m}]$, where m is the number of variables within the dataset.

The mean value of the variable $X^d_{:,j}$ where j indicates the variable and d indicates the dataset, is given as:

$$\mu_{X^d_{:,j}} = \frac{1}{n} \sum_{i=1}^{n} X^d_{i,j}. \tag{3.12}$$

While mean provides information to the central tendency of the variable, it is extremely sensitive to the extreme values (outliers) of the dataset, as only a few extreme values can significantly change the mean of the variable. For that reason, mode and median are also calculated, as it allows us to gain insight into the skewness and spread of the variable – e.g. a mean greater than the median indicates a positively skewed variable. Median of the variable represents the value that is found in the middle of the sorted set. If (X^d) indicates the set X^d sorted in an ascending order, median is defined as:

$$M_{X^d} = \begin{cases} X_{i,j}^{d\frac{n+1}{2}}, & \forall n \mod 2 = 0, \\ \frac{X_{i,j}^{d\frac{n}{2}} + X_{i,j}^{d\frac{n}{2}+1}}{2}, & \forall n \mod 2 = 1 \end{cases} \tag{3.13}$$

The mode m of the variable represents the value of the variable that occurs most frequently. A variable can be unimodal – if it only has a single value that appears most often, or multimodal – if there are more values that appear wth the largest frequency. Alternatively, a variable can be nonmodal, if it does not posses a mode (frequency $f_{X_{i,:}^d} = 1 \forall i$). Mode is obtained algorithmically, by creating a dictionary of values in the variable vector, and increasing the value of the dictionary for each value that is found during iteration through the vector. The value with the largest dictionary value is the mode of the variable.

The above define the metrics for the central tendency of the variable. But, a more important metric, especially when discussing comparison to the snythetic and simulation data is the dispersion of the data. If the original data is significantly more dispersed than the generated data, the model trained on it will have issues predicting the values which were not contained within the original dataset. The variance of the variable is defined as:

$$V_{X_{:,j}^d} = \frac{1}{n} \sum_{i=1}^{n} (X_{i,j}^d - \mu_{X_{:,j}^d})^2. \tag{3.14}$$

A commonly used expression of the variance is a standard deviation. It is defined as the square root of the variance:

$$\sigma_{X_{:,j}^d} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (X_{i,j}^d - \mu_{X_{:,j}^d})^2}, \tag{3.15}$$

and it provides a more precise measure. Generally, the lower value indicates less spread (lower dispersion) of the data around the mean, while the higher dispersion in regards to the central tendency. The standard deviation is commonly compared to the range – the difference between the maximum and minimum value of the variable:

$$\lozenge_{X^d_{:,j}} = \max(X^d_{:,j}) - \min(X^d_{:,j}). \tag{3.16}$$

If the range is much higher than the range indicated by the standard deviation, this is a good indicator of statistical outliers within the dataset. In the context of generated data, the ranges should be similar, if not equal to the original data, to assure that the model trained on the generated data has the ability to predict all possible values.

Final two descriptive statistics used are skewness and kurtosis. The skewness measures the asymmetry of a distribution, where a value of 0 indicates a fully simetrical distribution. If the value of skewness is positive, it indicates a PDF with a tail on the right side (a positively/right skewed), while a negative value indicates a negatively/left skewed distribution. Skewness Sk is defined as:

$$Sk_{X^d_{:,j}} = \frac{\frac{1}{n}\sum_{i=1}^{n}(X^d_{i,j} - \mu_{X^d_{:,j}})^3}{\sigma^3_{X^d_{:,j}}}. \tag{3.17}$$

Finally, kurtosis measures the degree of outliers in the distribution. It indicates whether the data has heavy tails or if it's concentrated around the mean. A value of 3 indicates a normal distribution, while a value greater than 3 indicates a leptokurtic distribution (heavy tails), and a value less than 3 indicates a platykurtic distribution (light tails) Kurtosis K is defined as:

$$K_{X^d_{:,j}} = \frac{\frac{1}{n}\sum_{i=1}^{n}(X^d_{i,j} - \mu_{X^d_{:,j}})^4}{\sigma^4_{X^d_{:,j}}}. \tag{3.18}$$

3.4 Synthetic data generation

3.4.1 Generating synthetic data using Copulas

The gaussian copula is a powerful statistical framework for modeling dependencies among random variables, widely used in areas like finance, risk analysis, and, importantly for the presented research, synthetic data generation. It builds on the concept

of a copula, which is a function that joins univariate marginal distributions to form a multivariate distribution. By separating the dependency structure from the marginal distributions, the gaussian copula allows for flexibility in modeling multivariate data with varying marginal behaviors. This separation makes it particularly suitable for scenarios where different variables follow distinct distributions but exhibit some level of interdependence [62].

Mathematically, let $X = (X_{:,1}, X_{:,2}, \ldots, X_{:,d})$ be a random vector with joint cumulative distribution function (CDF) $F_X(X_{i,1}, X_{i,2}, \ldots, X_{i,d})$ and marginal CDFs $F_{X_{i,j}}(X_{i,j})$ for $i = 1, 2, \ldots, d$. The gaussian copula models the joint CDF as:

$$F_X(X_1, X_2, \ldots, X_d) = C\left(F_{X_1}(X_{i,1}), F_{X_2}(X_{i,2}), \ldots, F_{X_d}(X_{i,d})\right), \tag{3.19}$$

where C is the copula function. For the gaussian copula, C is defined as:

$$C(u_1, u_2, \ldots, u_d) = \Phi_\Sigma\left(\Phi^{-1}(u_1), \Phi^{-1}(u_2), \ldots, \Phi^{-1}(u_d)\right), \tag{3.20}$$

where $u_i = F_{X_{i,j}}(X_{i,j})$ are uniform random variables on $[0, 1]$, Φ^{-1} is the quantile function (inverse CDF) of the standard normal distribution, and Φ_Σ is the CDF of a multivariate normal distribution with mean vector $\mathbf{0}$ and covariance matrix Σ. The matrix Σ encodes the dependency structure among the variables and is typically parameterized by a correlation matrix $\mathbf{R}$, with entries ρ_{ij} representing the Pearson correlation coefficients between variables $X_{i,j}$ and $X_{:,j}$.

The practical application of the gaussian copula begins with fitting it to empirical data. This involves two primary steps: (1) estimating the marginal distributions for each variable, and (2) determining the correlation structure. The marginal distributions can be modeled parametrically, such as using normal or exponential distributions, or non-parametrically through kernel density estimation. The dependency structure is captured by the correlation matrix $\mathbf{R}$, which is estimated from the transformed data. Once the gaussian copula model is fit, synthetic data can be generated by sampling from the copula.

To generate synthetic data, one starts by sampling from a multivariate normal distribu-

tion with mean vector $\mathbf{0}$ and covariance matrix Σ. Let $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_d)$ represent the sampled vector. Each component Z_i is then transformed to the uniform scale using the standard normal CDF Φ:

$$U_i = \Phi(Z_i), \quad i = 1, 2, \ldots, d. \tag{3.21}$$

These uniform variables are subsequently transformed to match the original marginals by applying the inverse CDFs:

$$X_{i,j} = F^{-1}_{X_{i,j}}(U_i), \quad i = 1, 2, \ldots, d. \tag{3.22}$$

The resulting X constitutes a synthetic dataset that mirrors the statistical properties of the original data, including the dependency structure encoded in Σ and the marginal distributions.

The gaussian copula's primary strength lies in its ability to model complex dependency structures while preserving the marginal characteristics of the data. However, its reliance on the multivariate normal distribution introduces limitations. One key limitation is its inability to accurately capture tail dependencies, particularly in cases involving extreme values. For instance, in financial modeling, where joint extreme events (e.g., simultaneous market crashes) are of significant interest, the gaussian copula may underestimate the likelihood of such co-occurrences due to its symmetric dependency structure. This has motivated the development of alternative copulas, such as the t-Copula, which accounts for heavy tails and exhibits stronger tail dependence [62].

Despite these limitations, the gaussian copula remains a foundational tool in multivariate modeling due to its computational efficiency, interpretability, and flexibility. Its ability to model dependencies across a wide range of applications, from synthetic data generation to stress testing in finance, underscores its versatility. Further research has extended its capabilities, integrating it with ML methods to better capture non-linear dependencies and improve robustness in high-dimensional settings [59]. In summary, the gaussian copula provides a powerful framework for modeling dependencies in multivariate data while maintaining individual marginal properties. Its theoretical foundations

and practical implementation continue to make it a cornerstone of statistical modeling and data analysis.

3.4.2 Generative artificial network approach to tabular data generation

The Conditional Tabular Generative Adversarial Network (CTGAN) is a sophisticated generative model designed specifically to address the challenges of generating synthetic tabular data. Unlike traditional generative adversarial networks (GANs), which struggle with the complexities of tabular data due to its mixed data types and inherent statistical characteristics, CTGAN introduces innovations tailored for this domain. Tabular data often includes both continuous and categorical variables, with categorical variables exhibiting highly imbalanced distributions. Moreover, relationships between features can be non-linear and involve intricate dependencies. CTGAN resolves these issues through a combination of conditional sampling, mode-specific normalization, and specialized training techniques, providing a robust framework for generating realistic synthetic tabular datasets [59].

The architecture of CTGAN comprises two key components: the generator G and the discriminator D. The generator takes as input a noise vector $Z \sim p_Z$ sampled from a prior distribution, typically a standard Gaussian, and produces a synthetic data point $G(Z)$. The discriminator, on the other hand, aims to distinguish between real data points $X \sim p_{\text{data}}$ and synthetic data points generated by G. The model is trained in an adversarial setting, where G attempts to generate realistic data to fool D, and D works to improve its classification of real versus synthetic data points. This adversarial training framework can be expressed through the minimax objective:

$$\min_G \max_D \mathbb{E}_{X \sim p_{\text{data}}}[\log D(X)] + \mathbb{E}_{Z \sim p_Z}[\log(1 - D(G(Z)))], \tag{3.23}$$

where p_{data} represents the real data distribution and p_Z the noise distribution. To handle the unique challenges of tabular data, CTGAN extends this framework by introducing a conditional vector that guides the generation process. This vector encodes specific categories or ranges of values for certain features, enabling the generator to

produce data that accurately reflects the conditional distributions present in the real dataset. This approach is particularly effective in addressing class imbalance, as it ensures that even underrepresented categories are modeled appropriately.

Another key innovation in CTGAN is mode-specific normalization, which is applied to continuous features. Continuous variables in tabular data often exhibit multimodal distributions, making standard normalization techniques inadequate. CTGAN addresses this by clustering the values of a continuous feature into distinct modes and normalizing each mode independently. Let $X_{:,j}$ represent a variable, and assume that it belongs to one of M modes. For a data point $X_{i,j}$ from mode m, the normalized value is computed as:

$$X_{i,j}^{(m)} = \frac{X_{i,j} - \mu_m}{\sigma_m}, \tag{3.24}$$

where μ_m and σ_m are the mean and standard deviation of mode m, respectively. This normalization ensures that the generator learns each mode separately, preserving the original feature's multimodal characteristics in the synthetic data.

CTGAN also incorporates logits-based representation for categorical features. Instead of representing categorical variables through one-hot encoding, which can lead to sparsity and gradient vanishing issues, CTGAN uses a continuous representation of the logits before applying the softmax function. This strategy enhances the generator's ability to capture complex interactions between categorical and continuous features, improving the quality of the synthetic data.

To further stabilize training and improve the quality of generated data points, CTGAN employs the Wasserstein GAN with gradient penalty (WGAN-GP) objective. Unlike the original GAN objective, which relies on the Jensen-Shannon divergence, WGAN-GP uses the Wasserstein distance as a measure of divergence between real and synthetic data distributions. The Wasserstein distance provides smoother gradients, making the optimization more stable and less prone to mode collapse. The gradient penalty term ensures that the discriminator function D satisfies the 1-Lipschitz constraint, as required by the Wasserstein GAN framework. The updated objective function is:

$$\mathcal{L} = \mathbb{E}_{Z\sim p_Z}[D(G(Z))] - \mathbb{E}_{X\sim p_{\text{data}}}[D(X)] + \lambda \mathbb{E}_{\hat{X}\sim p_{\hat{X}}}\left[(\|\nabla_{\hat{X}} D(\hat{X})\|_2 - 1)^2\right], \tag{3.25}$$

where $\hat{X}$ represents interpolated data points between real and synthetic data, and λ is a regularization parameter controlling the gradient penalty.

The training process for CTGAN alternates between optimizing the generator and the discriminator. During each iteration, the generator data points a noise vector Z and a conditional vector that specifies the feature values to be generated. These inputs guide the generator in producing a synthetic data point. The discriminator then evaluates the data point along with real data, providing feedback that helps the generator improve its outputs. Over successive iterations, the generator learns to approximate the true data distribution, resulting in high-quality synthetic data that preserves the statistical properties and dependencies of the original dataset.

CTGAN has demonstrated superior performance in generating realistic synthetic tabular data compared to traditional methods and other GAN-based models. It is particularly effective in scenarios involving high-dimensional data, imbalanced categorical features, and complex feature interactions. By maintaining the statistical integrity of the data while ensuring diversity, CTGAN enables the safe and effective use of synthetic data in a variety of applications, including ML model training, data augmentation, and privacy-preserving data sharing [58].

3.4.3 Combining copulas and generative networks for data generation

The CopulaGAN builds upon the foundational ideas of generative adversarial networks (GANs) and introduces a copula-based mechanism to explicitly model dependencies between variables in tabular data. Unlike CTGAN, which primarily relies on conditional sampling and mode-specific normalization to handle mixed data types and complex relationships, CopulaGAN leverages copulas to directly capture the joint dependency structure between features. This integration of statistical copulas into the GAN framework allows CopulaGAN to model non-linear and potentially asymmetric dependencies more effectively [59].

At the core of CopulaGAN is the transformation of the original data into a copula space, where the marginal distributions of each variable are uniform over the interval $[0, 1]$. This is achieved by applying the CDF of each feature to the original data. Let $X = (X_1, X_2, \ldots, X_d)$ represent the tabular data, and let $F_{X_{i,j}}(X_{i,j})$ denote the marginal CDF of the i-th variable. The transformed data in copula space, $\mathbf{U} = (U_1, U_2, \ldots, U_d)$, is computed as:

$$U_i = F_{X_{i,j}}(X_{i,j}), \quad i = 1, 2, \ldots, d. \tag{3.26}$$

In this space, the dependency structure among the variables is captured using a copula function C, as described by Sklar's theorem:

$$C(u_1, u_2, \ldots, u_d) = F_X\left(F_{X_1}^{-1}(u_1), F_{X_2}^{-1}(u_2), \ldots, F_{X_d}^{-1}(u_d)\right). \tag{3.27}$$

CopulaGAN models this dependency structure explicitly, often using parametric copulas such as the gaussian copula, which is defined by its correlation matrix $\mathbf{R}$. The copula-based approach enables CopulaGAN to preserve complex interdependencies that would be challenging to capture using only the adversarial framework of a standard GAN.

The generator and discriminator in CopulaGAN operate in this copula-transformed space. The generator learns to produce data points $\mathbf{U} = (U_1, U_2, \ldots, U_d)$ that follow the dependency structure encoded in the copula, while the discriminator distinguishes between real and synthetic data points. Once synthetic data is generated in copula space, it is transformed back to the original data space by applying the inverse marginal CDFs:

$$X_{i,j} = F_{X_{i,j}}^{-1}(U_i), \quad i = 1, 2, \ldots, d. \tag{3.28}$$

This mapping ensures that the synthetic data not only preserves the dependency structure but also aligns with the original marginal distributions of each feature.

A key advantage of CopulaGAN over CTGAN is its ability to explicitly model tail dependencies and asymmetric relationships, which are common in financial and risk-related

datasets. By utilizing copulas, CopulaGAN can better capture the joint behavior of variables under extreme conditions, an area where CTGAN's reliance on mode-specific normalization and conditional sampling may fall short. This makes CopulaGAN particularly suitable for applications that require accurate modeling of rare but critical events, such as stress testing and risk analysis [51].

Despite its advantages, CopulaGAN shares some of the typical challenges of GAN-based methods, including sensitivity to hyperparameter tuning and potential instability during training. Additionally, the need to estimate and work with copula parameters, such as the correlation matrix $\mathbf{R}$, introduces computational overhead compared to more straightforward GAN architectures. Nevertheless, its ability to combine the statistical rigor of copulas with the flexibility of GANs makes it a powerful tool for generating high-quality synthetic tabular data that preserves both the marginal properties and the dependency structure of the original data.

3.4.4 Tabular Variational Autoencoder approach to data generation

The Tabular Variational Autoencoder (TVAE) is a generative model specifically designed to handle the complexities of tabular data by leveraging the principles of variational autoencoders (VAEs). Unlike traditional autoencoders, which aim to learn a deterministic mapping between input data and a low-dimensional latent representation, VAEs introduce a probabilistic framework. This enables the generation of new synthetic data points by sampling from a latent space that captures the underlying distribution of the original dataset. TVAE extends this framework to effectively model both continuous and categorical variables, addressing the challenges posed by mixed data types and preserving the dependencies among features [12].

The architecture of TVAE consists of two main components: an encoder and a decoder. The encoder maps the input data $X = (X_1, X_2, \ldots, X_d)$ into a latent space by producing a mean vector $\boldsymbol{\mu}$ and a standard deviation vector $\boldsymbol{\sigma}$. The latent representation Z is then sampled from a multivariate normal distribution parameterized by these vectors:

$$Z \sim \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2)), \tag{3.29}$$

where $\text{diag}(\boldsymbol{\sigma}^2)$ is the diagonal covariance matrix. The decoder reconstructs the data from the latent representation by mapping Z back to the original data space, producing synthetic data points that resemble the original data distribution.

The training objective of TVAE is to optimize the evidence lower bound (ELBO), which balances two terms: the reconstruction loss and the Kullback-Leibler (KL) divergence. The reconstruction loss ensures that the synthetic data generated by the decoder closely matches the original data, while the KL divergence regularizes the latent space to follow a standard normal distribution. The ELBO is defined as:

$$\mathcal{L}_{\text{ELBO}} = \mathbb{E}_{q(Z|X)}[\log p(X|Z)] - D_{\text{KL}}(q(Z|X) \| p(Z)), \tag{3.30}$$

where $q(Z|X)$ is the approximate posterior distribution produced by the encoder, $p(Z)$ is the prior over the latent variables (typically standard normal), and $p(X|Z)$ represents the likelihood of the data given the latent representation. The first term measures how well the model reconstructs the data, while the second term enforces the latent space to align with the prior distribution.

A distinguishing feature of TVAE is its ability to handle categorical data effectively. Instead of directly reconstructing the original categorical variables, TVAE uses a logits-based approach to model categorical features in the decoder. Let $\mathbf{y}_i$ be a one-hot encoded representation of a categorical variable with k possible categories. The decoder predicts logits $\mathbf{l}_i \in \mathbb{R}^k$, which are then converted to probabilities using the softmax function:

$$P(y_i = j|Z) = \frac{\exp(l_{ij})}{\sum_{k=1}^{K} \exp(l_{ik})}, \tag{3.31}$$

where l_{ij} represents the j-th logit for the i-th variable. This probabilistic treatment ensures smooth gradients during training and helps maintain the relationships between categorical and continuous features.

For continuous variables, TVAE models their reconstruction as a Gaussian distribution with mean and variance predicted by the decoder. This approach allows TVAE to capture the statistical properties of continuous features, such as their variance and multimodality, without requiring explicit mode separation as in CTGAN.

Once trained, TVAE can generate synthetic tabular data by sampling from the latent space $Z \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, where $\mathbf{I}$ is the identity matrix. The decoder transforms these latent data points into synthetic data that mimics the original data's statistical properties and feature dependencies. This capability makes TVAE particularly useful for applications such as data augmentation, privacy-preserving data sharing, and the generation of synthetic datasets for machine learning model evaluation. A major advantage of TVAE over other generative models like GANs is its stability during training. Since VAEs are not adversarially trained, they avoid issues such as mode collapse, which can occur in GAN-based models. Moreover, the probabilistic nature of VAEs allows TVAE to produce a more continuous and smooth latent space, facilitating the generation of diverse and realistic synthetic data. However, TVAE may face challenges in capturing highly complex dependencies, especially in datasets with intricate non-linear relationships, where models like CTGAN or CopulaGAN might perform better [12]. In summary, TVAE offers a robust and flexible framework for generating synthetic tabular data by leveraging the strengths of variational autoencoders. Its ability to handle mixed data types, combined with a stable and interpretable training process, makes it a valuable tool for a wide range of data-centric applications.

3.4.5 Methods of evaluating generated synthetic data

The quality of synthetic data is evaluated using metrics designed to measure its fidelity and utility. Two important metrics are the Column Pair Score and the Column Shape Score, which assess the ability of the synthetic data to preserve the statistical relationships and distributional properties of the original data. The Column Pair Score evaluates the pairwise dependencies between columns by comparing the joint probability distributions of all column pairs in the real-world and synthetic datasets. Let

$P_{\text{real}}(X_{i,j}, X_{:,j})$ and $P_{\text{synthetic}}(X_{i,j}, X_{:,j})$ denote the joint distributions of columns $X_{:,j}$ and $X'_{:,j}$ in the real and synthetic datasets, respectively. The Column Pair Score, denoted as C_{pair}, is calculated as the average Jensen-Shannon (JS) divergence across all column pairs [59]:

$$C_{\text{pair}} = 1 - \frac{1}{N_{\text{pairs}}} \sum_{(i,j)} D_{\text{JS}}(P_{\text{real}}(X_{:,j}, X'_{:,j}) \| P_{\text{synthetic}}(X_{i,j}, X'_{:,j})), \tag{3.32}$$

where N_{pairs} is the total number of column pairs, and D_{JS} represents the JS divergence, which measures the similarity between two probability distributions. A score close to 1 indicates that the pairwise dependencies are well preserved. The Column Shape Score, denoted as C_{shape}, evaluates how well the marginal distributions of individual columns in the synthetic data match those of the real data. Let $P_{\text{real}}(X_{:,j})$ and $P_{\text{synthetic}}(X_{:,j})$ represent the marginal distributions of column $X_{:,j}$ in the real and synthetic datasets. The Column Shape Score is computed as [59]:

$$C_{\text{shape}} = 1 - \frac{1}{N_{\text{columns}}} \sum_{i} D_{\text{JS}}(P_{\text{real}}(X_{:,j}) \| P_{\text{synthetic}}(X_{:,j})), \tag{3.33}$$

where N_{columns} is the total number of columns. This metric quantifies how closely the synthetic data replicates the distributional characteristics of the real data for each feature. Both metrics are crucial for ensuring that synthetic data accurately reflects the underlying patterns and dependencies present in the original dataset.

3.5 Summary of created datasets

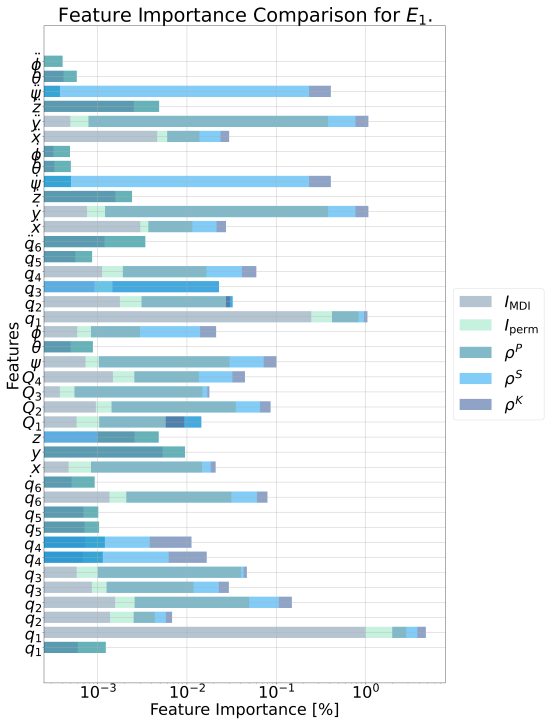
The goal of the presented chapter has been to realize the scientific contribution of defining the methodology for the collection of a dataset suitable for the development of an energy-prediction model for six DOF IRMs. This was realized through the description of three separate approaches – the collection of real-world data from a laboratory environment, collection from the simulated environment using the digital twin approach, and data synthetization approach, with four different methods. The methodology for the comparison of datasets was also given.

CHAPTER 4

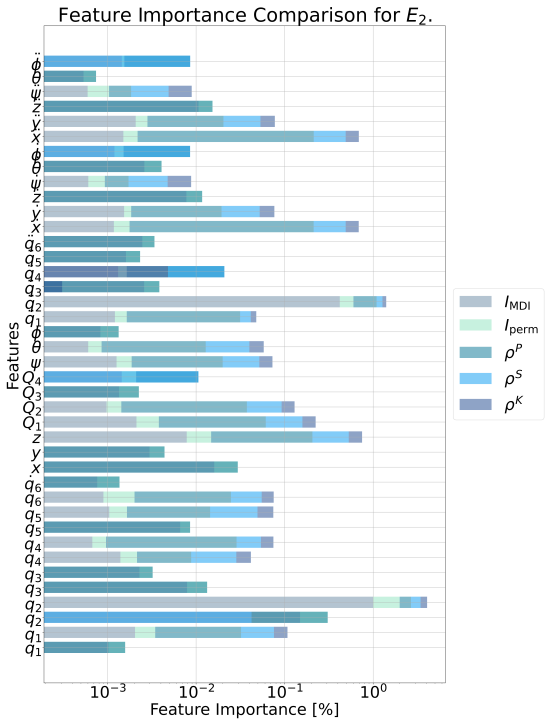
Results and discussion

4.1 Data analysis results

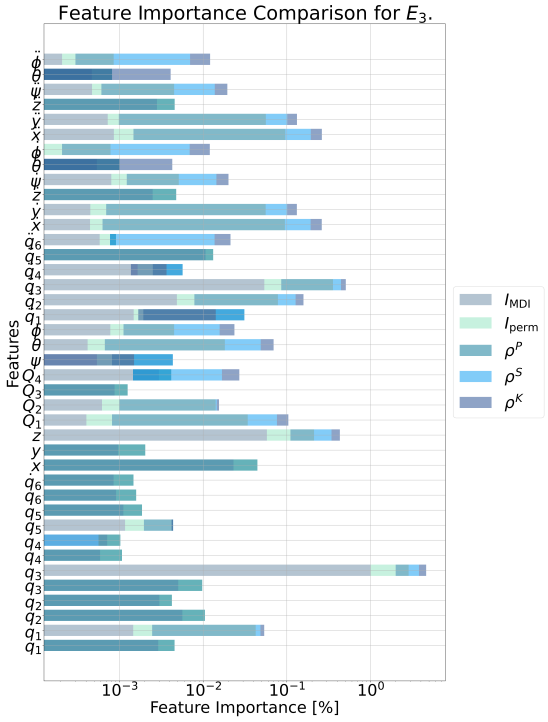
This section is dedicated to presenting the results of the data analysis and providing commentary on the findings. The primary objective here is to assess the effectiveness of synthetic method generation by comparing both types of synthetic datasets, with the original dataset obtained from experiments on the real robot. This evaluation is conducted using a variety of similarity metrics, including the column shape score and the column pair score, alongside an examination of the overall data distribution and key descriptive statistics. In addition to these metrics, feature importance serves as a critical input in the more advanced analytical methods applied in this study. While the aforementioned distributions and statistics are utilized as inputs during the data synthetization process, feature importance metrics play a different role. as they help identify which variables exert the most significant influence on a specific output. These outputs include the energy consumption of individual joints, as illustrated in Subfigures 4.1a through 4.1b, and the total energy, depicted in Subfigure 4.2c. The visual representations shown in these subfigures are derived by applying the feature importance metrics introduced in the methodology chapter. These metrics are computed relative to the targeted output, and their values are stacked to determine which features contribute most prominently. The insights obtained from this process are subsequently used within the MA (presumably a Model Architecture or Methodological Approach) to guide the optimization process, focusing on the features deemed most influential. Notably, this feature importance analysis is conducted on the real-world dataset, as it represents the principal subject of the optimization efforts.



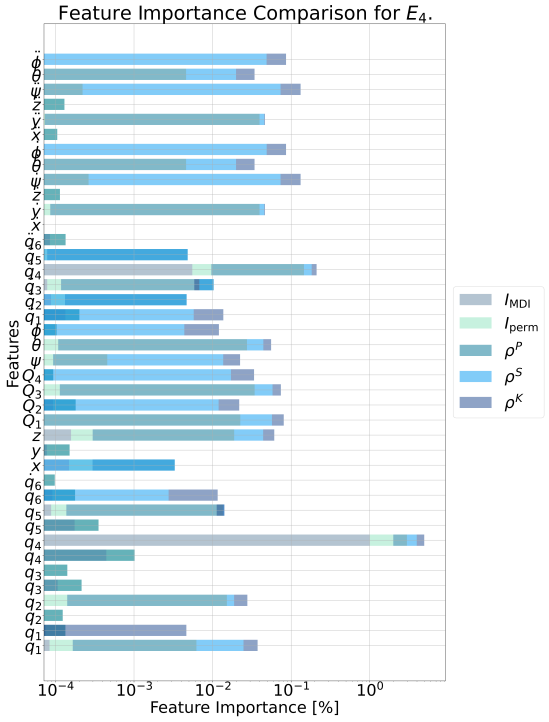
(a) The influence of individual features on the energy of the first joint.



(b) The influence of individual features on the energy of the second joint.

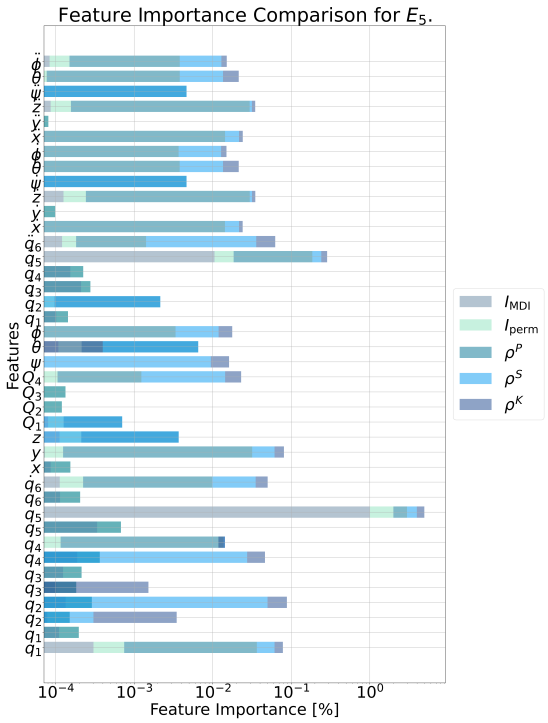


(c) The influence of individual features on the energy of the third joint.

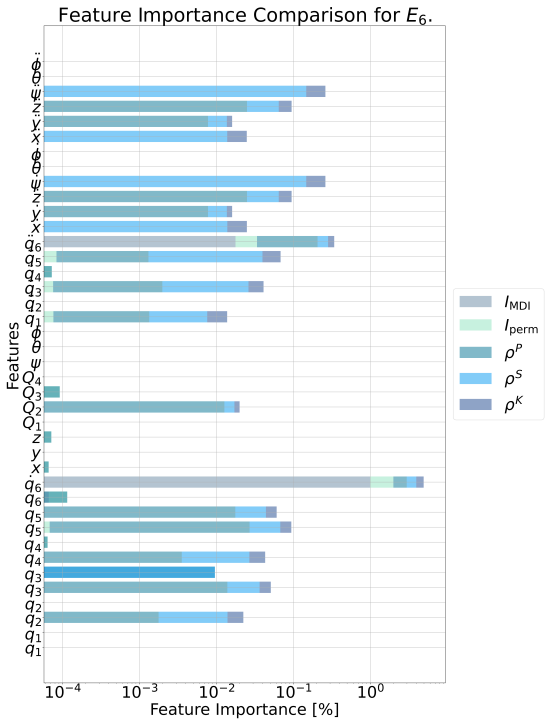


(d) The influence of individual features on the energy of the fourth joint.

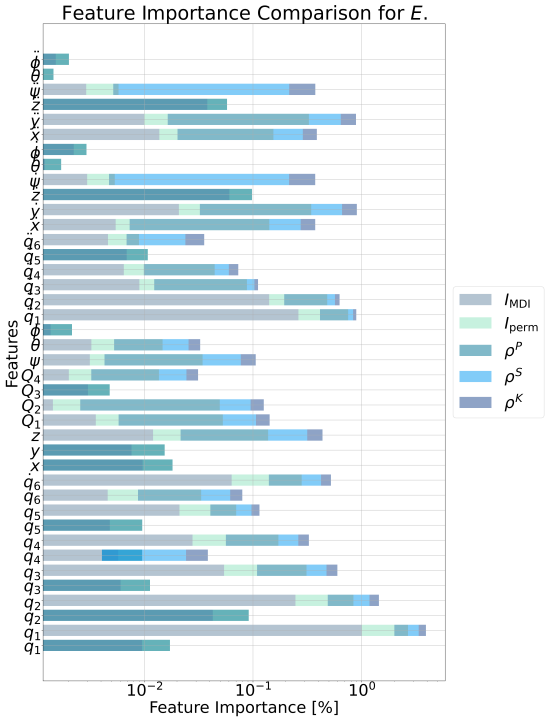
Figure 4.1: The influence of individual features on the energy of each joint.



(a) The influence of individual features on the energy of the fifth joint.



(b) The influence of individual features on the energy of the sixth joint.



(c) The influence of individual features on the energy of the entire manipulator.

Figure 4.2: The influence of individual features on the energy of each joint (cont.).

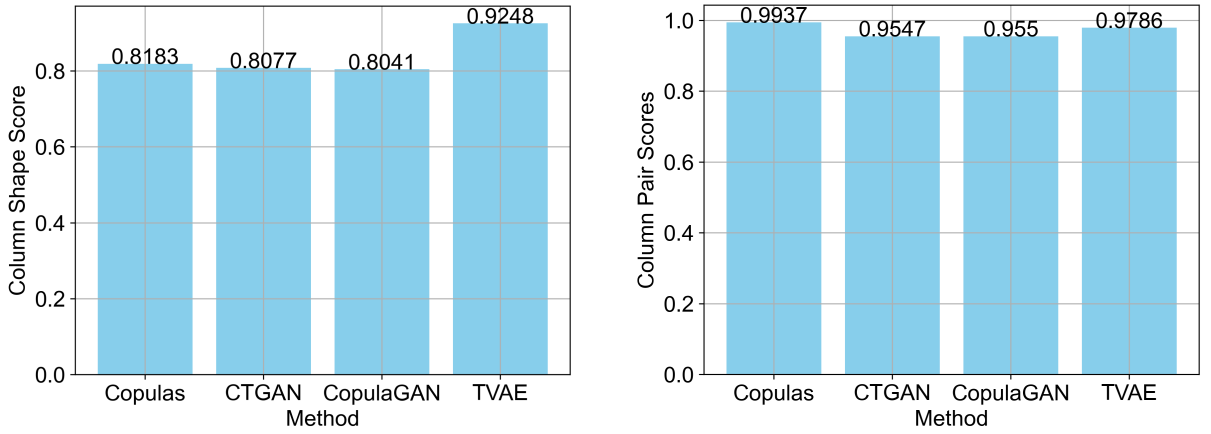
After observing the figures there are a few key points to be taken away from. First, the position and the speed of an individual joint generally shows the highest influence on that joints energy use, which is to be expected, indicating that the data follows common sense. In the similar manner, it can be observed that the variables indicating the position in the tool space (x, y, and z) have a higher influence on the output of the first three joints, while the orientation related variables (Euler's angles and quarternions) have a higher influence on the last three joints. In addition to it, it can be observed that overall, the speed and position of the first two joints continues to have influence on the joints. This is most apparent in the case of the total energy, where the speeds of the first two joints have the highest influences. This indicates that the MA algorithm, targeted optimization should focus on lowering the elements pertaining to these two joints, especially their speeds – in other words, the algorithm will focus on optimizing the parameters a_1^1, a_2^1, a_3^1, a_4^1, a_5^1, a_1^2, a_2^2, a_3^2, a_4^2, and a_5^2. This is done due to the assumption that these parameters will have a higher influence on the speed and position of the first two joints, directly lowering the two values.

4.2 Data synthetization results and comparison

4.2.1 Metrics comparison

The simplest way to indicate the performance of the synthetization algorithm is to evaluate the related metrics – column shape and column pair scores, which were defined in the previous chapter. The column shape scores are calculated comparing the real-world dataset collected from the IRM with the datasets generated by each of the four algorithms utilized for the data synthetization, and shown in Figure 4.3a. Observing the figure shows that the Copulas achieved a score of 0.82, CTGAN method achieved the score of 0.80, and TVAE achieved a noticeably higher score of 0.93. This indicates that the variables generated with TVAE had the highest similarities regarding the column shape. Evaluation based on a single metrics should always be avoided, so the column pair scores are also calculated and presented in Figure 4.3b. The results show

that Copulas achieved the highest score, with a value of 0.99, followed by TVAE with 0.98. CopulaGAN and CTGAN attained similar scores of 0.96 and 0.95, respectively. The scores are much more uniform in the presented example, with the Copulas showing the highest performance here. Still, it should be noted that Copulas showed a much poorer score in the previous analysis, meaning that currently TVAE, has the best overall performance. While the TVAE-based dataset seems to show the best performance, prior to the final selection of the synthetic dataset for further analysis, the distributions across methods should be compared.



(a) Column shape scores of datasets generated using different data synthetization methods.

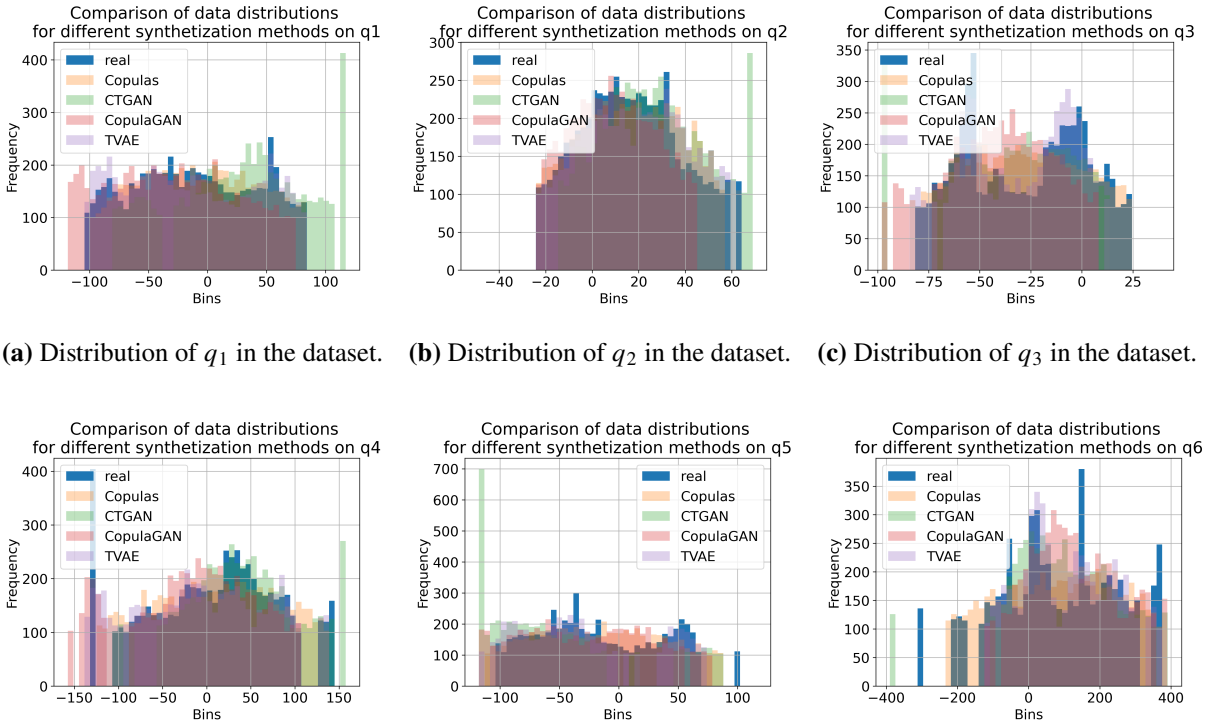
(b) Column pair scores of datasets generated using different data synthetization methods.

Figure 4.3: Numerical evaluation of synthetization methods.

4.2.2 Distribution comparisons

Distribution comparisons can be started with the joint positions, as given in Figure 4.4. In this, and following figures, the real dataset distribution is given using black color, with overlays given in different colors for different methods – blue for Copulas, red for CTGAN, green for CopulaGAN and purple for TVAE. The first noticeable change compared to the real data is the tendency of CTGAN to group large amounts of variables near the lower (for example, present in 4.4d) or higher (present in 4.4a) end of the range. The CopulaGAN shows poorer performance again, tending towards a uniform

distribution overall in some cases, showing a poor tracking of the scores. Copulas, while not having these issues, shows the tendency towards the normal distribution, while TVAE is the only dataset that follows the distribution closely in cases of more complex distributions such as 4.4c.



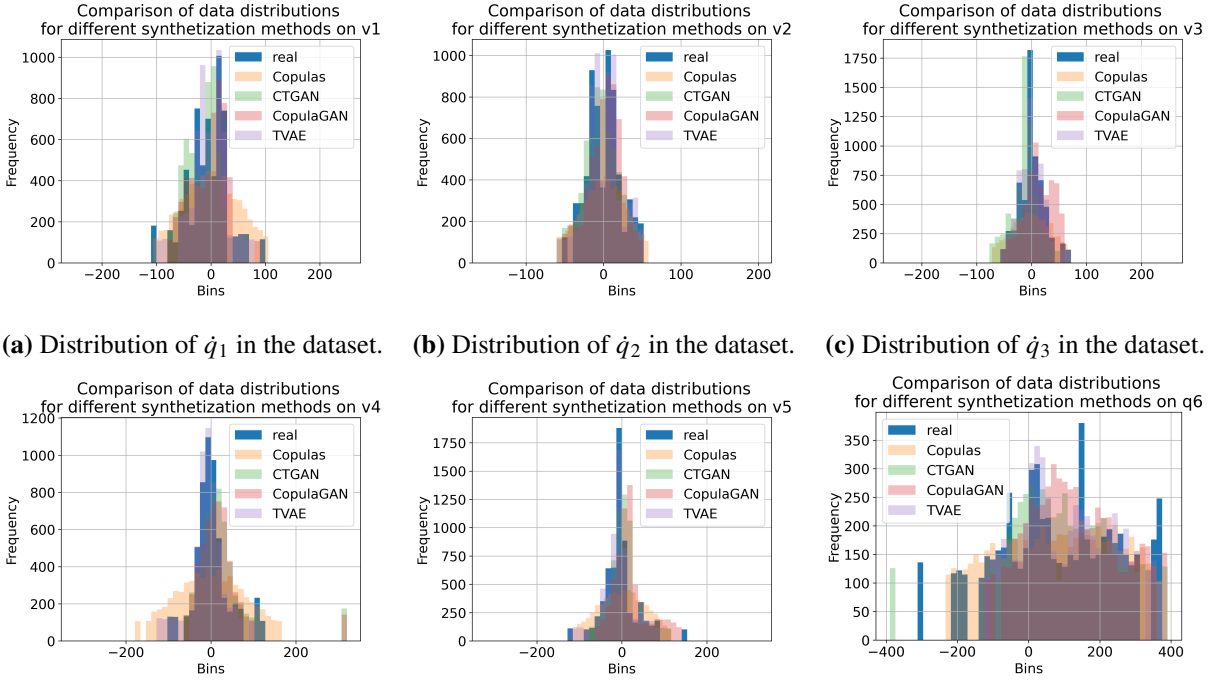
(a) Distribution of q_1 in the dataset. **(b)** Distribution of q_2 in the dataset. **(c)** Distribution of q_3 in the dataset.

(d) Distribution of q_4 in the dataset. **(e)** Distribution of q_5 in the dataset. **(f)** Distribution of q_6 in the dataset.

Figure 4.4: The distribution of measured joint angle positions in the dataset.

The datasets corresponding to the speed measurements, as depicted in Figure 4.5, exhibit a comparable overall behavior across the different generative methods. Among these, TVAE demonstrates a notable ability to accurately model certain distributions that are more complex in nature. This capability is particularly evident in subfigure 4.5b, where TVAE is the sole method that successfully captures the characteristic decline in speed values near zero, a feature that is clearly present in the real dataset. In contrast, the performance of the other methods is generally less effective in this regard. A particularly significant example is the Copulas approach, which tends to approximate the normal distribution, resulting in a less accurate representation of the real data. This

tendency is distinctly observable in Figure 4.5d, where the limitations of the Copulas method become apparent.



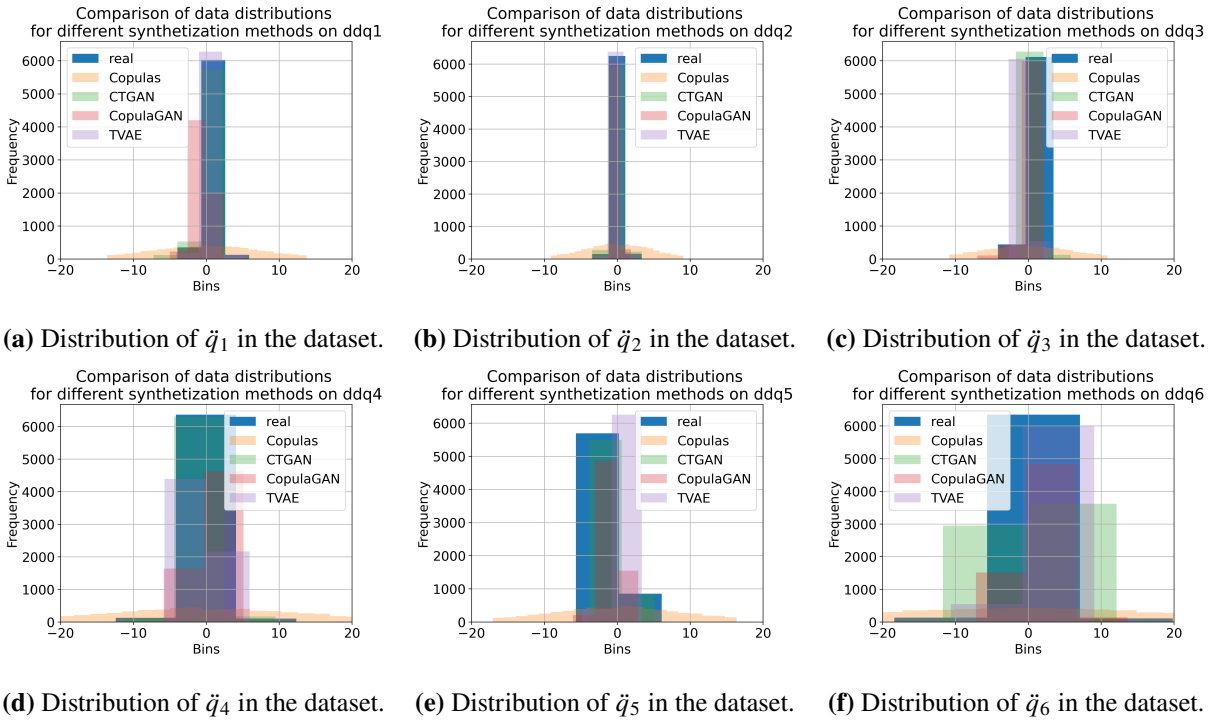
(a) Distribution of $\dot{q}_1$ in the dataset. **(b)** Distribution of $\dot{q}_2$ in the dataset. **(c)** Distribution of $\dot{q}_3$ in the dataset.

(d) Distribution of $\dot{q}_4$ in the dataset. **(e)** Distribution of $\dot{q}_5$ in the dataset. **(f)** Distribution of $\dot{q}_6$ in the dataset.

Figure 4.5: The distribution of measured joint angle speeds in the dataset

Regarding the acceleration data, it can be observed that the values in the dataset are generally quite low and are predominantly concentrated around zero. Only a relatively small portion of the data extends toward the higher or lower extremes of the variable's possible range. This pattern is entirely expected, given the nature of the dataset, which primarily records the robot's movement during operation. Significant changes in acceleration, whether increases or decreases, are typically limited to moments when a new trajectory is initiated, requiring sudden adjustments in both direction and speed. Due to the simplicity of this distribution, it is generally not challenging to replicate it synthetically. However, despite this, the Copulas method once again performs particularly poorly. It consistently exhibits a strong tendency to approximate a standard normal distribution centered at zero, which fails to accurately reflect the true distribution of the data. In comparison, the remaining methods manage to achieve a noticeably better fit

to the observed data distribution.

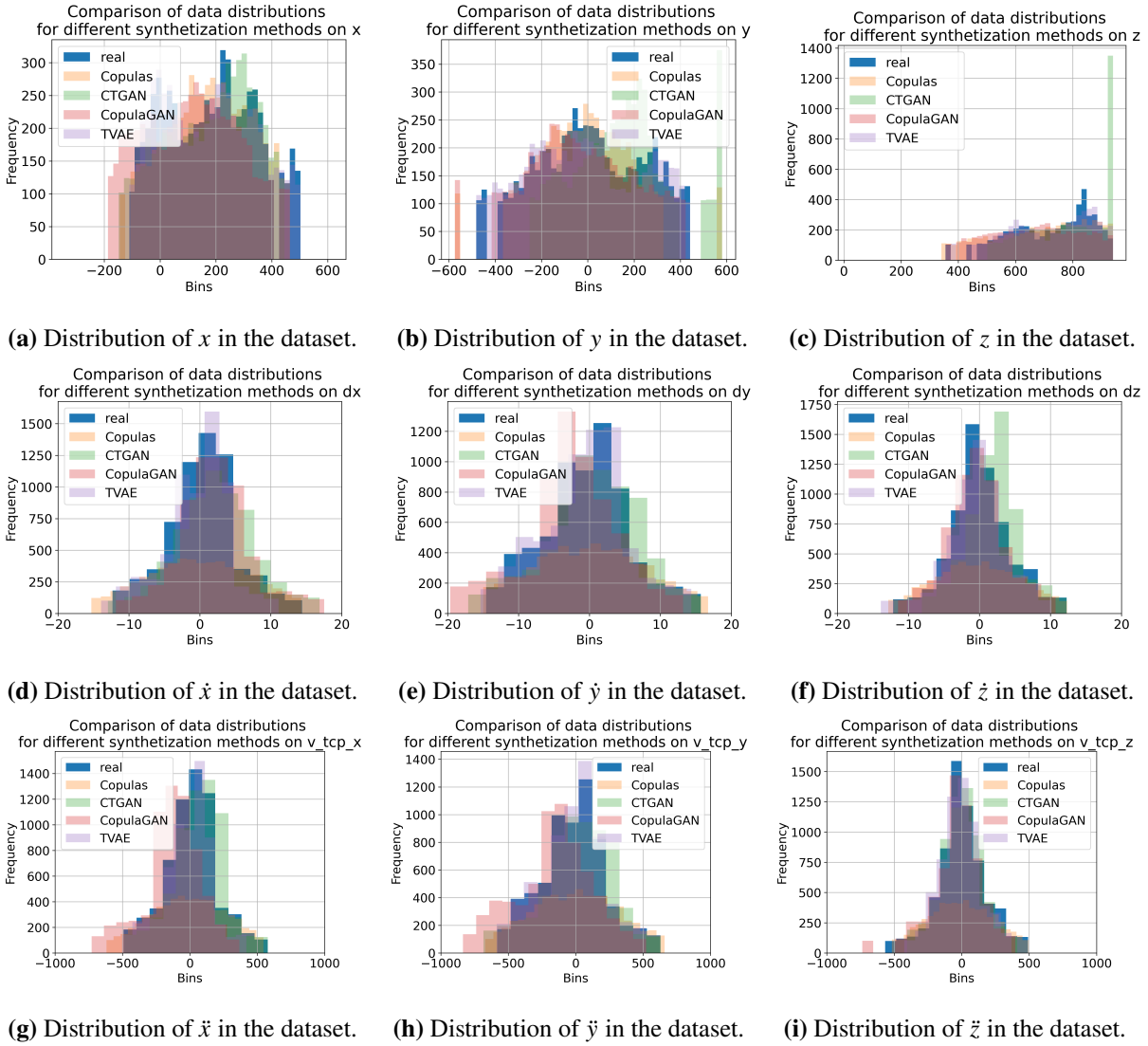


(a) Distribution of $\ddot{q}_1$ in the dataset. **(b)** Distribution of $\ddot{q}_2$ in the dataset. **(c)** Distribution of $\ddot{q}_3$ in the dataset.

(d) Distribution of $\ddot{q}_4$ in the dataset. **(e)** Distribution of $\ddot{q}_5$ in the dataset. **(f)** Distribution of $\ddot{q}_6$ in the dataset.

Figure 4.6: The distribution of measured joint angle accelerations in the dataset.

When analyzing the distributions of the positions of the robot's end-effector within the tool space, as illustrated in Figure 4.7, several important observations can be made. One notable point is the behavior of CTGAN, which consistently demonstrates a tendency to generate a disproportionately large number of values located near the outer extremes of the variable ranges. This characteristic is indicative of its poor performance in accurately modeling the real data distributions. This issue is especially apparent in subfigures 4.7b and 4.7c, where CTGAN's generated data diverges significantly from the patterns observed in the actual dataset. Although CTGAN performs somewhat better when modeling distributions that are more centrally concentrated, such as the one shown in Figure 4.7d, where it manages a closer approximation to the real data, its persistent tendency to favor extreme values undermines its overall effectiveness. Consequently, despite some limited success in specific cases, this flaw renders CTGAN unsuitable for continued use in this context.



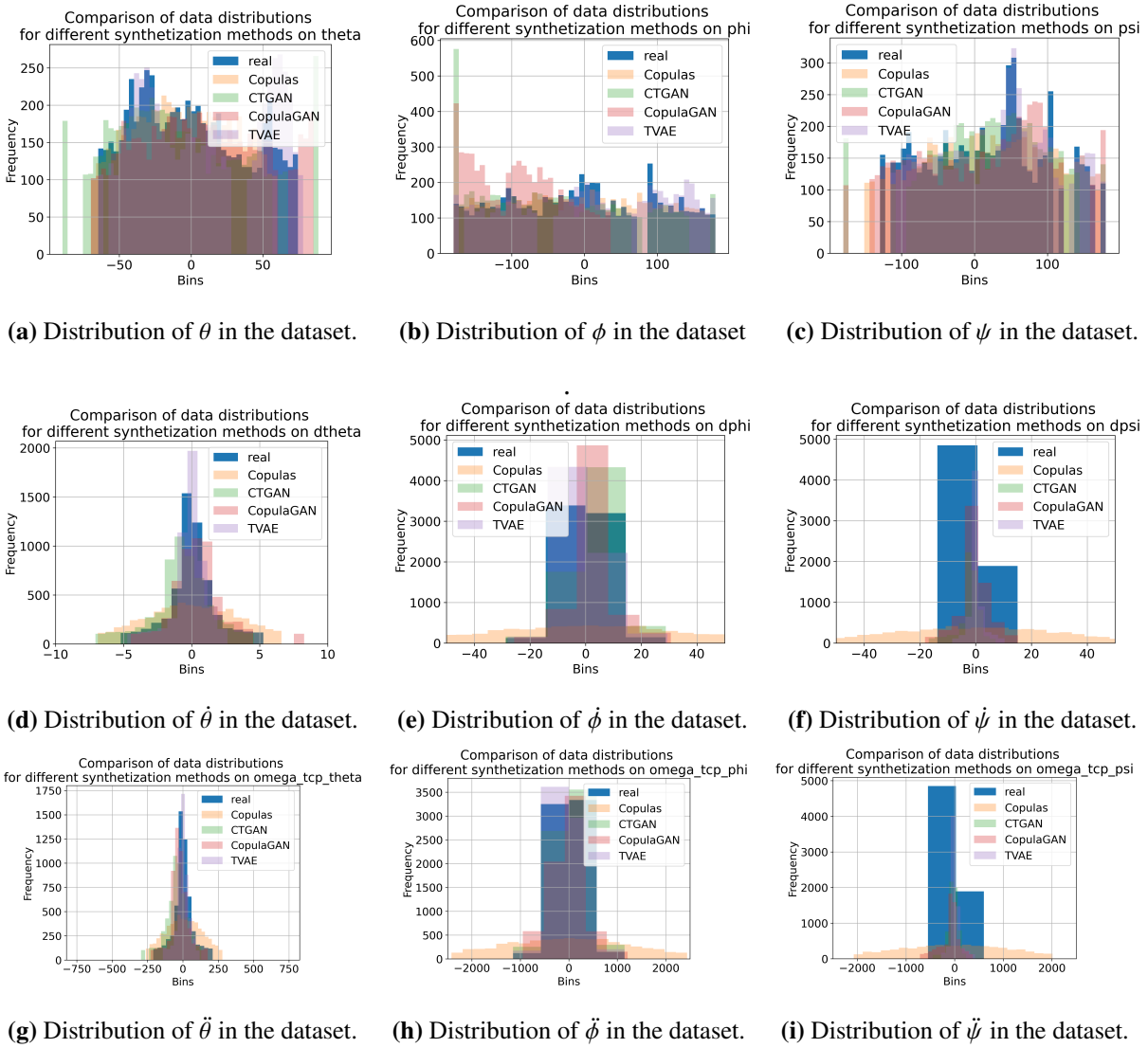
(a) Distribution of x in the dataset. **(b)** Distribution of y in the dataset. **(c)** Distribution of z in the dataset.

(d) Distribution of $\dot{x}$ in the dataset. **(e)** Distribution of $\dot{y}$ in the dataset. **(f)** Distribution of $\dot{z}$ in the dataset.

(g) Distribution of $\ddot{x}$ in the dataset. **(h)** Distribution of $\ddot{y}$ in the dataset. **(i)** Distribution of $\ddot{z}$ in the dataset.

Figure 4.7: The distribution of TCP positions, speeds and accelerations in the dataset.

In case of orientations and speeds, in Figure 4.8, the similar is shown, especially in the case of 4.8a and 4.8b, where CopulaGAN and CTGAN tend to generate large amounts of values at the extremes of the range. Copulas method shows the tendency of generating normally distributed variables (4.8e, 4.8h) when faced with data that has centrally focussed distribution. TVAE overall shows decent performance, with minimal errors, such as the tendency to generate somewhat more data centrally in distributions that are centered around 0.



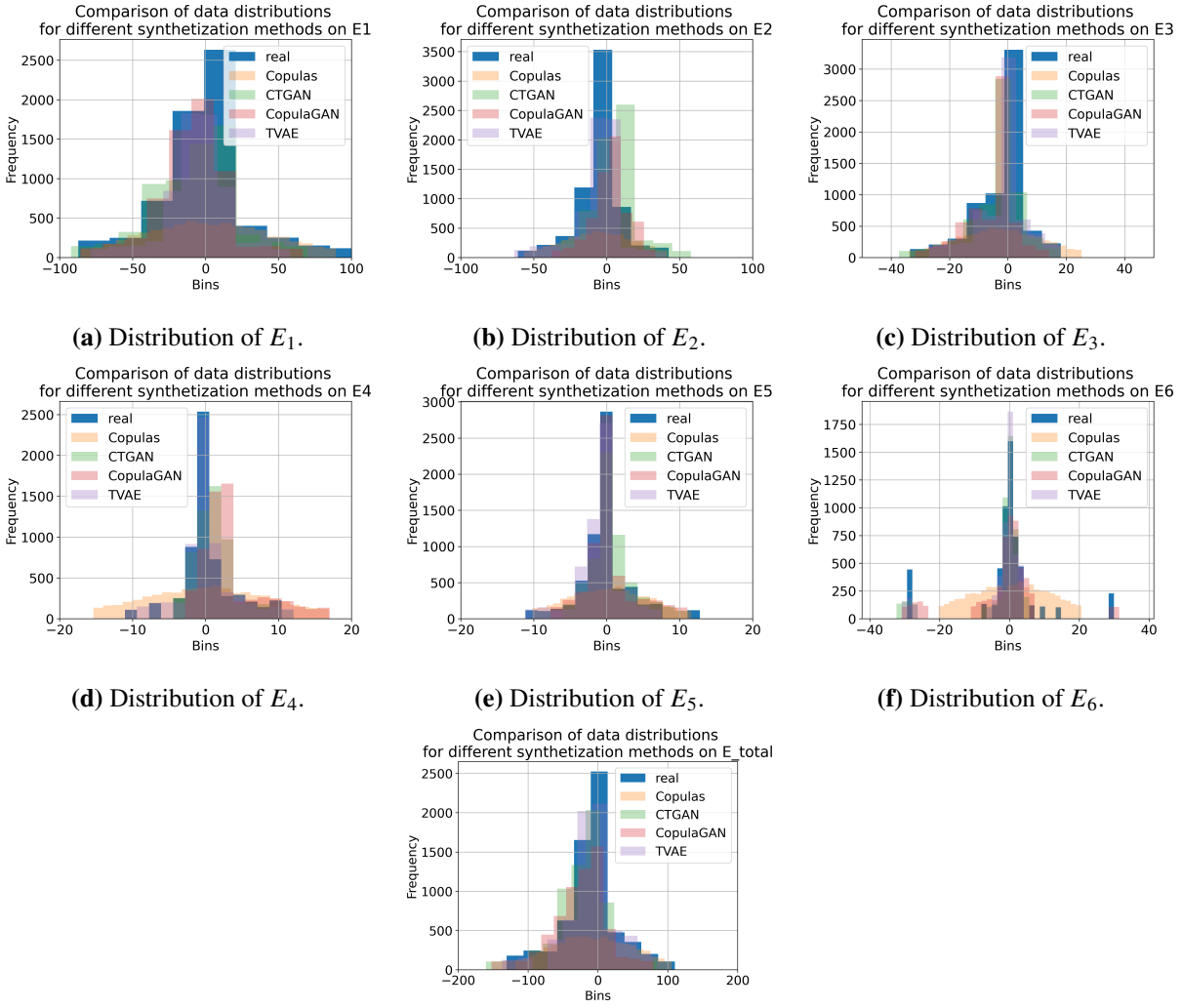
(a) Distribution of θ in the dataset. **(b)** Distribution of ϕ in the dataset **(c)** Distribution of ψ in the dataset.

(d) Distribution of $\dot{\theta}$ in the dataset. **(e)** Distribution of $\dot{\phi}$ in the dataset. **(f)** Distribution of $\dot{\psi}$ in the dataset.

(g) Distribution of $\ddot{\theta}$ in the dataset. **(h)** Distribution of $\ddot{\phi}$ in the dataset. **(i)** Distribution of $\ddot{\psi}$ in the dataset.

Figure 4.8: The distribution of TCP orientations, speeds and accelerations in the dataset.

Finally, the energy analysis (Figure 4.9), shows similar information as previously noted. Most interesting is the distribution of the sixth joint (4.9f), which is shown to be multivariate, with additional peaks of data present near the extremes of the range. All methods, except Copulas, show some values centered around these peaks, but fail to synthetize enough data in a small enough area to truly recreate them, showing the limitation of the synthetization methods.



(a) Distribution of E_1.

(b) Distribution of E_2.

(c) Distribution of E_3.

(d) Distribution of E_4.

(e) Distribution of E_5.

(f) Distribution of E_6.

(g) Distribution of total used energy in the dataset.

Figure 4.9: The distribution of measured joint angle accelerations in the dataset.

Based on the above analysis it can be concluded that TVAE is the best performing method overall. CTGAN and CopulaGAN not only showed poor scores when tested with evaluation metrics, but have also shown tendencies towards the generation of non-existing data near the extremes of the real data range, which are not present in the real data. Copulas, while scoring well when evaluated with metrics, fails to model data with high central tendency in the distribution, defaulting to the modeling of normally distributed data.

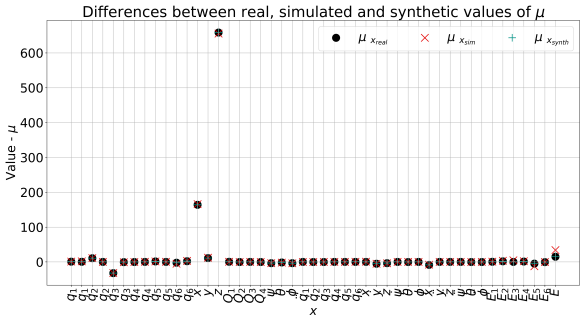
The next step is to compare the TVAE selected data with the real data (collected in a laboratory environment for the ABB IRB 120 IRM), and simulated data (collected within a RobotStudio simulated environment).

4.2.3 Descriptive statistics comparison

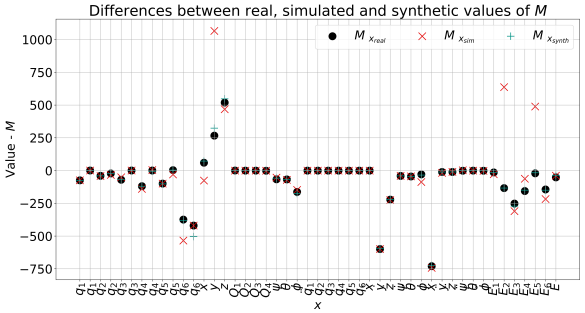
Figure 4.10 shows the comparison of values for individual descriptive statistics, as described in the previous chapter. Each of the graphs show a different statistic, with values for individual variables in the dataset. The value of the statistic for the given variable within the real dataset is given with a black dot, while the values from the simulated dataset are indicated with the red cross, and the values from the synthetic dataset are indicated with a cyan 'x'. The main point of the analysis is to observe if there are large differences between certain values, as large differences in statistics may indicate that one of the generated datasets does not represent the original set well.

Subfigure 4.10a shows the difference in the mean values of the dataset, indicating the central tendency of the datasets. Observing the values for all three datasets for different variables, a significant overlap between most values can be seen. For example, the position of the second joint (q_2) shows an almost perfect overlap across all three datasets. Still, some of the other values show a difference. This is especially visible for outputs E_1 to E_6. Most of the values that show a change show that the simulated dataset differs more in comparison to the synthetic dataset. In some cases, such as E_1, or the total energy E, the values show overlap on the graph when comparing synthetic data and real data, while simulated data shows a different value. The values for mode in Subfigure 4.10b, show similar trends, with the central tendency of the synthetic and real dataset variables being significantly closer than the ones in the real dataset. This difference is much less visible when observing the standard deviation of the variables (4.10d) and the range (4.10c). These four points together indicate that while the ranges follow a similar distribution in the real and simulated dataset, there are certain shifts between the real data (and synthetic data based on it), indicating that the simulation data

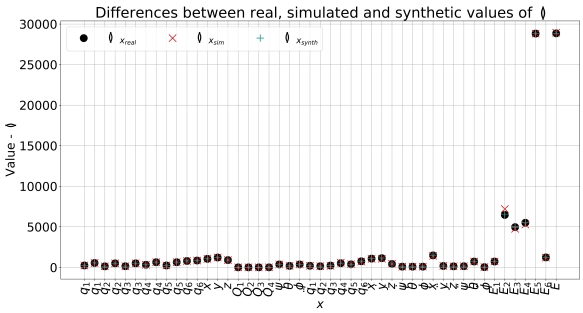
is not necessarily the best representative of the real values.



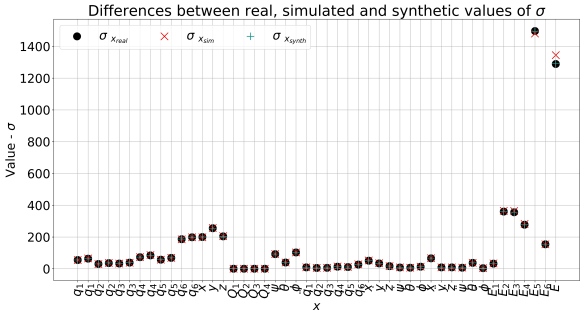
(a) Comparison of mean values across datasets.



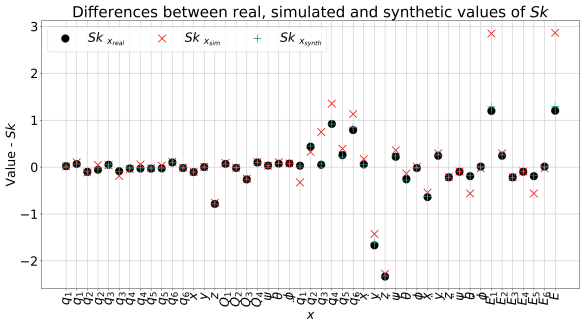
(b) Comparison of mode values across datasets.



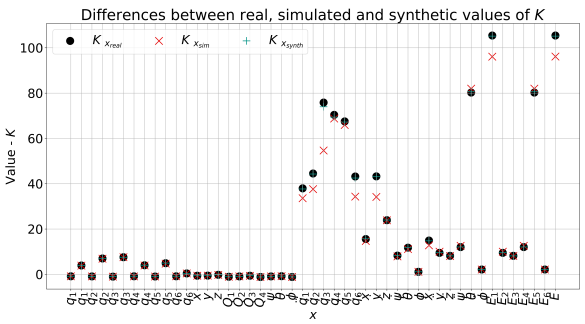
(c) Comparison of range values across datasets.



(d) Comparison of σ for variables across datasets.



(e) Comparison of skewness across datasets.



(f) Comparison of kurtosis across datasets.

Figure 4.10: The difference between the descriptive statistics of the real-world and synthetic datasets in comparison to the real-world dataset.

This is confirmed by the kurtosis (4.10f), where all the values are similar. meaning that most of the data has a similar wideness around the central point, even if the central point may not necessarily overlap, suggesting a possible shift of data between the real and synthetic data. This is further confirmed by the skewness as shown in subfigure 4.10e, showing, once more, similarity between synthetic and real-world data, with a

shift visible in most variables. Overall, the similarities between datasets are visible, but not so significant that it would be possible to automatically presume that modeling the real data would not be possible with either dataset, although they indicate that simulation data may have poorer performance.

4.3 Performance of ML models on validation and test data

4.3.1 Hyperparameter selection of ML models

The process of hyperparameter selection represents a fundamental component in the analysis phase of any machine learning (ML) study. It provides critical insights into the structural characteristics and configurations of the models that demonstrate the best performance during the training phase. This information not only guides the interpretation of the results but is also essential for ensuring the repeatability and reproducibility of the study, which are key aspects of any rigorous scientific investigation.

The real-world data and the simulation data used in this study were both generated by selecting 1,000 random points from within the operating space of the IRM. At each of these points, the measurement of relevant variables, as previously described in the preceding chapter, was carried out at regular time intervals of 0.025 seconds. Due to variations in the lengths of the paths followed during the experiments, the total number of data points collected differed slightly between the two datasets. Specifically, the dataset obtained from the real-world operation of the IRM contains a total of 75,792 data points, whereas the dataset gathered from the simulation environment includes 75,141 data points. In contrast to these datasets, the synthetic dataset was not subject to the same real-world constraints regarding the number of data points. Therefore, for consistency and convenience, the number of data points in the synthetic dataset was fixed at exactly 75,000. These synthetic data points were generated from an additional dataset, which, like the real-world dataset, was collected through laboratory experiments. This auxiliary dataset was created using 100 randomly selected points within the operating space and resulted in a total of 6,998 data points. The separate test

set was similarly constructed using 100 randomly selected points within the operating space, which produced 7,297 data points in total.

For the purpose of cross-validation during model training, a five-fold cross-validation procedure was employed across all datasets. In the case of the real-world dataset, this involved dividing the data into two folds, each containing 15,159 data points, and three folds, each comprising 15,158 data points. The simulation dataset was partitioned into one fold with 15,029 data points and four folds, each with 15,028 data points. The synthetic dataset, having an exact total of 75,000 data points, was evenly split into five folds, each consisting of 15,000 data points. As a result, in each fold of the cross-validation process, approximately 60,000 data points were used for training. More precisely, this figure corresponds to the combined total of the four training folds and amounts to 60,632 or 60,630 data points in the real-world dataset, and 60,111 or 60,112 data points in the simulation dataset, depending on the specific fold excluded for validation. The validation set size in each fold corresponded exactly to the size of the fold designated for validation in that particular iteration. The test set, used consistently across all experiments, remained the same in every case, comprising 7,297 data points collected directly from the real-world IRM system, as previously mentioned.

The results presented in Table 4.1 indicate consistently high performance across most models, with the coefficient of determination ($\overline{R^2}$) values reaching 0.999 and exhibiting minimal variability ($\sigma_{R^2} = 0.001$) for nearly all outputs and data types (real-world, simulated, and synthetic). This exceptional consistency underscores the models' ability to generalize well across cross-validation folds, reflecting the stability and reliability of the training process. Similarly, the mean absolute error ($\overline{\text{MAE}}$) values are remarkably low for most configurations, with the lowest value observed for $E5$ in the simulated dataset ($\overline{\text{MAE}} = 0.001$, $\sigma_{\text{MAE}} = 0.001$). These results highlight the precision of the models in approximating the target outputs. One of the most significant trends is the uniform use of the 'identity' activation function (ϕ) across all models, suggesting its suitability for this specific application and the nature of the data. The initial learning rate (α - init.) also follows a consistent pattern, predominantly set to $1 \cdot 10^{-4}$ for models with simpler

architectures (e.g., hidden layer size = 10) and occasionally increased to 0.1 for larger configurations (e.g., $(50, 50, 50)$). This indicates a deliberate balance between learning speed and convergence stability. The learning rate type (α) is consistently constant ('const.'), and the regularization rate ($L2$) remains low (0.01), which likely contributes to the stability and robustness of the models by preventing overfitting. The hyperparameter configurations of the best-performing model, defined as the one with the lowest $\overline{\text{MAE}}$ among models with equivalent $\overline{R^2}$, are found in the synthetic dataset for $E5$. This model achieves $\overline{R^2} = 0.999$, $\sigma_{R^2} = 0.001$, $\overline{\text{MAE}} = 0.001$, and $\sigma_{\text{MAE}} = 0.001$. Its hyperparameters include the 'identity' activation function (ϕ), an initial learning rate (α - init.) of 0.0001, a single hidden layer size of 10, a constant learning rate (α), an $L2$ regularization rate of 0.01, and the 'lbfgs' solver. This combination represents an optimal balance between simplicity and effectiveness, leading to superior performance. Notably, an outlier is observed for the simulated dataset model for E, where the $\overline{\text{MAE}}$ increases dramatically to 4.348, with a standard deviation (σ_{MAE}) of 0.316. This result is associated with a distinct solver ('adam') and a reduced hidden layer configuration $((10, 10))$, combined with a learning rate initialization of 0.0001. This deviation suggests potential challenges in optimizing this specific output using the 'adam' solver or an inadequacy of the selected hyperparameters for this dataset. It may also reflect sensitivity to solver choice in cases of more complex data behavior. In conclusion, while the models demonstrate remarkable consistency and high performance across most configurations, the synthetic dataset for E highlights the importance of careful solver and hyperparameter selection. The results emphasize the effectiveness of small, consistent learning rates, low regularization, and the 'lbfgs' solver in achieving robust performance. The hyperparameter configuration of the best-performing model (synthetic E) can serve as a baseline for further optimization, offering a reliable starting point for future model development and tuning efforts.

Table 4.1: The results of best models found using grid search and cross validation procedures, for MLP (R^2 – average value of coefficient of determination across folds, σ_{R^2} – standard deviation of R^2 score across folds, $\bar{\text{MAE}}$ – mean absolute error, σ_{MAE} – standard deviation of MAE across folds, ϕ – activation function, α - init. – Initial learning rate, α – learning rate type, $L2$ – regularization rate).

Data	Out	$\overline{R^2}$	σ_{R^2}	$\overline{\text{MAE}}$	σ_{MAE}	ϕ	α - init.	Hidden layer sizes	α	$L2$	Solver
	E_1	0.999	$1 \cdot 10^{-3}$	$1.6 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	identity	0.1	(50 50 50)	const.	$1 \cdot 10^{-2}$	lbfgs
	E_2	0.999	$1 \cdot 10^{-3}$	$8 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	0.1	(50 50 50 50)	const.	1	lbfgs
	E_3	0.999	$1 \cdot 10^{-3}$	$6 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
real	E_4	0.999	$1 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_5	0.999	$1 \cdot 10^{-3}$	$2 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_6	0.999	$1 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E	0.999	$1 \cdot 10^{-3}$	$2.2 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	identity	0.1	(10 10)	const.	$1 \cdot 10^{-2}$	lbfgs
	E_1	0.999	$1 \cdot 10^{-3}$	$1.7 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	identity	0.1	(50 50 50)	const.	$1 \cdot 10^{-2}$	lbfgs
	E_2	0.999	$1 \cdot 10^{-3}$	$9 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_3	0.999	$1 \cdot 10^{-3}$	$6 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
synth	E_4	0.999	$1 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_5	0.999	$1 \cdot 10^{-3}$	$2 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_6	0.999	$1 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E	0.999	$1 \cdot 10^{-3}$	$2.1 \cdot 10^{-2}$	$2 \cdot 10^{-3}$	identity	0.1	(10 10 10)	const.	0.1	lbfgs
	E_1	0.999	$1 \cdot 10^{-3}$	$8 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	0.1	lbfgs
	E_2	0.999	$1 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_3	0.999	$1 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
sim	E_4	0.999	$1 \cdot 10^{-3}$	$2 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_5	0.999	$1 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E_6	0.999	$1 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	identity	$1 \cdot 10^{-4}$	10	const.	$1 \cdot 10^{-2}$	lbfgs
	E	0.993	$1 \cdot 10^{-3}$	4.348	0.316	identity	$1 \cdot 10^{-4}$	(10 10)	const.	$1 \cdot 10^{-2}$	adam

The results for the PAR models, shown in Table 4.2, reveal strong performance across nearly all configurations, with $\overline{R^2}$ values consistently reaching 0.999 and low variability ($\sigma_{R^2} = 0.001$) across folds. Similar to the trends observed in the MLP models from the previous tables, the PAR models maintain high accuracy, demonstrating their ability to generalize well across datasets. However, the $\overline{\text{MAE}}$ values for PAR models are generally higher than those for MLP, ranging between 0.037 and 0.049 for most outputs—still within acceptable limits for many practical applications. A key observation is the influence of hyperparameters C and ϵ. Most models perform well with smaller C

values (e.g., $C = 0.1$ or 0.5), though certain outputs, such as $E3$ and $E4$ in the synthetic dataset, achieve strong performance even with $C = 10$, indicating that higher regularization can be beneficial for specific cases. The value of ϵ is uniformly set to 0.1 across all configurations, suggesting it is a robust choice for stability and precision. The best-performing model, defined by the lowest $\overline{\text{MAE}}$, is found in the synthetic dataset for $E2$ ($\overline{\text{MAE}} = 0.037$, $\sigma_{\text{MAE}} = 0.004$). This model uses $C = 0.1$, $\epsilon = 0.1$, $\epsilon_{\text{insensitive}} = \epsilon$, and a tolerance of 0.00001. This hyperparameter combination illustrates the effectiveness of low regularization and tight tolerance in driving model precision. Notably, PAR models exhibit higher $\overline{\text{MAE}}$ than MLP for most outputs, indicating that MLP may be more suitable for tasks requiring extremely low error margins. For E in the synthetic dataset, PAR shows significant degradation, with $\overline{R^2} = 0.991$, $\overline{\text{MAE}} = 4.91$, and increased variability ($\sigma_{R^2} = 0.004$, $\sigma_{\text{MAE}} = 0.234$). This outlier mirrors similar behavior in MLP models, suggesting that synthetic E data presents challenges for both architectures, likely due to inherent complexity or inconsistencies in its generation. Compared to MLP models, PAR models use a simpler hyperparameter configuration, focusing mainly on C, ϵ, and $\epsilon_{\text{insensitive}}$, unlike MLP's broader set involving learning rates, hidden layers, and solvers. While this simplicity aids tuning, it may limit fine-grained optimization, as seen in the $\overline{\text{MAE}}$ comparison. In summary, PAR models demonstrate strong and consistent performance, especially with low regularization and tolerance values. However, MLP models achieve superior error minimization, making them more suitable for tasks requiring exceptionally high accuracy. The persistent challenge of modeling E in the synthetic dataset highlights the need for further investigation into this particular case.

Table 4.2: The results of best models found using grid search and cross validation procedures, for PAR (R^2 – average value of coefficient of determination across folds, σ_{R^2} – standard deviation of R^2 score across folds, $\bar{\text{MAE}}$ – mean absolute error, σ_{MAE} – standard deviation of MAE across folds, ϕ – activation function, LR - init. – Initial learning rate, LR – learning rate type, $L2$ – regularization rate).

Data	**Out**	$\overline{R^2}$	σ_{R^2}	$\overline{\textbf{MAE}}$	$\sigma_{\textbf{MAE}}$	C	ϵ	**Fit intercept**	$\epsilon_{\textbf{insensitive}}$	**Tolerance**
	E_1	0.999	$1 \cdot 10^{-3}$	$4.2 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	0.5	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-4}$
	E_2	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$6 \cdot 10^{-3}$	0.1	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-4}$
	E_3	0.999	$1 \cdot 10^{-3}$	$3.8 \cdot 10^{-2}$	$2 \cdot 10^{-3}$	0.5	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-4}$
real	E_4	0.999	$1 \cdot 10^{-3}$	$4.6 \cdot 10^{-2}$	$4 \cdot 10^{-3}$	0.1	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_5	0.999	$1 \cdot 10^{-3}$	$4.5 \cdot 10^{-2}$	$4 \cdot 10^{-3}$	0.1	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_6	0.999	$1 \cdot 10^{-3}$	$4.1 \cdot 10^{-2}$	$1.5 \cdot 10^{-2}$	1	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E	0.999	$1 \cdot 10^{-3}$	$4.4 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	0.1	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_1	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$4 \cdot 10^{-3}$	0.1	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_2	0.999	$1 \cdot 10^{-3}$	$3.7 \cdot 10^{-2}$	$7 \cdot 10^{-3}$	1	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_3	0.999	$1 \cdot 10^{-3}$	$3.8 \cdot 10^{-2}$	$2 \cdot 10^{-3}$	10	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-5}$
synth	E_4	0.999	$1 \cdot 10^{-3}$	$4.4 \cdot 10^{-2}$	$4 \cdot 10^{-3}$	10	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_5	0.999	$1 \cdot 10^{-3}$	$4.2 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	0.1	0.1	False	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_6	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	0.5	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E	0.999	$1 \cdot 10^{-3}$	$4.9 \cdot 10^{-2}$	$6 \cdot 10^{-3}$	1	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_1	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	1	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_2	0.999	$1 \cdot 10^{-3}$	$3.7 \cdot 10^{-2}$	$4 \cdot 10^{-3}$	0.1	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_3	0.999	$1 \cdot 10^{-3}$	$3.8 \cdot 10^{-2}$	$2 \cdot 10^{-3}$	0.5	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
sim	E_4	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$7 \cdot 10^{-3}$	0.5	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_5	0.999	$1 \cdot 10^{-3}$	$4.1 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	0.1	0.1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E_6	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	0.1	0.1	True	$\epsilon^2_{\text{insensitive}}$	$1 \cdot 10^{-5}$
	E	0.991	$4 \cdot 10^{-3}$	4.91	0.234	0.1	1	True	$\epsilon_{\text{insensitive}}$	$1 \cdot 10^{-5}$

Table 4.3 provides an overview of the results for SVR models, where the performance is evaluated across real, synthetic, and simulated datasets. The results consistently show $\overline{R^2}$ values of 0.999 across almost all outputs, with a standard deviation (σ_{R^2}) of 0.001, reflecting robust and highly consistent performance across folds. However, the $\overline{\text{MAE}}$ values display greater variability, ranging from 0.036 to 0.044 for most outputs, with a

notable outlier for E in the simulated dataset, where $\overline{\text{MAE}} = 4.423$ and $\sigma_{\text{MAE}} = 0.262$. This pattern mirrors challenges observed in earlier tables with the E output in synthetic and simulated datasets, highlighting it as a consistently problematic case across different modeling frameworks. The hyperparameter configuration across all models is remarkably uniform, with $C = 10$ for most cases and a polynomial kernel of degree 2 ($Degree = 2$) being employed universally. The ε-insensitive loss is fixed at $\varepsilon = 0.1$, while the kernel scale parameter (γ) is set to "scale" for all configurations. This uniformity simplifies model interpretation and underscores the reliability of this specific SVR configuration across diverse data types. The best-performing model, based on the lowest $\overline{\text{MAE}}$, is found in the synthetic dataset for $E1$, where $\overline{\text{MAE}} = 0.036$ and $\sigma_{\text{MAE}} = 0.003$. This model uses $C = 10$, $Degree = 2$, $\varepsilon = 0.1$, and a linear kernel. These parameters align closely with the broader trend observed in the table, reinforcing the effectiveness of this specific hyperparameter set. Compared to the PAR and MLP models discussed in earlier tables, the SVR models show slightly higher $\overline{\text{MAE}}$ values for most outputs. However, their performance remains competitive, particularly given their consistency in achieving perfect or near-perfect $\overline{R^2}$ values. The challenges with the E output, particularly in simulated datasets, persist, as evidenced by the significant increase in $\overline{\text{MAE}}$ and variability (σ_{MAE}) for this case. This outlier may suggest fundamental differences in the characteristics of the E output in synthetic and simulated datasets that all modeling approaches struggle to address effectively. In summary, the SVR models exhibit strong, consistent performance with minimal variability across outputs, driven by a uniform hyperparameter configuration. While slightly less precise than MLP models in terms of minimizing $\overline{\text{MAE}}$, SVR demonstrates robustness across datasets. The recurring challenges with the E output in synthetic and simulated datasets warrant further investigation, as this issue transcends modeling frameworks. The hyperparameter settings observed here, particularly for the best-performing model (synthetic $E1$), provide a reliable baseline for future SVR implementations.

Table 4.3: The results of best models found using grid search and cross validation procedures, for SVR (R^2 – average value of coefficient of determination across folds, σ_{R^2} – standard deviation of R^2 score across folds, $\bar{\text{MAE}}$ – mean absolute error, σ_{MAE} – standard deviation of MAE across folds, ϕ – activation function, LR - init. – Initial learning rate, LR – learning rate type, $L2$ – regularization rate).

Data	**Out**	$\overline{R^2}$	σ_{R^2}	$\overline{\textbf{MAE}}$	$\sigma_{\textbf{MAE}}$	C	**Degree**	ε	γ	**Kernel**
	E_1	0.999	$1 \cdot 10^{-3}$	$4.1 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	10	2			
	E_2	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	1	2			
	E_3	0.999	$1 \cdot 10^{-3}$	$4.4 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	10	2			
real	E_4	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$6 \cdot 10^{-3}$	10	2			
	E_5	0.999	$1 \cdot 10^{-3}$	$4.1 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	10	2			
	E_6	0.999	$1 \cdot 10^{-3}$	$3.8 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	10	2			
	E	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	10	2			
	E_1	0.999	$1 \cdot 10^{-3}$	$3.6 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	10	2			
	E_2	0.999	$1 \cdot 10^{-3}$	$3.8 \cdot 10^{-2}$	$4 \cdot 10^{-3}$	1	2			
	E_3	0.999	$1 \cdot 10^{-3}$	$4.2 \cdot 10^{-2}$	$1.2 \cdot 10^{-2}$	1	2			
synth	E_4	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	1	2	0.1	scale	linear
	E_5	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$6 \cdot 10^{-3}$	10	2			
	E_6	0.999	$1 \cdot 10^{-3}$	$3.7 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	10	2			
	E	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$4 \cdot 10^{-3}$	10	2			
	E_1	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$2 \cdot 10^{-3}$	10	2			
	E_2	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	10	2			
	E_3	0.999	$1 \cdot 10^{-3}$	$4.1 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	10	2			
sim	E_4	0.999	$1 \cdot 10^{-3}$	$4.1 \cdot 10^{-2}$	$7 \cdot 10^{-3}$	10	2			
	E_5	0.999	$1 \cdot 10^{-3}$	$4.0 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	1	2			
	E_6	0.999	$1 \cdot 10^{-3}$	$3.9 \cdot 10^{-2}$	$3 \cdot 10^{-3}$	10	2			
	E	0.992	$1 \cdot 10^{-3}$	4.423	0.262	10	2			

The results for the XGB models, given in Table 4.4, highlight their exceptional performance across most outputs and datasets, with $\overline{R^2}$ values frequently nearing or achiev-

ing 1 and very low variability ($\sigma_{R^2} \leq 0.002$). These outcomes underline XGB's robustness and its ability to consistently approximate ground truth relationships across varied datasets (real, synthetic, and simulated). The $\overline{\text{MAE}}$ values show more variability, ranging from 0.035 to 9.985, indicating generally high precision, though certain outputs—particularly E—introduce challenges with consistently higher errors. Across all datasets, the majority of models share uniform hyperparameters. The subsample ratio ($Subsample = 1$) and learning rate ($\alpha = 0.1$) are fixed for nearly all configurations, along with $X_L^{\%} = X_N^{\%} = X_T^{\%} = 1$, indicating the reliability of these choices. The maximum tree depth ($\max(d)$) is predominantly set to 6 or 7, balancing complexity and overfitting risk. Regularization terms L_1 and L_2 are sparsely used, typically $L_1 = 0$ or $L_2 = 0.5$, and play a minimal role given the already high $\overline{R^2}$ values. For the simulated dataset, $E5$ achieves the best overall performance, with $\overline{R^2} = 0.999$, $\sigma_{R^2} = 0.001$, $\overline{\text{MAE}} = 0.035$, and $\sigma_{\text{MAE}} = 0.012$. This model uses $Subsample = 1$, $\max(d) = 6$, and minimal regularization ($L_1 = 0$, $L_2 = 0.1$). However, output E remains challenging across datasets, with high $\overline{\text{MAE}}$ values (e.g., 8.05 for real, 9.985 for synthetic, 8.575 for simulated), suggesting intrinsic complexity or noise. Compared to SVR, PAR, and MLP models, XGB shows higher error margins for E, despite robust $\overline{R^2}$ values. While SVR and PAR often yield lower $\overline{\text{MAE}}$ for many outputs, XGB's scalability and ability to handle nonlinearities contribute to its broader applicability. For simpler outputs (e.g., $E4, E5, E6$), XGB is competitive, often matching or outperforming other models. For instance, in the synthetic dataset, $E6$ achieves perfect $\overline{R^2}$ with minimal error ($\overline{\text{MAE}} = 0.062$). The consistent performance gap for E across models confirms it as a particularly difficult case. Higher $\overline{\text{MAE}}$ and σ_{MAE} reflect instability, even with consistent hyperparameters. Overall, XGB delivers reliable performance for most outputs, as seen in the optimal simulated $E5$ model, underscoring the utility of modest tree depth, full subsampling, and minimal regularization. However, for complex outputs like E, further hyperparameter tuning or data preprocessing may be needed to reduce error, as XGB, while strong in general, struggles to minimize errors in such cases.

Table 4.4: The results of best models found using grid search and cross validation procedures, for XGB (R^2 – average value of coefficient of determination across folds, σ_{R^2} – standard deviation of R^2 score across folds, $\bar{\text{MAE}}$ – mean absolute error, σ_{MAE} – standard deviation of MAE across folds, ϕ – activation function, LR - init. – Initial learning rate, LR – learning rate type, $L2$ – regularization rate).

Data	**Out**	$\overline{R^2}$	σ_{R^2}	$\overline{\textbf{MAE}}$	$\sigma_{\textbf{MAE}}$	$X_L^{\%}$	$X_N^{\%}$	$X_T^{\%}$	α	**max**(d)	n	L_1	L_2	**Subsample**
	E_1	0.999	$2 \cdot 10^{-3}$	0.409	0.154	1		1	$1 \cdot 10^{-1}$	7	100	0	0.5	0.75
	E_2	0.966	$4 \cdot 10^{-2}$	0.627	0.415	1		1	$1 \cdot 10^{-1}$	5	100	0	1	1
	E_3	0.948	$6.3 \cdot 10^{-2}$	0.601	$4.8 \cdot 10^{-2}$	0.75		1	$1 \cdot 10^{-1}$	3	100	0.5	1	1
real	E_4	0.999	$2 \cdot 10^{-3}$	$7.6 \cdot 10^{-2}$	$5.2 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	7	100	0	1	1
	E_5	0.997	$2 \cdot 10^{-3}$	$7.4 \cdot 10^{-2}$	$1.7 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	6	100	1	1	1
	E_6	1	0	$5.2 \cdot 10^{-2}$	$2.1 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	7	100	0	0.5	1
	E	0.928	$1.3 \cdot 10^{-2}$	8.05	1.663	1		1	$1 \cdot 10^{-1}$	6	100	0	0.5	0.5
	E_1	0.983	$2.4 \cdot 10^{-2}$	0.868	0.687	1		1	$1 \cdot 10^{-1}$	7	50	0.5	0.5	1
	E_2	0.959	$5.5 \cdot 10^{-2}$	0.791	0.635	1		1	$1 \cdot 10^{-1}$	5	100	0.5	0	1
	E_3	0.993	$1.7 \cdot 10^{-2}$	0.189	0.2	1		1	$1 \cdot 10^{-1}$	7	100	0.5	$1 \cdot 10^{-1}$	1
synth	E_4	0.999	$1 \cdot 10^{-3}$	$6.3 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$	1	1	1	$1 \cdot 10^{-1}$	7	100	0.5	$1 \cdot 10^{-1}$	1
	E_5	0.999	$1 \cdot 10^{-3}$	$8.7 \cdot 10^{-2}$	$2.9 \cdot 10^{-2}$	1		$7.5 \cdot 10^{-1}$	$1 \cdot 10^{-1}$	5	100	$1 \cdot 10^{-1}$	0.5	1
	E_6	1	0	$6.2 \cdot 10^{-2}$	$2.3 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	7	100	1	0	1
	E	0.879	$3.3 \cdot 10^{-2}$	9.985	2.312	1		$7.5 \cdot 10^{-1}$	$1 \cdot 10^{-1}$	3	100	1	0.5	0.5
	E_1	1	$1 \cdot 10^{-3}$	0.25	$7.4 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	5	100	0	0.5	1
	E_2	0.999	$4 \cdot 10^{-3}$	0.15	$5.6 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	6	100	0	1	1
	E_3	0.999	$2 \cdot 10^{-3}$	$9.6 \cdot 10^{-2}$	$3.9 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	7	100	0	$1 \cdot 10^{-1}$	1
sim	E_4	0.999	$2 \cdot 10^{-3}$	$4.8 \cdot 10^{-2}$	$4.6 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	6	100	0	$1 \cdot 10^{-1}$	1
	E_5	0.999	$1 \cdot 10^{-3}$	$3.5 \cdot 10^{-2}$	$1.2 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	6	100	0	$1 \cdot 10^{-1}$	1
	E_6	1	0	$4.9 \cdot 10^{-2}$	$1.9 \cdot 10^{-2}$	1		1	$1 \cdot 10^{-1}$	6	100	0.5	$1 \cdot 10^{-1}$	1
	E	0.968	$4 \cdot 10^{-3}$	8.575	0.384	1		1	$1 \cdot 10^{-1}$	3	100	$1 \cdot 10^{-1}$	0	0.5

To summarize results from PAR, SVR, MLP, and XGB models, a comprehensive assessment of different machine learning approaches applied to real, synthetic, and simulated datasets can be given. Across these models, a number of consistent trends, strengths, and challenges emerge, offering valuable insights into their performance characteristics and suitability for various tasks. Across all models and datasets, the average R^2 values ($\overline{R^2}$) are consistently high, typically reaching 0.999 for most outputs. This demonstrates all models' ability to explain the variance in the data effectively, regardless of the dataset type (real, synthetic, or simulated). However, XGB occasionally shows slightly lower R^2 values (e.g., $\overline{R^2} = 0.928$ for real E and $\overline{R^2} = 0.879$ for synthetic E), indicating reduced model fit for more complex outputs. While R^2 remains consis-

tently high, the mean absolute error ($\overline{\text{MAE}}$) and its variability (σ_{MAE}) differ significantly across models. MLP achieves the lowest $\overline{\text{MAE}}$ values overall, with outputs like synthetic $E5$ attaining $\overline{\text{MAE}} = 0.001$. PAR and SVR show slightly higher $\overline{\text{MAE}}$ values but maintain robust performance, with minimal variability. XGB demonstrates greater variability in $\overline{\text{MAE}}$, especially for E, where errors spike significantly (e.g., $\overline{\text{MAE}} = 9.985$ for synthetic E). This suggests that while XGB is highly effective for general tasks, it struggles with outputs requiring extreme precision. Across all models and datasets, the E output consistently exhibits higher $\overline{\text{MAE}}$ values and greater variability. This is particularly evident for synthetic and simulated datasets, where the performance of all models degrades. This recurring trend suggests inherent challenges in modeling E, possibly due to its complexity, noise, or irregularities in the dataset itself. Each model relies on distinct hyperparameter configurations to achieve optimal performance. PAR models show consistent results with small regularization (C), fixed $\epsilon = 0.1$, and low tolerances (0.00001). SVR models excel with linear kernels, small ε, and moderate C values (e.g., $C = 10$). MLP models benefit from constant learning rates, small hidden layer sizes (e.g., 10 neurons), and the use of the ‘lbfgs‘ solver. XGB models rely on modest tree depths (6 or 7), subsampling, and minimal regularization, though occasional outliers (e.g., synthetic E) highlight sensitivity to task-specific configurations. The simulated datasets generally exhibit slightly better performance than synthetic datasets, except for outputs like E. This aligns with the clarified distinction between "synthetic" and "simulated" datasets, as the synthetic data likely retains stronger consistency with the real data distribution. MLP demonstrates the best overall precision, achieving the lowest $\overline{\text{MAE}}$ values across outputs and datasets, making it ideal for tasks requiring extreme accuracy. SVR and PAR show competitive performance with high R^2 values and moderate errors, excelling in simplicity and stability. XGB offers strong generalization and scalability but exhibits higher error variability for complex outputs like E, suggesting it is better suited for tasks prioritizing interpretability and computational efficiency over extreme precision. The results highlight the strengths and limitations of each model. MLP emerges as the most precise option, while SVR and PAR provide robust and

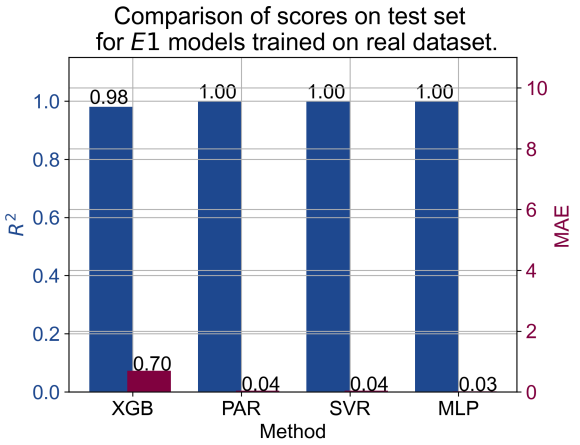
stable alternatives. XGB demonstrates strong performance for general tasks but struggles with complex outputs like E. The recurring challenges with E across all models and datasets underscore the need for further investigation into this output's characteristics and the potential benefits of task-specific tuning or data preprocessing. These findings emphasize the importance of selecting models based on task requirements, balancing precision, scalability, and robustness.

4.3.2 Results of developed ML models on prepared test data

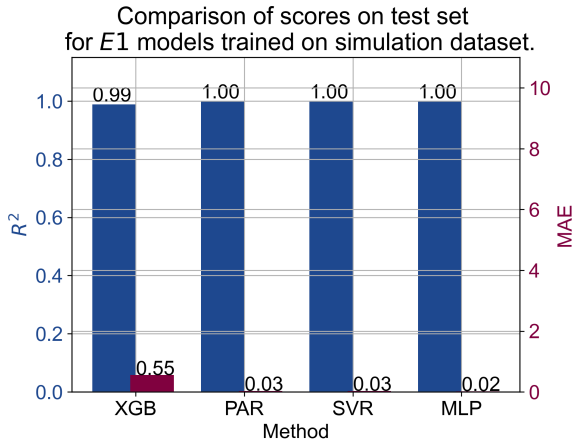
While the hyperparameter analysis is important to determine which exact model configuration should be selected for deployment, the performance metrics derived from training and validation are not necessarily fully indicative of real-world behavior. In practical applications, factors such as data noise, distribution shifts, and measurement inaccuracies can influence outcomes. Because of this, additional testing is essential to validate model robustness and generalization. To this end, testing is conducted on a completely separate testing dataset, distinct from both the training and validation datasets. This dataset was collected using the real IRM in a controlled laboratory environment, following the same methodology described in detail earlier in this thesis. The models evaluated and scored in the following section are those trained on their respective datasets—real, synthetic, or simulated—using the best-performing hyperparameters identified in the previous section. These models are then applied to the new test dataset to evaluate their generalization capabilities.

In other words, the results depicted in the following figures summarize the performance metrics of four machine learning methods—XGB, PAR, SVR, and MLP—when tested on data originating from three sources: real, simulated, and synthetic datasets. The performance of each method is assessed using two key evaluation metrics: the coefficient of determination (R^2) and the mean absolute error (MAE). Figure 4.11 specifically presents the evaluation results for the energy of the first joint. In the real-world dataset, all methods except XGB achieve a perfect R^2 score of 1.0, meaning they completely account for the variability in the target variable. XGB, while still performing well, trails

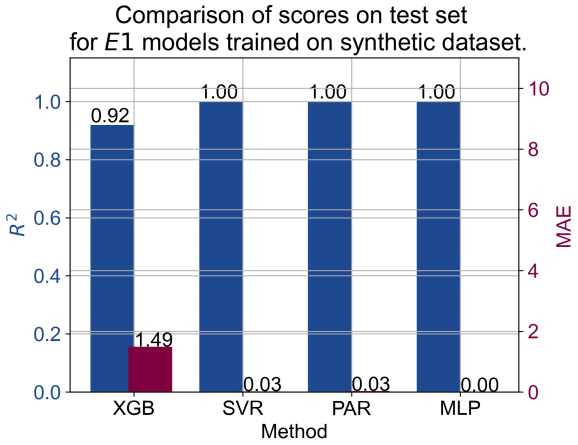
slightly with an R^2 of 0.98. In terms of MAE, XGB demonstrates a higher error of 0.7, whereas PAR, SVR, and MLP maintain minimal errors of 0.04, 0.04, and 0.03, respectively, indicating higher predictive accuracy. For the simulated dataset, a similar pattern emerges. PAR, SVR, and MLP again achieve perfect R^2 scores of 1.0, showing strong consistency in model performance. XGB, however, shows a drop in performance with an R^2 of 0.92. The MAE for XGB rises significantly to 1.49, contrasting with the low errors of 0.03 for both PAR and SVR, and an error of 0.0 for MLP, which effectively predicts the target values without measurable deviation. In the synthetic dataset (Figure 4.11c), the trend continues. PAR, SVR, and MLP sustain their perfect R^2 scores of 1.0, while XGB slightly improves compared to the simulated dataset, reaching an R^2 of 0.99. For MAE, XGB records a moderate error of 0.55, while PAR, SVR, and MLP again display very low errors of 0.03, 0.03, and 0.02, respectively. A clear and consistent trend across all datasets is the superior performance of PAR, SVR, and MLP, which not only maintain perfect R^2 scores but also exhibit minimal MAE values, underscoring their precision and robustness. These results suggest that these methods are highly reliable in capturing the underlying data relationships and generating accurate predictions. In contrast, XGB, while still delivering high R^2 values close to 1.0, consistently underperforms relative to the other methods, particularly in terms of MAE, with errors that are significantly higher—especially in the simulated dataset. This consistent underperformance in terms of MAE indicates that while XGB is effective at modeling general trends in the data, its predictions deviate more from the actual values compared to those of PAR, SVR, and MLP. The observed performance patterns emphasize the reliability and stability of PAR, SVR, and MLP across different datasets, suggesting their suitability for scenarios where high precision and low error margins are critical for success.



(a) Test scores on real-world dataset, using E_1 as output



(b) Test scores on simulated dataset, using E_1 as output



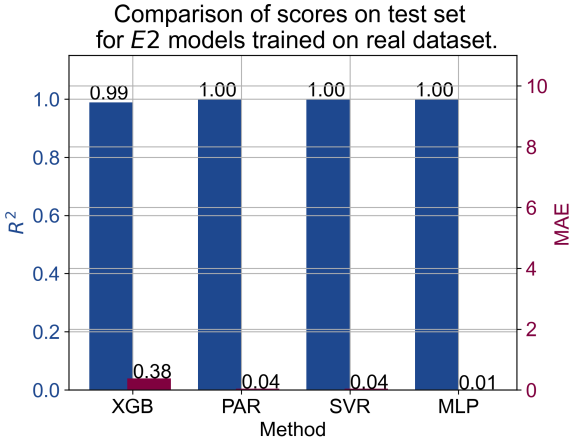
(c) Test scores on synthetic dataset, using E_1 as output

Figure 4.11: Scores on the test set for E_1

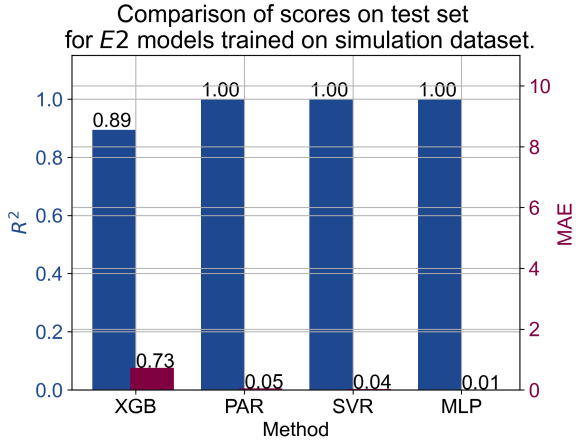
Figure 4.12 presents performance metrics for E_2 models, evaluated on test sets from three distinct datasets: real-world (4.12a), simulated (4.12b), and synthetic (4.12c). The R^2 scores indicate that PAR, SVR, and MLP maintain a perfect value of 1.0 across all datasets, reflecting their ability to fully capture the variability in the data and model the underlying relationships with complete accuracy and reliability. This consistent performance highlights their strong generalization capabilities across varying data distributions. In contrast, the XGB method achieves an R^2 of 0.89, which, while still high, is noticeably lower than the perfect scores of the other models and indicates some loss in predictive fidelity, suggesting that XGB may be less effective in capturing cer-

tain data characteristics. In terms of MAE, XGB records a value of 0.73, significantly higher than the corresponding errors of PAR, SVR, and MLP, which are 0.05, 0.04, and 0.01, respectively, clearly highlighting XGB's comparatively reduced precision and the greater deviation of its predictions from actual values. In the real dataset, this pattern remains consistent: PAR, SVR, and MLP again achieve perfect R^2 scores of 1.0, reaffirming their stability and precision in handling real-world data, while XGB attains a slightly reduced R^2 of 0.99. Though close to 1.0, XGB's score still reflects a marginal decline in performance, reinforcing the notion that it may not match the accuracy of the other methods. Regarding MAE, XGB shows an error of 0.38, which is substantially higher than the values for PAR, SVR, and MLP—0.04, 0.04, and 0.01, respectively—confirming the superior precision of the latter methods in real-world conditions. This dataset reaffirms the robustness of PAR, SVR, and MLP, which continue to demonstrate high accuracy and consistency with minimal prediction error. For the simulation dataset, a similar pattern is observed: R^2 remains at 1.0 for PAR, SVR, and MLP, while XGB reaches a lower value of 0.88, again indicating reduced effectiveness in capturing data variability, possibly due to the simulated data's structural differences. The MAE for XGB increases notably to 1.16, whereas PAR, SVR, and MLP record much smaller errors of 0.03, 0.03, and 0.0, respectively, further illustrating the pronounced performance gap in precision and reinforcing the consistency of the other methods. These findings suggest that although XGB is capable of modeling general data patterns, it consistently falls short in prediction accuracy compared to the other methods, particularly for this dataset, where both the error magnitude and variability are elevated. Across all three datasets, a clear and consistent trend emerges in which PAR, SVR, and MLP exhibit exceptional stability and precision, achieving perfect R^2 scores and minimal MAE values, which confirms their robustness and strong generalization across varied data conditions and testing environments. In contrast, XGB, despite achieving relatively high R^2 scores close to 1.0, consistently underperforms in comparison, with higher MAE values reflecting less precise predictions and indicating potential limitations in handling certain data complexities. These observa-

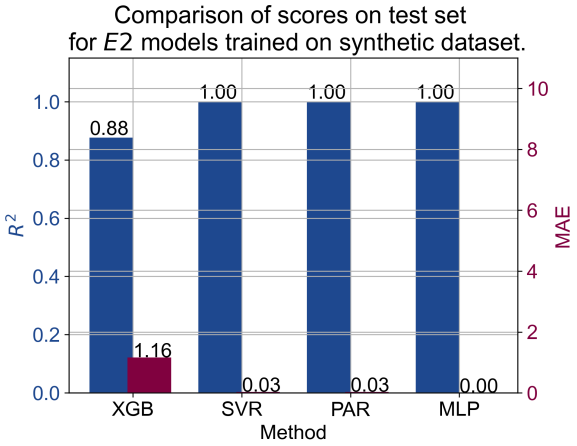
tions underscore the comparative reliability and effectiveness of PAR, SVR, and MLP for applications where both high accuracy and low prediction error are essential, particularly in scenarios where even small deviations in prediction can impact performance or decision-making outcomes.



(a) Test scores on real-world dataset, using E_2 as output



(b) Test scores on simulated dataset, using E_2 as output



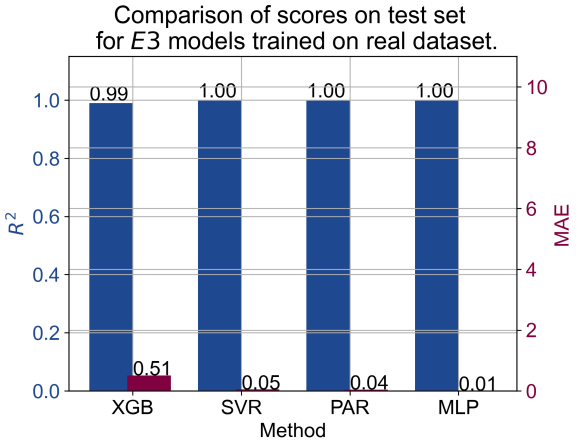
(c) Test scores on synthetic dataset, using E_2 as output

Figure 4.12: Scores on the test set for E_2

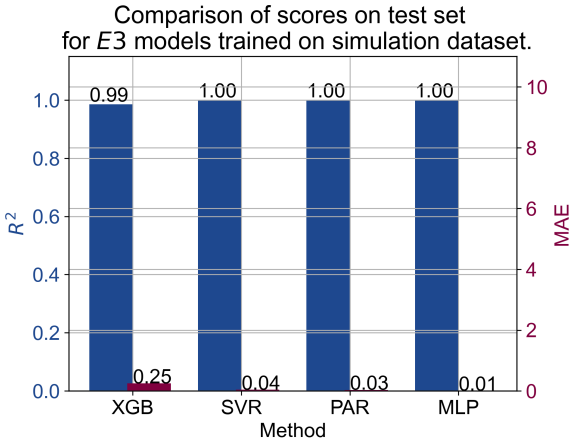
The results presented in Figure 4.13 display the performance of E_3 models evaluated on the test sets across three datasets. In the real dataset, the R^2 values indicate strong performance for all methods, with SVR, PAR, and MLP each achieving perfect scores of 1.0, demonstrating their capacity to fully capture the variability in the data. XGB attains a slightly lower R^2 of 0.99, which still reflects high accuracy but suggests a minor

reduction in its ability to model the data as precisely as the other methods. In terms of MAE, XGB demonstrates a relatively high value of 0.51, in contrast to the significantly lower errors of 0.05, 0.04, and 0.01 recorded by SVR, PAR, and MLP, respectively. These results highlight XGB's near-optimal modeling capability in terms of variance explained, while clearly indicating higher prediction error compared to the other methods, suggesting less precise individual predictions. The simulation dataset follows a similar pattern, with SVR, PAR, and MLP maintaining perfect R^2 values of 1.0, confirming their consistent ability to model simulated data accurately. XGB again shows a slight reduction in performance, achieving an R^2 of 0.91, indicating that it struggles more under simulated conditions. The MAE values reflect this trend, with XGB recording an error of 0.61, while SVR, PAR, and MLP exhibit minimal errors of 0.03, 0.03, and 0.0, respectively, further emphasizing the greater precision of the latter methods. This dataset confirms the robustness and reliability of SVR, PAR, and MLP across data types while illustrating a modest decline in XGB's accuracy and precision. In the synthetic dataset, SVR, PAR, and MLP maintain their strong and consistent performance, each achieving an R^2 of 1.0, demonstrating their ability to adapt and generalize effectively even in artificially generated data. XGB, however, exhibits a notable drop in R^2, achieving a significantly lower value of 0.25, suggesting that it faces considerable challenges in capturing the variability in this dataset. The corresponding MAE values show a similar pattern: XGB records a high error of 0.25, while SVR, PAR, and MLP maintain minimal errors of 0.04, 0.03, and 0.01, respectively, further confirming XGB's reduced precision in this context. This performance gap indicates that XGB is particularly sensitive to the characteristics of synthetic data, potentially due to differences in data distribution or complexity. The overall trend observed across the figures underscores the robustness and adaptability of SVR, PAR, and MLP, as evidenced by their consistent R^2 scores of 1.0 and minimal MAE values across all datasets, reflecting their stability and precision in various data conditions. In contrast, XGB, while demonstrating strong R^2 values in the real and simulation datasets, shows comparatively higher prediction errors and a significant decline in performance on the synthetic dataset, highlighting its limitations in

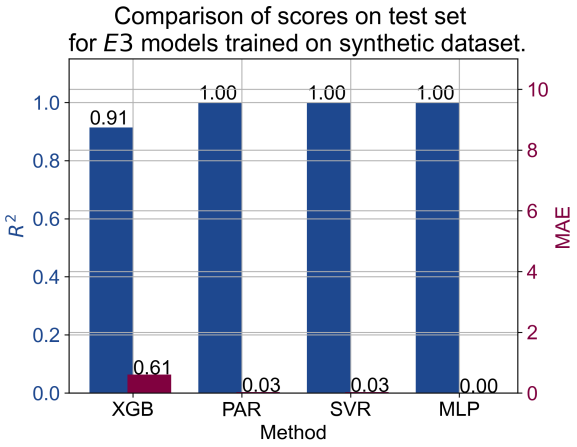
generalizing to different data types. These observations confirm the superior stability, precision, and reliability of SVR, PAR, and MLP across diverse data characteristics and testing environments.



(a) Test scores on real-world dataset, using E_3 as output



(b) Test scores on simulated dataset, using E_3 as output

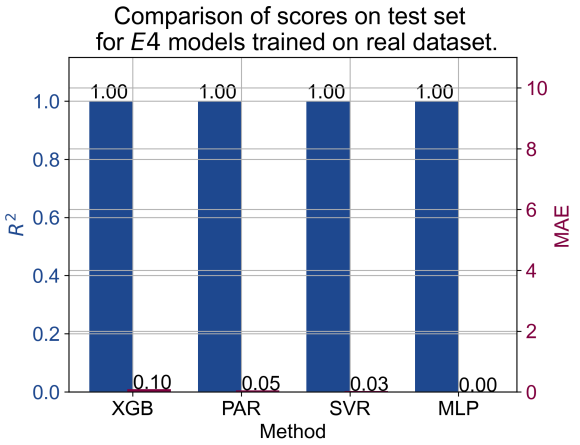


(c) Test scores on synthetic dataset, using E_3 as output

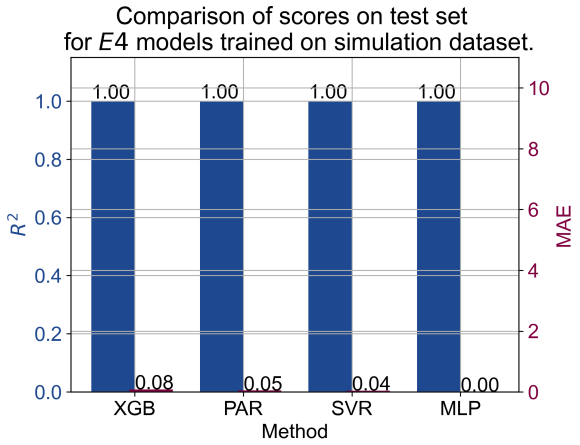
Figure 4.13: Scores on the test set for E_3

The same is shown in Figure 4.14 for the performance of E_4 models. In the real dataset, all methods demonstrate a perfect R^2 score of 1.0, indicating their ability to fully capture the variability in the dataset. However, the MAE values reveal distinct differences in precision. XGB records an MAE of 0.1, which, while modest, is higher than those of the other methods. PAR and SVR achieve slightly lower errors of 0.05 and 0.03, respectively, whereas MLP exhibits the lowest MAE at 0.0. This suggests that all meth-

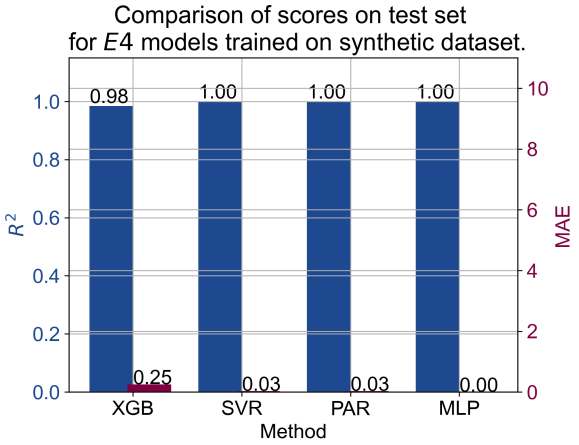
ods are highly effective at modeling the data, but MLP provides exceptional predictive precision. In the simulation dataset, the R^2 values again reflect near-perfect modeling capabilities, with SVR, PAR, and MLP maintaining scores of 1.0. XGB achieves an R^2 of 0.98, slightly trailing the other methods. The MAE results further distinguish the methods, as XGB records a value of 0.25, which is notably higher than those of SVR, PAR, and MLP, which achieve 0.03, 0.03, and 0.0, respectively. These findings demonstrate the consistent reliability of SVR, PAR, and MLP in minimizing error, whereas XGB exhibits slightly reduced precision in comparison. In the synthetic dataset, the R^2 values for all methods reach the optimal score of 1.0, illustrating their ability to fully explain the data's variability. The MAE values, however, reveal subtle variations in accuracy. XGB exhibits a slightly higher error of 0.08, compared to 0.05 for PAR, 0.04 for SVR, and 0.0 for MLP. This reinforces the observation that while all methods are capable of modeling the synthetic dataset with high accuracy, MLP consistently achieves the lowest prediction error across datasets. Overall, these figures highlight the remarkable robustness and precision of PAR, SVR, and MLP, which maintain perfect R^2 scores and minimal MAE values across all datasets. In contrast, XGB, while demonstrating strong modeling capabilities, exhibits slightly higher prediction errors, particularly in the simulation dataset. The consistency of these trends across datasets underscores the stability and accuracy of PAR, SVR, and MLP, with MLP emerging as the most precise method across all datasets.



(a) Test scores on real-world dataset, using E_4 as output



(b) Test scores on simulated dataset, using E_4 as output



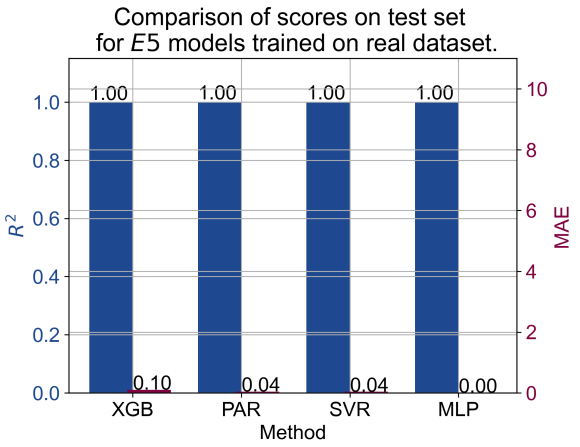
(c) Test scores on synthetic dataset, using E_4 as output

Figure 4.14: Scores on the test set for E_4

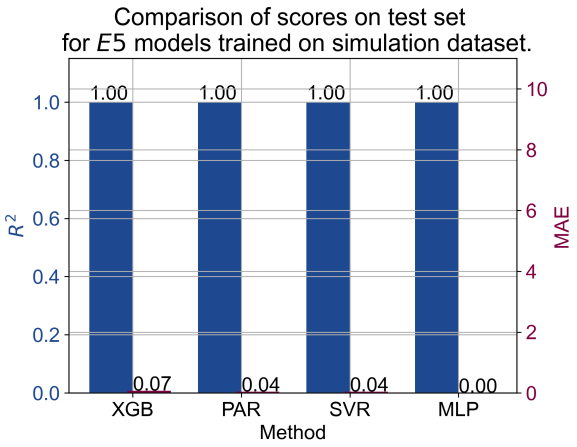
The performance of models targeting the energy of the fifth joint is presented in Figure 4.15, which illustrates their predictive capabilities across three distinct datasets: real-world, simulated, and synthetic. In the real dataset, all methods achieve the maximum possible R^2 score of 1.0, signifying their complete ability to explain the variance in the dataset and accurately model the underlying relationships. However, despite identical R^2 values, differences in prediction precision become evident when considering the MAE results. XGB reports an error of 0.1, which, although relatively small and indicative of good performance, is still higher than those recorded by the other models. PAR and SVR each achieve lower MAE values of 0.04, reflecting improved predictive accu-

racy and better alignment with the true values. MLP outperforms all others by achieving an MAE of 0.0, indicating perfect precision with no measurable deviation from the actual data. These results suggest that while all methods are capable of modeling the data effectively, MLP demonstrates a clear advantage in minimizing prediction errors, providing the most precise results in real-world conditions. In the simulation dataset, the trend continues with PAR, SVR, and MLP again maintaining perfect R^2 scores of 1.0, confirming their robustness in handling simulated data. XGB, however, records a slightly reduced R^2 of 0.94, indicating a minor decrease in its ability to fully capture data variability under simulated conditions. The MAE results further reinforce this observation: XGB reports an error of 0.2, noticeably higher than PAR and SVR, which both maintain minimal errors of 0.03, and MLP, which again achieves a perfect MAE of 0.0. This consistent pattern emphasizes that although XGB performs reasonably well overall, its predictive accuracy is less reliable compared to the other methods, particularly when applied to simulation data, where slight deficiencies in precision become more apparent. In the synthetic dataset, all methods once again achieve the maximum R^2 score of 1.0, indicating that they are fully capable of modeling the variance in artificially generated data. However, the MAE values continue to distinguish the models' levels of precision. XGB records an error of 0.07, which, although lower than in previous datasets, remains higher than the MAE values of 0.04 for both PAR and SVR, and significantly higher than MLP's error-free result of 0.0. These findings reaffirm the ability of PAR, SVR, and MLP to consistently deliver high-precision predictions across diverse data sources, with MLP standing out as the most accurate and stable method. Overall, the figures underscore the robustness and reliability of PAR, SVR, and MLP, which consistently achieve perfect R^2 scores and minimal MAE values, demonstrating their effectiveness in both variance explanation and precise prediction. In contrast, while XGB consistently delivers strong R^2 scores across datasets, its comparatively higher MAE values suggest reduced predictive precision, particularly in the simulation and synthetic datasets where subtle performance gaps become more pronounced. These findings illustrate the comparative stability, accuracy, and superior precision of PAR,

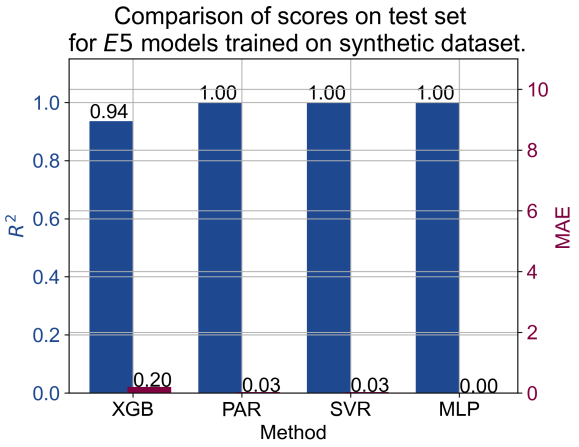
SVR, and MLP, with MLP emerging as the most reliable method for tasks requiring consistently high accuracy and minimal prediction error.



(a) Test scores on real-world dataset, using E_5 as output



(b) Test scores on simulated dataset, using E_5 as output

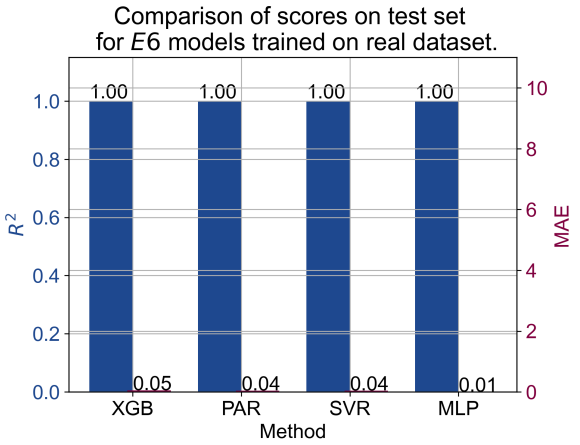


(c) Test scores on synthetic dataset, using E_5 as output

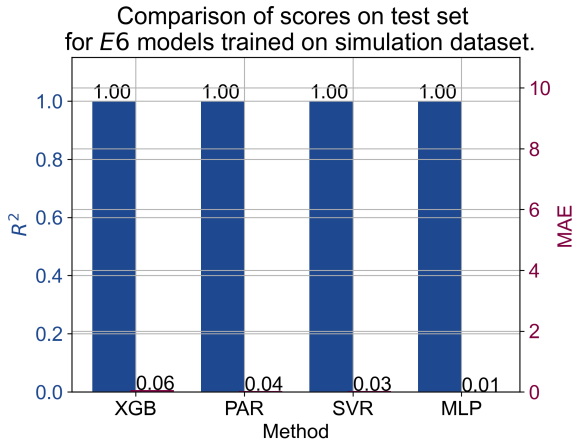
Figure 4.15: Scores on the test set for E_5

The performance of models for E_6 is illustrated in Figure 4.16, which presents their evaluation across real, simulated, and synthetic datasets. In the real dataset, all methods exhibit a perfect R^2 value of 1.0, indicating their complete capability to explain the variability in the data and accurately model its underlying structure. However, the MAE values reveal differences in predictive precision. XGB demonstrates an error of 0.05, which, while relatively low, is slightly higher than the errors reported by PAR and SVR, both of which achieve values of 0.04. MLP, consistent with trends observed in previ-

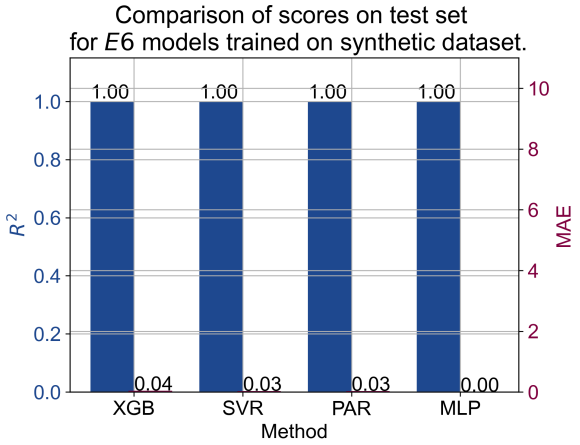
ous figures, achieves the lowest MAE at 0.01, showcasing its exceptional prediction accuracy and minimal deviation from true values. These results indicate that while all methods are effective in modeling real-world data, MLP maintains a clear advantage in minimizing prediction errors, offering the highest level of precision. The simulation dataset further reflects the strong modeling capabilities of all methods, with R^2 values uniformly perfect at 1.0 for every model, confirming their ability to capture variance in simulated data accurately. The MAE results, however, highlight subtle differences in precision among the methods. XGB records an error of 0.04, marginally higher than the values for PAR and SVR, both at 0.03, indicating a slight edge for these models in predictive accuracy. MLP once again demonstrates its robustness with an MAE of 0.0, reflecting perfect prediction accuracy in the simulation environment and further confirming its consistent reliability. This uniformity in performance across PAR, SVR, and MLP emphasizes their robustness and ability to generalize well to simulated data conditions. In the synthetic dataset, the R^2 scores remain perfect at 1.0 for all models, reinforcing their capacity to fully explain the variance in artificially generated data. However, as in the other datasets, the MAE values differentiate the models' levels of precision. XGB shows an MAE of 0.06, which is slightly higher than PAR's 0.04 and SVR's 0.03, again suggesting a modest reduction in predictive precision. MLP, maintaining its consistent trend from previous datasets, achieves an MAE of 0.01, underscoring its superior accuracy and its ability to minimize prediction error across all data types. Overall, the figures illustrate a robust and reliable performance by all methods, evidenced by their consistently perfect R^2 scores across real, simulated, and synthetic datasets. However, the MAE results reveal that while XGB performs well in terms of variance explanation, its prediction error is slightly higher than that of PAR, SVR, and particularly MLP, which consistently achieves the lowest MAE across all scenarios. These observations reaffirm the stability, accuracy, and precision of PAR, SVR, and MLP, with MLP standing out as the most reliable method for minimizing predictive errors, making it especially suitable for applications where high accuracy and low deviation are critical.



(a) Test scores on real-world dataset, using E_6 as output



(b) Test scores on simulated dataset, using E_6 as output



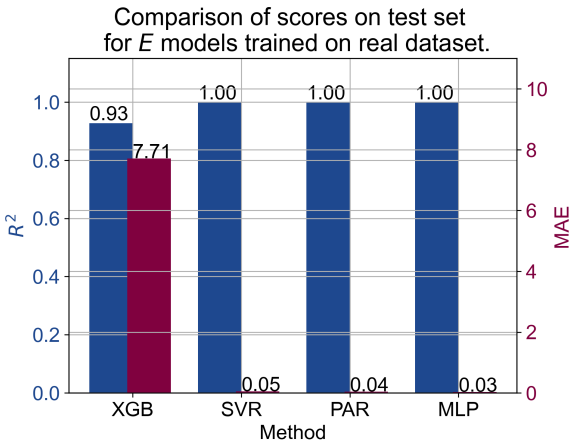
(c) Test scores on synthetic dataset, using E_6 as output

Figure 4.16: Scores on the test set for E_6

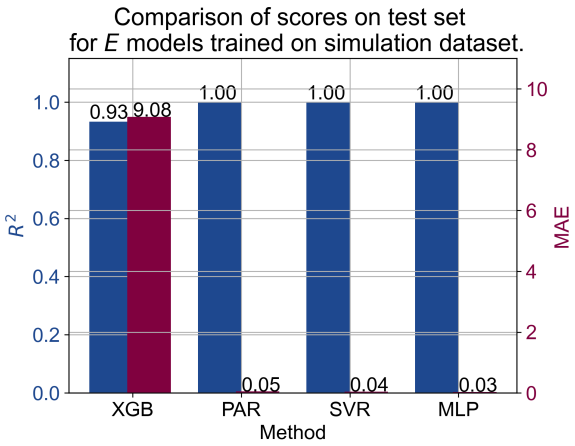
Finally, the R^2 and MAE scores for the total energy models are shown in Figure 4.17. The figure presents the performance of E models evaluated across real, simulation, and synthetic datasets using four methods: XGB, PAR, SVR, and MLP. The metrics R^2 and MAE offer insights into each model's ability to explain data variability and the precision of their predictions. In the real dataset, XGB achieves an R^2 of 0.93, which, although high, is slightly lower than the perfect scores of 1.0 recorded by PAR, SVR, and MLP, indicating a slight reduction in its explanatory power. The MAE values provide a clearer contrast in predictive precision. XGB records a relatively high error of 7.71, markedly exceeding the errors of PAR, SVR, and MLP, which achieve much lower

values of 0.05, 0.04, and 0.03, respectively. These results suggest that while XGB can model the general trends in the data, its individual predictions are substantially less accurate compared to the other methods. The simulation dataset reveals more pronounced performance differences. MLP and SVR maintain perfect R^2 scores of 1.0, indicating their continued ability to capture data variability with complete accuracy, while PAR achieves a slightly lower score of 0.99. XGB, however, sees a further drop in performance, recording an R^2 of 0.85, which reflects a significant decline in its explanatory power under simulated conditions. The MAE values align with this trend: XGB reports the highest error at 9.59, while PAR follows with a lower but still considerable error of 3.26. In contrast, MLP and SVR demonstrate superior accuracy, with much lower MAE values of 1.59 and 1.56, respectively. These figures highlight that MLP and SVR not only explain the data well but also maintain high precision, whereas XGB continues to struggle in producing accurate predictions. In the synthetic dataset, XGB's R^2 score improves slightly to 0.93, once again trailing behind the perfect scores achieved by PAR, SVR, and MLP, which maintain their ability to fully explain the variance in synthetic data. However, the MAE values sharply distinguish the models' predictive performance. XGB records a substantial error of 9.08, significantly higher than the errors of PAR, SVR, and MLP, which achieve 0.05, 0.04, and 0.03, respectively. This pattern reinforces the earlier observations that, despite retaining reasonable explanatory power, XGB's predictions are consistently less precise across all datasets. Overall, the figures underscore a clear and consistent trend: while XGB demonstrates commendable R^2 values, suggesting it can generally model the data's structure, its significantly higher MAE values across all datasets reveal a marked weakness in achieving precise predictions. Conversely, PAR, SVR, and MLP consistently excel in both explanatory power and prediction accuracy, with MLP and SVR performing particularly well and showing minimal variation in error across datasets. These results emphasize the reliability and robustness of MLP and SVR for applications where both high accuracy and low prediction error are critical, while also highlighting areas where XGB may require further hyperparameter tuning or methodological adjustments to improve its precision

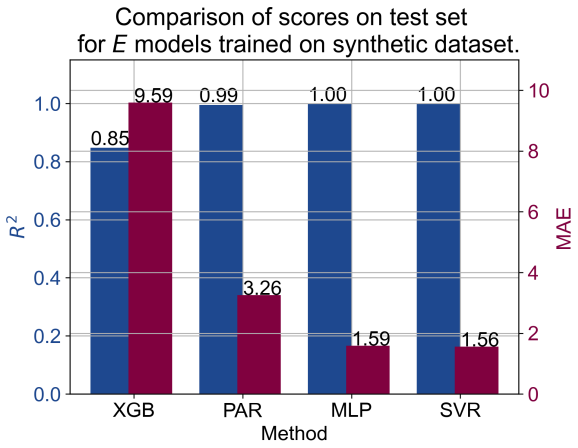
and reliability.



(a) Test scores on real-world dataset, using E as output



(b) Test scores on simulated dataset, using E as output



(c) Test scores on synthetic dataset, using E as output

Figure 4.17: Scores on the test set for E

Across all the figures provided, consistent trends emerge regarding the comparative performance of the models evaluated on real, simulation, and synthetic datasets. The models XGB, PAR, SVR, and MLP were assessed using R^2, measuring the proportion of explained variance, and MAE, which quantifies prediction error. These metrics collectively highlight the strengths and limitations of each method. In terms of R^2, all models generally perform well, with PAR, SVR, and MLP frequently achieving perfect scores of 1.0 across datasets. XGB, while showing strong R^2 values in most cases, tends to lag slightly behind, particularly in synthetic and simulation datasets, where its

scores are occasionally below 0.9. This suggests that XGB, although capable of explaining a substantial portion of the variance, does not consistently match the robustness of the other methods in this regard. The MAE values reveal more pronounced differences between the models. PAR, SVR, and MLP consistently achieve low MAE values, indicating high predictive precision. MLP, in particular, emerges as the most accurate model overall, as it not only maintains perfect R^2 scores across nearly all cases but also achieves the lowest MAE values, often approaching zero. SVR and PAR also demonstrate strong predictive capabilities, with consistently low errors across datasets, making them reliable choices as well. XGB, however, exhibits notably higher MAE values in many cases, particularly in the simulation and synthetic datasets, where its prediction errors are considerably larger. This highlights a key limitation in XGB's predictive precision, despite its strong explanatory power. The overall trends suggest that while XGB is a robust model in certain contexts, its performance is less reliable when precision is critical. Conversely, PAR, SVR, and especially MLP, display exceptional consistency and reliability across all datasets. Among these, MLP stands out as the best model overall, combining perfect R^2 scores with minimal MAE values, demonstrating both excellent explanatory power and superior predictive accuracy. This makes MLP the most versatile and reliable choice for modeling tasks across diverse data characteristics, which is why it was selected for further analysis and comparison.

4.4 Comparison of selected models for fitness function

To compare the performance of the selected MLP algorithm trained on the synthetic data, we will compare it to the results of equations obtained with the LE. The comparison is given in Figure 4.18.

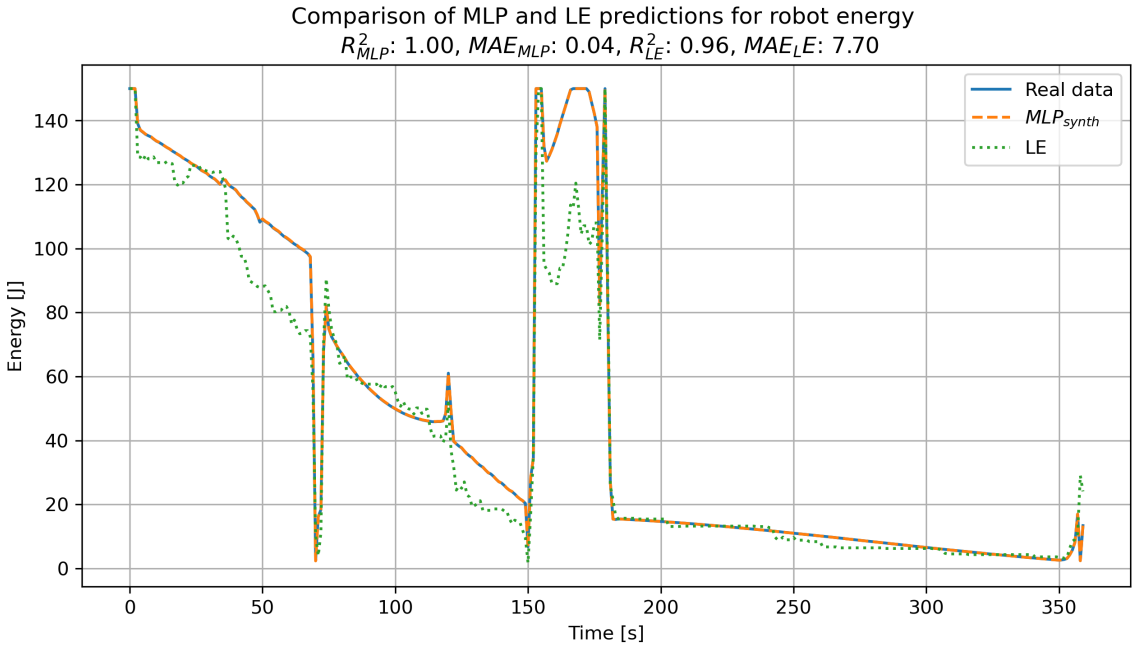


Figure 4.18: An example of the score comparison between the ML-based and LE model, section of the test data

As the figure shows, the LE is significantly less precise compared to the ML-based MLP model. The MLP model follows the real data nearly perfectly, while LE shows significant errors. There are many reasons why LE model may have a more prominent error – such as the precision of the measurements, and other internal coefficients of the robot.

Another issue is the computational complexity of the equations obtained with LE. The prediction using these equations (sans loading time of functions into memory) is approximately 1.06 seconds. For comparison, the MLP runs a prediction in milliseconds. While the difference may not seem significant, the fact that these equations are going to be used as a fitness function needs to be taken into the account. As the calculation is repeated tens of thousands of time, this difference becomes very significant. Another issue is the size of the models. While the stored MLP model takes approximately 300 KB, the LE models take up 39.6 MB of memory. So while the LE model may be further tuned to achieve higher precision, even then it would not be appropriate for use as a fitness function. Due to the size of LE equations, they are provided as link in

Appendix E.

4.5 Optimization results

4.5.1 Performance of different configurations of GA algorithms

The optimization performance is measured by generating random parameter values for each path to simulate diverse and realistic scenarios. Both the starting and ending points of the path, defined within the joint space of the robot, are randomly selected within feasible and physically realistic bounds. Additionally, the duration of each path is randomized within a range of 1 to 3 seconds, ensuring variability in path length and execution time. Following this initialization, the optimization process is carried out on a total of 500 of these randomly generated paths to evaluate the effectiveness and robustness of the optimization algorithms under varied conditions. All algorithms are executed using a uniform population size of 100 individuals, evolved over 250 generations, providing a consistent basis for performance comparison across different configurations.

Table 4.5 presents the performance scores of various GA (Genetic Algorithm) variations using a single-objective optimization function. Three distinct recombination methods—Random, Average, and Differential—are assessed, each under different configurations of crossover probability ($P(C)$), mutation probability ($P(M)$), and, specifically for the Differential method, scaling factor (F). The evaluation metrics include the percentage improvement over the initial, randomly generated path and the corresponding standard deviation (σ) to reflect consistency across trials. Improvement is computed as the relative gain compared to the performance of the path before optimization begins. The tested values for crossover probability $P(C)$ are 90% and 95%, while mutation probability $P(M)$ is evaluated at 1% and 5%. For the Differential recombination method, which replaces mutation with the scaling factor F, two values are tested: 0.5 and 1.5. The Random recombination method achieves its highest improvement of 51.96% when configured with $P(C) = 0.95$ and $P(M) = 0.05$, indicating that a higher

crossover rate combined with a moderate mutation rate facilitates more effective exploration and exploitation of the solution space. However, as the mutation rate is reduced to 0.01, the performance drops significantly, with a minimum improvement of 34.31% observed at $P(C) = 0.9$, underscoring the importance of maintaining a sufficient mutation level for this method. The standard deviation for the Random method remains low, between 0.21 and 0.24, suggesting stable and repeatable performance across different optimization runs. For the Average recombination method, the highest recorded improvement is 50.35%, achieved with $P(C) = 0.9$ and $P(M) = 0.05$. This result indicates that a slightly lower crossover rate, in combination with a moderate mutation probability, can also produce strong optimization outcomes. Notably, the performance of this method drops sharply when $P(M)$ is decreased to 0.01, similar to the Random method, highlighting a dependence on mutation for effective search. The standard deviation for Average recombination is slightly lower, ranging from 0.19 to 0.22, indicating marginally better consistency and stability than the Random method. The Differential recombination method, which replaces the mutation operation with a scaling factor F, generally shows lower performance compared to the other two methods. Its highest improvement is 39.33%, achieved under the configuration $P(C) = 0.95$ and $F = 0.05$. When the scaling factor is increased to 0.15, performance declines modestly, suggesting that smaller scaling factors are more advantageous for this recombination approach. The standard deviation values for the Differential method are slightly higher, between 0.22 and 0.25, indicating greater variability in outcomes and potentially less reliability across trials. Overall, the Random recombination method demonstrates the highest peak performance in terms of percentage improvement, though the Average method produces comparable results with slightly greater consistency and lower variability. The Differential method, despite its lower overall performance, provides valuable insights into the role of scaling in genetic algorithm recombination and offers a different balance between exploration and exploitation. Among the methods, Random recombination appears most suitable for maximizing performance under specific parameter configurations, but the selection of an appropriate method ultimately depends

on the characteristics of the specific optimization problem, desired performance levels, and acceptable trade-offs between efficiency and result stability.

Table 4.5: The scores of the GA algorithms for given parameters, using a single-objective function.

Recombination	$P(C)$	$P(M)$	F	Improvement [%]	σ
Random	0.95	0.05	-	51.96	0.21
Random	0.95	0.01	-	34.64	0.23
Random	0.9	0.05	-	39.11	0.24
Random	0.9	0.01	-	34.31	0.21
Average	0.95	0.05	-	45.91	0.21
Average	0.95	0.01	-	35.06	0.22
Average	0.9	0.05	-	50.35	0.2
Average	0.9	0.01	-	35.69	0.19
Differential	0.95	-	0.05	39.33	0.25
Differential	0.95	-	0.15	38.53	0.24
Differential	0.9	-	0.05	37.28	0.22
Differential	0.9	-	0.15	37.4	0.23

Table 4.6 summarizes the performance of GA variations using a multi-objective optimization function. Performance is measured by improvement percentage and standard deviation (σ). For the Random recombination method, the highest improvement, 50.15%, occurs with $P(C) = 0.95$ and $P(M) = 0.05$. As $P(M)$ decreases to 0.01, the improvement declines significantly to 33.01–33.61%. The results suggest that maintaining moderate mutation levels is critical for this method. Standard deviations remain relatively stable, ranging from 0.22 to 0.27. The Average recombination method performs best with $P(C) = 0.90$ and $P(M) = 0.05$, achieving an improvement of 48.85%. Lower mutation probabilities reduce performance to approximately 33%, highlighting the importance of mutation in this approach. The method exhibits slightly lower variability ($\sigma = 0.20$ to 0.23) compared to Random recombination. The Differential recombination method delivers its best improvement, 37.84%, with $P(C) = 0.95$ and $F = 0.05$.

As F increases to 0.15, performance slightly decreases to 36.18–36.78%, and standard deviations rise marginally to 0.26–0.27. This suggests that smaller scaling factors may be more effective for this method. Overall, Random recombination achieves the highest peak performance, though Average recombination provides strong results with slightly lower variability. Differential recombination demonstrates reliable, though comparatively modest, improvements. The choice of method depends on the desired balance between performance and consistency.

Table 4.6: The scores of the GA algorithms for given parameters, using a multi-objective function.

Recombination	P(C)	P(M)	F	Improvement [%]	σ
Random	0.95	0.05	-	50.15	0.23
Random	0.95	0.01	-	33.61	0.24
Random	0.90	0.05	-	38.03	0.27
Random	0.90	0.01	-	33.01	0.22
Average	0.95	0.05	-	44.42	0.22
Average	0.95	0.01	-	33.19	0.23
Average	0.90	0.05	-	48.85	0.22
Average	0.90	0.01	-	33.78	0.20
Differential	0.95	-	0.05	37.84	0.27
Differential	0.95	-	0.15	36.78	0.26
Differential	0.90	-	0.05	35.91	0.25
Differential	0.90	-	0.15	36.18	0.26

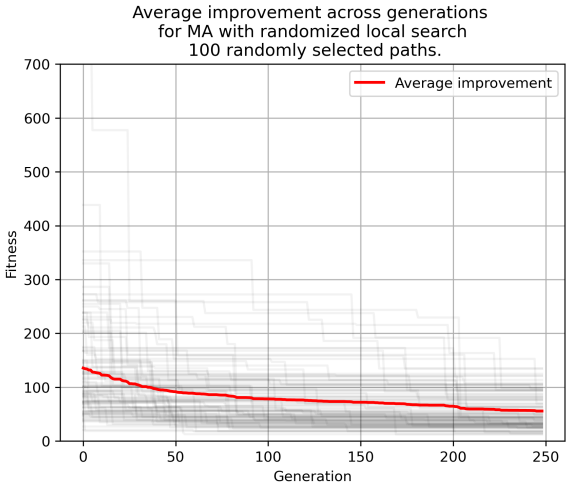
While the results of the multi-objective algorithm seem to be slightly lower on average, this is well within the expected statistical fluctuation of scores in a stochastic method such as GA, especially since it is applied on the random paths. The bigger issue with the multi-objective path optimization is the necessity for a much more time-demanding fitness function, with the fitness function being by far the most computationally complex part of the GA algorithm. Considering the models obtained for individual joints are not significantly smaller, this means that the training process for such multi-objective

algorithm is significantly more complex. This is then reflected in the execution times. Comparing the single-objective with a multi-objective GA with random recombination, the single-objective executes the search in 9.35 seconds ($N = 100, \sigma = 0.14$), while the multi-objective takes 26.82 s ($N = 100, \sigma = 0.22$). Both algorithms were run for 10 generations with 100 candidate solutions in population for the timing test in question.

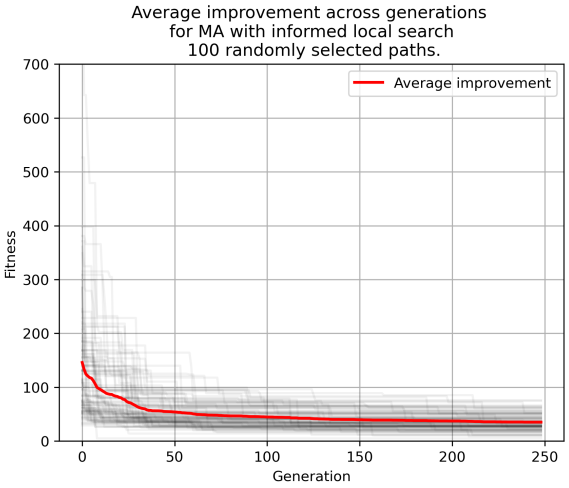
4.5.2 Performance of MA

MA is constructed using the most successful algorithm from the previous section—GA with the random recombination method configured with $P(C) = 95\%$ and $P(M) = 5\%$. As mentioned before, two variants of MA are tested to evaluate their effectiveness. The first variant is the random search of the surrounding area around the best candidate solution identified in each generation. In this approach, 1,000 solutions are evaluated within a defined neighborhood, specifically within 5 differentiation steps in both the positive and negative directions from the current best solution, ensuring comprehensive local exploration. The second variant is the so-called informed search, which focuses on optimizing only the parameters related to the first two joints of the robot. This is implemented by systematically reducing the values of these parameters by a factor of 0.9 and retesting the resulting configuration—a process that is iterated 20 times to refine the solution.

The MA employing randomized search achieves an improvement of 52.59%, with a standard deviation of 0.21, indicating both a significant enhancement over the initial solution and stable performance across trials. The informed search MA achieves a higher improvement of 59.79%, along with a slightly lower standard deviation of 0.18, suggesting greater precision and consistency in results. Figure 4.19 illustrates the improvements for both of these algorithms, showing not only their respective performance levels but also that the randomized local search variant tends to converge faster during the optimization process, highlighting its efficiency in finding improved solutions within fewer generations.



(a) Path improvement for MA with randomized local search



(b) Path improvement for MA with informed local search

Figure 4.19: Comparison of path improvements for two applied local search strategies

Based on both the presented improvement data and the overall scores, it is clear that the informed search is clearly superior. The randomized search performance is similar to the GA. While it might be improved by increasing the number of searched points, this would significantly adversly affect the time, which is already significantly higher than the MA with informed search. Randomized search MA, with the same settings used for timing GA, takes 20.12 seconds ($N = 100, \sigma = 0.12$), while informed MA takes 9.56 seconds ($N = 100, \sigma = 0.34$)

4.5.3 Energy-use comparison of robot paths generated with MA

The final presented result, given in Table 4.7, compares the energy of the optimized path with the linear path between the two points. The linear path was selected due to it being the default movement used to move the robot between two points in ABB RobotStudio. The measurements are performed with the code given in Appendix G, and the previously described measurement code, on the real ABB IRB 120 in a laboratory environment. The measurements were performed on five randomly selected paths. The results show that the optimization performance introduces a significant energy savings

compared to the default linear path.

Table 4.7: Comparison of few randomly selected paths, between the energy use of a linear point-to-point movement and the optimized path. *Gene* – the path parameters obtained via optimization, q^S – starting positions in joint space, q^E – ending positions in joint space, E_L – energy of the linear path, E_O – energy use of the optimized path, Δ – improvement with optimization.

Gene						q^S	q^E	E_L	$E_\mathrm{optimized}$	Δ [%]
0.00	0.62	−0.12	−0.12	−1.18	0.00	-14.32	120.32	75.21	24.52	67.39
−0.00	−0.10	−0.08	0.08	0.00	−0.14	-22.92	131.78			
9.45	1.60	5.85	1.25	7.85	−4.00	5.73	123.19			
3.30	7.00	−4.70	−9.20	−3.85	−5.05	40.11	140.37			
8.15	1.30	7.60	3.65	−7.50	−7.65	48.70	114.59			
1.00	6.75	4.30	7.45	−5.55	6.40	-54.43	166.16			
1.12	1.05	0.42	0.04	−0.74	−0.08	28.65	166.16	37.57	19.44	48.25
−0.16	0.65	−0.12	−0.06	−0.02	0.13	-28.65	117.46			
−1.25	8.75	5.20	1.40	2.35	5.25	37.24	117.46			
−9.40	0.25	−8.30	0.05	−2.95	3.20	20.05	163.29			
3.05	−5.80	1.35	−4.35	−6.25	3.60	51.57	166.16			
8.25	−7.40	0.00	−8.95	4.90	−1.85	37.24	114.59			
−0.24	−0.00	0.98	0.00	−1.20	0.43	22.91	114.59	92.20	25.16	72.71
0.03	−0.00	−0.85	1.15	−1.02	0.38	-40.11	120.32			
2.20	3.20	6.60	2.00	−6.45	5.95	22.92	137.51			
8.45	−2.35	−7.30	3.90	−1.05	−1.35	-20.05	126.05			
4.30	8.05	2.95	1.05	−3.55	−5.20	-25.78	128.92			
−9.70	−2.15	−1.25	8.25	−1.70	7.45	-20.05	114.59			
−0.01	0.94	0.00	0.00	−0.08	0.15	-42.97	143.24	38.76	18.00	53.56
0.01	0.00	0.00	0.13	0.18	0.00	28.65	146.10			
6.70	−8.45	−1.30	0.20	−4.10	−8.00	31.51	154.69			
4.50	0.35	−8.25	7.00	−8.50	8.65	51.57	117.46			
1.85	4.60	−1.90	4.45	−3.55	7.45	-40.11	123.19			
6.15	5.95	−6.50	−7.20	−9.70	3.90	22.92	143.24			
−0.00	−0.02	0.00	−0.97	0.10	−0.19	-40.11	143.24	38.39	29.26	23.78
0.00	1.00	−0.01	0.08	0.00	0.00	-22.92	154.69			
−7.95	5.90	4.70	−9.25	−7.45	−7.05	51.57	140.37			
8.80	−8.00	−1.35	7.15	0.25	1.45	8.59	137.51			
3.10	−4.40	−0.15	5.60	4.65	−3.55	25.78	134.65			
−3.15	−8.65	−9.05	6.00	−1.55	−1.70	20.06	131.78			

4.6 Summary

This chapter provided and compared the results of different parts of the methodology. First, it realized the scientific contribution of the energy consumption model by defining it as a data-driven model developed using the MLP method on synthetic data, as that model showed the best performance on all of the tested outputs. While synthetic data did not perform better than real-world data, it was selected due to simpler use and application in real-world applications. The model, as defined, has been used as a fitness function in an MA algorithm, which used the feature importances determined at the start of the chapter to perform a localized search and improve the results. This algorithm was compared to different algorithms present in past research. The results show that the proposed algorithm has significant improvement compared to classic GA, as well as comparing it to the classic movement the IRMs utilize between two points.

CHAPTER 5

Conclusion

The presented doctoral thesis described the methodology necessary to obtain the dataset of IRM energy use, from a real-world environment and simulation, and generate synthetic data from the real dataset. The datasets were extensively compared and used in the ML-modelling process using four algorithms, out of which the best performing one was picked to serve as a fitness function of an MA optimization process. The feature importance analysis performed using correlation metrics and ML-driven metrics suggests that the positions and speeds of the first two joints have the highest influence on the energy use for the IRM. The synthetic and real-world dataset show extremely high similarity on descriptive statistics, which is to be expected, due to the nature of how synthetic data is generated. On the other hand, simulation data shows a difference, specifically in those metrics that indicate its central tendency, while the metrics indicating ranges show higher similarities. This indicates the possibility of real-world data having similarly shaped distribution to the original data, but shifted, possibly due to outside influences that are not included within the simulation. This is most likely the reason why the ML models trained on simulation data perform poorly on the prediction of values in the real dataset. On the other hand, the synthetic data performs just as well as the real data, indicating that there is no particular need for full data collection, and the models can be based on real-world data augmented with synthetic models. Comparing the data-driven model to a pure numerical model obtained with LE algorithm, the data driven model follows the real data much more closely, indicating its superiority. This, best performing model – namely the MLP model trained on synthetic data, is used as a fitness function, allowing for optimization of paths interpolated with a fifth-order polynomial. MA based on the GA with random recombination was used for

optimization, and it has shown the ability to improve randomly placed and linear paths. The results suggest that data-driven models do have a higher performance on real-world data compared to classical models. In addition to that, the synthetic data-based models show the highest performing scores, on par with the models trained on real-world data, indicating that the generation of data in this case is possible. This indicates the possibility of simpler data collection on robots which are targets for the optimization. Finally, the MA has shown that the targeted search around the best found solution in a generation can significantly improve the performance of the optimization algorithms.

There are certain limitations to the presented work that need to be noted. The presented methodology is limited to six-axis IRMs, and it would require significant adjustments to tune different robot configurations – especially parallel configurations. Additionally, the entirety of the research is based on the ABB IRB 120 IRM. While research exists indicating the possibility of a limited transfer to different robots is possible, this point needs to be researched additionally. Future work will serve to use the framework presented in this doctoral thesis with the goal of expanding the capabilities to more general IRM data, developing datasets applicable to a wider set of IRMs.

To finalize, a summary of scientific contributions is provided. This thesis defined the process by which the data can be collected to develop data-driven methods for energy modeling of six DOF IRMs. The proposed process, based on data synthetization, serves to develop a usable dataset on a fraction of data normally used, while keeping the same characteristics as the full dataset collected on the real-world IRM. This enables faster collection of data which can be used to develop data-driven models that are more precise in energy consumption prediction than conventional analytical models or models developed on simulation data. These models can then be applied as the target function of an optimization process, to lower the energy consumption. This is the final scientific contribution of this doctoral thesis, with the MA combining GA with local search based adjustment of features with high influence. The proposed algorithm shows the possibility of improving the performance up to 72% compared to the linear point-to-point path.

Bibliography

[1] A Ilemobayo, J. et al. “Hyperparameter Tuning in Machine Learning: A Comprehensive Review”. In: *Journal of Engineering Research and Reports* 26.6 (2024), pp. 388–395.

[2] Afzal, S. et al. “Building energy consumption prediction using multilayer perceptron neural network-assisted models; comparison of different optimization algorithms”. In: *Energy* 282 (2023), p. 128446.

[3] Ahsan, M. M. et al. “Effect of data scaling methods on machine learning algorithms and model performance”. In: *Technologies* 9.3 (2021), p. 52.

[4] Alhijawi, B. et al. “Genetic algorithms: Theory, genetic operators, solutions, and applications”. In: *Evolutionary Intelligence* 17.3 (2024), pp. 1245–1256.

[5] Amiri, N. et al. “Optimization and control of an energy-efficient vibration-driven robot”. In: *Journal of Vibration and Control* 30.9-10 (2024), pp. 2184–2199.

[6] Ardema, M. D. *Newton-Euler Dynamics*. Springer Science & Business Media, 2004.

[7] Barenji, A. V. et al. “A digital twin-driven approach towards smart manufacturing: reduced energy consumption for a robotic cellular”. In: *International Journal of Computer Integrated Manufacturing* 34.7-8 (2020), pp. 1–16.

[8] Baressi Šegota, S. et al. “Path planning optimization of six-degree-of-freedom robotic manipulators using evolutionary algorithms”. In: *International Journal of Advanced Robotic Systems* 17.2 (2020), p. 1729881420908076.

[9] Baressi Šegota, S. et al. “Determining normalized friction torque of an industrial robotic manipulator using the symbolic regression method”. In: *XVI International Conference for Young Researchers" Technical Sciences. Industrial Management 2023"*. 2023, pp. 5–8.

[10] Bernardes, E. et al. “Quaternion to Euler angles conversion: A direct, general and computationally efficient method”. In: *PLOS One* 17.11 (2022), e0276302.

[11] Bilancia, P. et al. “An Overview of Industrial Robots Control and Programming Approaches”. In: *Applied Sciences* 13.4 (2023), p. 2582.

[12] Bruin, S. de et al. “Autoencoders as Tools for Program Synthesis”. In: *arXiv preprint arXiv:2108.07129* (2021).

[13] Chang, X. et al. “A reinforcement learning enhanced memetic algorithm for multi-objective flexible job shop scheduling toward Industry 5.0”. In: *International Journal of Production Research* 63.1 (2025), pp. 119–147.

[14] Chen, T. et al. “Xgboost: A scalable tree boosting system”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 785–794.

[15] Chen, X. et al. “Support Vector Machines Principles and Actually Example”. In: *Procedia Computer Science* 243 (2024), pp. 2–11.

[16] Choi, H. et al. “On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward”. In: *Proceedings of the National Academy of Sciences* 118.1 (2021), e1907856118.

[17] Chutima, P. “A comprehensive review of robotic assembly line balancing problem”. In: *Journal of Intelligent Manufacturing* 33.1 (2022), pp. 1–34.

[18] Featherstone, R. et al. “Robot dynamics: equations and algorithms”. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 1. IEEE. 2000, pp. 826–834.

[19] Gadaleta, M. et al. “Extensive experimental investigation for the optimization of the energy consumption of a high payload industrial robot with open research dataset”. In: *Robotics and Computer-Integrated Manufacturing* 68 (2021), p. 102046.

[20] Gao, Y. et al. “Energy consumption prediction for 3-RRR PPM through combining LSTM neural network with whale optimization algorithm”. In: *Mathematical Problems in Engineering* 2020 (2020), pp. 1–17.

[21] Garcia-Rubio, C. et al. “Synthetic Generation of Electrical Consumption Traces in Smart Homes”. In: *International Conference on Ubiquitous Computing and Ambient Intelligence*. Springer. 2022, pp. 681–692.

[22] Garriz, C. et al. “Trajectory optimization in terms of energy and performance of an industrial robot in the manufacturing industry”. In: *Sensors* 22.19 (2022), p. 7538.

[23] Guan, Y. et al. “On robotic trajectory planning using polynomial interpolations”. In: *2005 IEEE International Conference on Robotics and Biomimetics-ROBIO*. IEEE. 2005, pp. 111–116.

[24] Gunasekara, N. et al. “Gradient boosted trees for evolving data streams”. In: *Machine Learning* 113.5 (2024), pp. 3325–3352.

[25] Haghi, R. et al. “Wind Turbine damage equivalent load assessment using Gaussian process regression combining measurement and synthetic data”. In: *Energies* 17.2 (2024), p. 346.

[26] Hastie, T. et al. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer, 2009.

[27] He, Y.-X. et al. “Interpreting deep forest through feature contribution and mdi feature importance”. In: *ACM Transactions on Knowledge Discovery from Data* (2024).

[28] Hernandez, M. et al. “Synthetic data generation for tabular health records: A systematic review”. In: *Neurocomputing* 493 (2022), pp. 28–45.

[29] Hirayama, T. et al. “Speed effects in touching behaviours: impact on perceived relationships in robot-robot interactions”. In: *Advanced Robotics* 38.7 (2024), pp. 492–509.

[30] Hodson, T. O. “Root mean square error (RMSE) or mean absolute error (MAE): When to use them or not”. In: *Geoscientific Model Development Discussions* 2022 (2022), pp. 1–10.

[31] Holcomb, Z. *Fundamentals of descriptive statistics*. Routledge, 2016.

[32] Iversen, O. J. M. *Friction in Robotic Manipulators: Modeling, Compensation and Simulation*. Norwegian University of Science and Technology, 2002.

[33] James, G. et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[34] Jaramillo-Morales, M. F. et al. “Energy estimation for differential drive mobile robots on straight and rotational trajectories”. In: *International Journal of Advanced Robotic Systems* 17.2 (2020), p. 1729881420909654.

[35] Jiang, P. et al. “Energy consumption prediction and optimization of industrial robots based on LSTM”. In: *Journal of Manufacturing Systems* 70 (2023), pp. 137–148.

[36] Kleeberger, K. et al. “A survey on learning-based robotic grasping”. In: *Current Robotics Reports* 1 (2020), pp. 239–249.

[37] Koza, J. R. et al. “Hierarchical genetic algorithms operating on populations of computer programs.” In: *IJCAI*. Vol. 89. 1989, pp. 768–774.

[38] Lin, H.-I. et al. “BN-LSTM-based energy consumption modeling approach for an industrial robot manipulator”. In: *Robotics and Computer-Integrated Manufacturing* 85 (2024), p. 102629.

[39] Lipkin, H. “A note on Denavit-Hartenberg notation in robotics”. In: *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 47446. 2005, pp. 921–926.

[40] Ltd., A. *ABB IRB 120 3/0.6 Product Manual*. ABB Ltd.: Zurich, Switzerland, 2022.

[41] Ltd., A. *ABB IRB 120 3/0.6 Product Manual*. ABB Ltd.: Zurich, Switzerland, 2022.

[42] Ltd., A. *ABB IRB 120 Datasheet*. ABB Ltd.: Zurich, Switzerland, 2022.

[43] Ltd., A. *Operating manual RobotStudio*. ABB Ltd.: Zurich, Switzerland, 2022.

[44] Ltd., A. *Technical reference manual - RAPID Instructions, Functions and Data types*. ABB Ltd.: Zurich, Switzerland, 2022.

[45] Lu, C. et al. “Human-Robot Collaborative Scheduling in Energy-efficient Welding Shop”. In: *IEEE Transactions on Industrial Informatics* (2023).

[46] Luneckas, M. et al. “Hexapod robot gait switching for energy consumption and cost of transport management using heuristic algorithms”. In: *Applied sciences* 11.3 (2021), p. 1339.

[47] Luo, C. et al. “Knowledge-driven two-stage memetic algorithm for energy-efficient flexible job shop scheduling with machine breakdowns”. In: *Expert Systems with Applications* 235 (2024), p. 121149.

[48] Mahayana, D. “Data-Driven LightGBM Controller for Robotic Manipulator”. In: *IEEE Access* 12 (2024), pp. 40883–40893.

[49] Mannino, M. et al. “Is this real? Generating synthetic data that looks real”. In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 2019, pp. 549–561.

[50] McCarthy, J. M. *Introduction to theoretical kinematics*. MIT press, 1990.

[51] Miletic, M. et al. “Challenges of using synthetic data generation methods for tabular microdata”. In: *Applied Sciences* 14.14 (2024), p. 5975.

[52] Nagelkerke, N. J. et al. “A note on a general definition of the coefficient of determination”. In: *Biometrika* 78.3 (1991), pp. 691–692.

[53] Nguyen, A. et al. “An analysis of state-of-the-art activation functions for supervised deep neural network”. In: *2021 International conference on system science and engineering (ICSSE)*. IEEE. 2021, pp. 215–220.

[54] Nonoyama, K. et al. “Energy-efficient robot configuration and motion planning using genetic algorithm and particle swarm optimization”. In: *Energies* 15.6 (2022), p. 2074.

[55] O’Reilly, O. M. *Intermediate Dynamics for Engineers: Newton-Euler and Lagrangian Mechanics*. Cambridge University Press, 2020.

[56] Osiński, B. et al. “Simulation-based reinforcement learning for real-world autonomous driving”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 6411–6418.

[57] Oztemel, E. et al. “Literature review of Industry 4.0 and related technologies”. In: *Journal of Intelligent Manufacturing* 31 (2020), pp. 127–182.

[58] Parise, O. et al. “CTGAN-driven synthetic data generation: A multidisciplinary, expert-guided approach (TIMA)”. In: *Computer Methods and Programs in Biomedicine* 259 (2025), p. 108523.

[59] Patki, N. et al. “The synthetic data vault”. In: *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2016, pp. 399–410.

[60] Pedregosa, F. et al. “Scikit-learn: Machine learning in Python”. In: *The Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[61] Reyes-Davila, E. et al. “Differential evolution: a survey on their operators and variants”. In: *Archives of Computational Methods in Engineering* 32.1 (2025), pp. 83–112.

[62] Savran, E. et al. “Synthetic data generation using Copula model and driving behavior analysis”. In: *Ain Shams Engineering Journal* 15.12 (2024), p. 103060.

[63] Schoettler, G. et al. “Meta-reinforcement learning for robotic industrial insertion tasks”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 9728–9735.

[64] Schofield, S. et al. “An improved semi-synthetic approach for creating visual-inertial odometry datasets”. In: *Graphical Models* 126 (2023), p. 101172.

[65] Sevšek, L. et al. “Determining the influence and correlation for parameters of flexible forming using the random forest method”. In: *Applied Soft Computing* (2023), p. 110497.

[66] Shrivastava, A. et al. “Failure control and energy optimization of multi-axes space manipulator through genetic algorithm approach”. In: *Journal of the Brazilian Society of Mechanical Sciences and Engineering* 43 (2021), pp. 1–17.

[67] Sihag, N. et al. “A systematic literature review on machine tool energy consumption”. In: *Journal of Cleaner Production* 275 (2020), p. 123125.

[68] Srinivas, G. L. et al. “Optimization approaches of industrial serial manipulators to improve energy efficiency: A review”. In: *IOP Conference Series: Materials Science and Engineering*. Vol. 912. IOP Publishing. 2020, p. 032058.

[69] Swanborn, S. et al. “Energy efficiency in robotics software: A systematic literature review”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 2020, pp. 144–151.

[70] Tamizi, M. G. et al. “A review of recent trend in motion planning of industrial robots”. In: *International Journal of Intelligent Robotics and Applications* 7.2 (2023), pp. 253–274.

[71] Tang, Z. et al. “Learning-Based Control for Soft Robot–Environment Interaction with Force/Position Tracking Capability”. In: *Soft Robotics* (2024).

[72] VanDerHorn, E. et al. “Digital Twin: Generalization, characterization and implementation”. In: *Decision Support Systems* 145 (2021), p. 113524.

[73] Vysockỳ, A. et al. “Reduction in robotic arm energy consumption by particle swarm optimization”. In: *Applied Sciences* 10.22 (2020), p. 8241.

[74] Wang, X. et al. “An Investigation of Real-Time Robotic Polishing Motion Planning Using a Dynamical System”. In: *Machines* 12.4 (2024), p. 278.

[75] Yoshikawa, T. *Foundations of robotics: analysis and control*. MIT press, 1990.

[76] Zhang, J. et al. “Applications of artificial neural networks in microorganism image analysis: a comprehensive review from conventional multilayer perceptron to popular convolutional neural network and potential visual transformer”. In: *Artificial Intelligence Review* 56.2 (2023), pp. 1013–1070.

[77] Zhang, M. et al. “A data-driven method for optimizing the energy consumption of industrial robots”. In: *Journal of Cleaner Production* 285 (2021), p. 124862.

[78] Zhao, W. et al. “Sim-to-real transfer in deep reinforcement learning for robotics: a survey”. In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2020, pp. 737–744.

List of Figures

List of Tables

List of Abbreviations

AI – Artificial Intelligence
ANN – Artificial neural networks
CDF – Cumulative distribution function
CSV – Comma-separated values
CTGAN – Conditional tabular generative adversarial network
CV – Cross-validation
DE – Differential evolution
DOF – Degree–of–freedom
ELBO – Evidence lower bound
EP – Elasto–Plastic
GA – Genetic algorithm
GBT – Gradient boosted trees
IFR – International Federation of Robotics
IRM – Industrial robotic manipulator
JS – Jensen–Shannon divergence
KL – Kullback–Leibler divergence
LSTM – Long–short term memory
MA – Memetic algorithm
MAPE – Mean absolute percentage error
MDI – Mean decrease in impurity
MLP – Multilayer perceptron
MSE – Mean squared error
OOB – Out–of–bag
PAR – Passive agressive regressor
PDF – Probability density function
PID – Proportional – Integral – Derivative
PPM – planar parallel manipulator
PSO – Particle swarm optimization
RBF – Radial basis function
RF – Random Forests
RMSE – Root mean squared error
RWS – Roulette wheel selection
SGD – Stochastic gradient descent
SVR – Support vector regressor
TCP – Tool center point
TVAE – Tabular Variational Autoencoder
VNS – Variable neighbourhood search
XGB – XGBoost

LIST OF TABLES

List of Symbols

Symbol	Description
E	Total energy consumed by the robotic joint.
$M(q, \dot{q}, \ddot{q})$	Torque as a function of joint angle q, angular velocity $\dot{q}$, and angular acceleration $\ddot{q}$.
q	Joint angle.
$\dot{q}$	Angular velocity of the joint.
$\ddot{q}$	Angular acceleration of the joint.
Δt	Time interval between consecutive data points.
N	Total number of discrete data points.
m_k	Mass of link k.
v^k	Linear velocity of the center of mass of link k relative to the base coordinate system L_0.
ω_k	Angular velocity of the center of mass of link k relative to L_0.
r_i'	Position vector of particle i relative to the center of mass.
p_i'	Linear momentum of particle i.
L_i	Angular momentum of particle i.
L	Total angular momentum of the body.
J	Moment of inertia of the body.
I_{xx}, I_{yy}, I_{zz}	Principal moments of inertia about the x, y, and z axes.
$I_{xy}, I_{xz}, I_{yx}, I_{yz}, I_{zx}, I_{zy}$	Products of inertia.
D	Inertia tensor matrix.
θ_k	Joint angle for link k (rotation about z^{k-1} axis).
d_k	Offset distance along z^{k-1} axis for link k.
a_k	Link length (distance along x^k axis for link k).
α_k	Link twist angle (rotation about x^k axis for link k).
T_{base}^{end}	Homogeneous transformation matrix from base to end-effector.
$R(q)$	3×3 orientation matrix of the end-effector.
$p(q)$	3×1 position vector of the end-effector.
v_1^T	Perspective vector (typically $[0\ 0\ 0]$), transposed.
σ	Scaling coefficient (typically 1).
r^1	Normal vector of the end-effector.
r^2	Sliding (or moving) vector of the end-effector.
r^3	Approaching vector of the end-effector.
T_{k-1}^k	Transformation matrix from frame $k-1$ to frame k.
L	Lagrangian of the system, defined as kinetic energy minus potential energy.
T	Kinetic energy of the robotic manipulator.
U	Potential energy of the robotic manipulator.
$D_{kj}(q)$	Inertia tensor element between joints k and j as a function of joint positions q.
g^k	Gravitational acceleration component along axis k.

Continued on next page

Table 5.1 – continued from previous page

Symbol	Description
m_j	Mass of link j.
$c_j^k(q)$	Position of the center of mass of link j along axis k, as a function of q.
$C_{kj}^i(q)$	Velocity coupling matrix element (Coriolis/centrifugal force term) for joint i.
$h_i(q)$	Gravity-related term for joint i.
$b_i(q)$	Friction-related term for joint i.
τ_i	Actuator torque at joint i.
T_0^0	Initial base transformation matrix, equal to the identity matrix.
$D(q)$	Manipulator inertia tensor up to the current joint.
Δc^i	Center of mass of link i relative to its own coordinate system.
D_i'	Inertia tensor of link i in its local coordinate system.
$z^{i-1}(q)$	Joint transformation vector for joint $i-1$.
$R_0^{i-1}(q)$	Rotation matrix from base to joint $i-1$.
i^3	Unit vector along the z-axis in the base frame.
$T_0^i(q)$	Composite homogeneous transformation matrix from base to joint i.
$c^i(q)$	Center of mass of link i in global coordinates.
$D_i(q)$	Inertia tensor of link i relative to the base coordinate system.
$J^k(q)$	Jacobian matrix for link k.
$A^k(q)$	Linear velocity component of Jacobian for link k.
$B^k(q)$	Angular velocity component of Jacobian for link k.
ξ	Joint type indicator ($\xi = 1$ for rotary, $\xi = 0$ for linear).
$A_{ki}^j(q)$	Element of matrix $A^j(q)$ corresponding to axis k and joint i.
F_{EP}	Friction force as per Elasto-Plastic friction model.
μ_s	Static friction coefficient.
μ_k	Kinetic friction coefficient.
v	Relative velocity between contacting surfaces.
$f(x)$	General form of the polynomial used for interpolation.
a_i	Coefficients of the polynomial, adjustable parameters for path shaping.
m	Number of coefficients in the polynomial (polynomial order).
n	Degree of the highest-order term in the polynomial.
t	Time at which the movement is evaluated (independent variable).
y	True output value associated with a data point.
$X_{i,:}$	Input data vector (row i of dataset X).
X	Full dataset of input vectors.
$\hat{y}$	Predicted output value generated by the model.
M	Machine learning model.
w	Trainable parameters of the model.
ϵ	Prediction error (difference between $\hat{y}$ and y).
$\mathcal{L}$	Model loss (another term for error ϵ).
H	Set of hyperparameters defining the model's structure and training behavior.
h_i	List of possible values for hyperparameter i.
N	Total number of hyperparameter combinations (number of models trained).
X^V	Validation input dataset (subset of X).

Continued on next page

Table 5.1 – continued from previous page

Symbol	Description
Y^V	Validation output dataset (true outputs corresponding to X^V).
X^{TR}	Training input dataset.
Y^{TR}	Training output dataset.
$Y^{\hat{V}}$	Predicted outputs for the validation set.
R^2	Coefficient of determination (statistical measure of model fit).
SS_{res}	Residual sum of squares (unexplained variance).
SS_{tot}	Total sum of squares (total variance in data).
$\overline{Y}$	Mean of the target output values.
MAE	Mean absolute error.
n	Number of data points in the dataset.
k	Number of cross-validation splits (folds).
$X_{:,j}$	Feature column j of the dataset X.
$X'_{:,j}$	Standardized version of feature column j.
μ_j	Mean of feature column j.
σ_j	Standard deviation of feature column j.
$a_j^{(l)}$	Output of neuron j in layer l of an MLP.
$w_{ij}^{(l)}$	Weight connecting neuron i in layer $l-1$ to neuron j in layer l.
$b_j^{(l)}$	Bias term for neuron j in layer l.
ϕ	Activation function used in neural networks.
$\mathcal{L}$	Loss function used for training a model.
$z_j^{(l)}$	Weighted input to neuron j in layer l.
$\delta_j^{(l)}$	Error term for neuron j in layer l (used in backpropagation).
η	Learning rate, controlling the step size in weight updates.
η_t	Learning rate at iteration t.
γ	Learning rate reduction factor (for adaptive/invscaling schedules), or exponent in invscaling.
$\mathcal{L}_{\text{reg}}$	Regularized loss function (includes L2 penalty).
$f(X_{i,:})$	Output of the SVR model for input $X_{i,:}$.
ϵ	Tolerance margin in SVR within which no penalty is applied.
ξ_i, ξ_i^*	Slack variables measuring deviation from the ϵ margin.
C	Regularization parameter in SVR, balancing margin violations and model complexity.
$\phi(X_{i,:})$	Non-linear transformation of $X_{i,:}$ to higher-dimensional space.
$K(X_{i,j})$	Kernel function applied to inputs $X_{i,:}$ and $X_{:,j}$.
α_i, α_i^*	Lagrange multipliers associated with slack variables in SVR.
ε	Width of epsilon-insensitive tube in SVR (alternative notation for ϵ).
w	Current weight vector of the model.
w_{prev}	Weight vector from the previous iteration.
τ	Step size for weight update in Passive Aggressive Regressor.
$\|w - w_{\text{prev}}\|^2$	Squared Euclidean distance between current and previous weight vectors.
$\|X_{i,:}\|^2$	Squared norm of the input feature vector $X_{i,:}$.
$f_t(X_{i,:})$	Prediction of the t-th decision tree in Gradient Boosted Trees.
$\hat{y}^{(t)}$	Model prediction at iteration t.
$\hat{y}^{(t-1)}$	Model prediction at iteration $t-1$.

Continued on next page

Table 5.1 – continued from previous page

Symbol	Description
η^{f_t}	Learning rate for the t-th tree.
$\mathcal{L}(y, \hat{y})$	Loss function comparing true value y and prediction $\hat{y}$.
$\mathcal{L}^{(t)}$	Objective function at iteration t in Gradient Boosted Trees.
$\Omega(f_t)$	Regularization term for the tree f_t.
T	Number of leaves in a decision tree.
w_j	Weight assigned to the j-th leaf in a decision tree.
γ	Regularization parameter penalizing the number of leaves in a tree.
λ	Regularization parameter penalizing the magnitude of leaf weights.
g_i	First derivative (gradient) of the loss with respect to $\hat{y}_i$.
h_i	Second derivative (Hessian) of the loss with respect to $\hat{y}_i$.
I_j	Set of data points assigned to leaf j.
$\mathcal{R}$	Loss reduction achieved by splitting a node in the tree.
I_L, I_R	Sets of data points in the left and right child nodes of a split.
I	Set of data points in the parent node before splitting.
a_i^j	Coefficient of order i for the path of joint j.
t	Generation index in evolutionary algorithms.
f_i	Fitness value of the i-th individual (to be minimized).
f_i'	Adjusted fitness value of the i-th individual (inverse of f_i).
F_{TOTAL}	Total sum of adjusted fitness values.
P_i	Probability of selection for the i-th individual in roulette wheel selection.
C_i	Cumulative probability for the i-th individual.
r	Uniformly generated random value in $[0, 1]$ used for selection.
$\mathcal{F}(\S)$	Fitness function for single-objective optimization.
$\mathcal{F}(\S_\infty, \S_\in, \cdots, \S_\setminus)$	Fitness function for multi-objective optimization.
$\mathcal{P}$	Population of candidate solutions.
N	Population size (number of individuals).
t	Generation index.
$\mathbf{p}_1$, $\mathbf{p}_2$	Parent candidate solutions (chromosomes).
$\mathbf{o}_1$	Offspring solution resulting from crossover.
g	Gene in a chromosome.
g'	Mutated gene value.
δ	Small random perturbation applied during mutation.
$\mathcal{P}^{(t+1)}$	Population at generation $t + 1$.
$\mathcal{P}_0$	Initial population.
$\mathbf{p}_i^0$	i-th individual in the initial population.
$\mathcal{F}(\mathbf{p}_i^0)$	Fitness function evaluated on individual $\mathbf{p}_i^0$.
R	Reproduction operator.
C	Crossover operator.
$P(C)$	Probability of performing crossover.
$P(M)$	Probability of performing mutation.
$\mathbf{p}_A$, $\mathbf{p}_B$	Two selected parent solutions.
A	Matrix representation of parent $\mathbf{p}_A$.

Continued on next page

Table 5.1 – continued from previous page

Symbol	Description
B	Matrix representation of parent $\mathbf{p}_B$.
a_i^j	Coefficient from matrix A, for joint j, coefficient i.
b_i^j	Coefficient from matrix B, for joint j, coefficient i.
Y	Resulting offspring matrix from recombination.
y_i^j	Element of offspring matrix Y, for joint j, coefficient i.
p_c	Probability of crossover in the genetic algorithm.
p_m	Probability of mutation in the genetic algorithm.
G	Maximum number of generations.
$P_{\text{offspring}}$	Offspring population generated in each generation.
$\mathcal{P}$	Population of candidate solutions.
N	Population size.
t	Generation index (iteration count in evolutionary algorithm).
$\mathbf{p}_i$	i-th individual (candidate solution) in the population.
$\mathbf{p}'_i$	Locally refined version of $\mathbf{p}_i$ after local search.
LocalSearch$(\cdot)$	Local search operator applied to a candidate solution.
$f(\mathbf{p})$	Fitness function evaluating the quality of a solution $\mathbf{p}$.
$\mathcal{N}(\mathbf{p}_i)$	Neighborhood of solution $\mathbf{p}_i$ (set of nearby solutions considered in local search).
Y'	Randomly perturbed solution derived from a base solution Y during local search.
$U(I)$	Uniform random selection of a value from set I.
I	Set of possible discretization steps: $I = [\pm 0.25, \pm 0.20, \pm 0.15, \pm 0.10, \pm 0.05, 0]$.
n	Number of data points in dataset.
ij	Dimensionality of solution (total number of coefficients in a solution).
$Y^d = [y_1, y_2, \cdots, y_n]$	Output variable vector for dataset d.
$X^d_{:,j}$	j-th feature column of dataset d.
$\mu_{X^d_{:,j}}$	Mean of feature $X^d_{:,j}$.
μ_{Y^d}	Mean of output variable vector Y^d.
$\rho^P_{X^d_{:,j},Y^d}$	Pearson's correlation coefficient between $X^d_{:,j}$ and Y^d.
$\rho^S_{X^d_{:,j},Y^d}$	Spearman's correlation coefficient between $X^d_{:,j}$ and Y^d.
$\rho^K_{X^d_{:,j},Y^d}$	Kendall's correlation coefficient between $X^d_{:,j}$ and Y^d.
$\varrho(x_i)$	Rank of element x_i in $X^d_{:,j}$.
$\varrho(y_i)$	Rank of element y_i in Y^d.
sgn$(\cdot)$	Sign function returning 1, -1, or 0.
$f_t(X_{i,:})$	Prediction of the t-th tree in a random forest for input $X_{i,:}$.
$\hat{y}$	Aggregated prediction of the random forest for input $X_{i,:}$.
T	Total number of trees in the random forest.
D_i	Bootstrap sample of data used to train tree i.
F	Set of all features.
F_{split}	Subset of features considered for a split.
ΔI_v	Reduction in impurity at node v.
I_v, I_L, I_R	Variances of the target variable in node v, and its children L and R.
$\lvert v\rvert, \lvert L\rvert, \lvert R\rvert$	Number of data points in node v, and its children L and R.

Continued on next page

Table 5.1 – continued from previous page

Symbol	Description
$I_{MDI}(X_{:,j})$	Mean decrease in impurity importance for feature $X_{:,j}$.
$\mathcal{N}_t$	Set of nodes in tree t.
$\mathbb{1}(\cdot)$	Indicator function, equal to 1 if condition is true, otherwise 0.
$I_{perm}(X_{:,j})$	Permutation importance of feature $X_{:,j}$.
$A_{orig}^{(t)}$	Accuracy of tree t on original data.
$A_{perm}^{(t)}$	Accuracy of tree t after permuting feature $X_{:,j}$.
$X = (X_{:,1}, X_{:,2}, \ldots, X_{:,d})$	Random vector.
$F_X(X_{i,1}, X_{i,2}, \ldots, X_{i,d})$	Joint cumulative distribution function (CDF).
$F_{X_{i,j}}(X_{i,j})$	Marginal CDF of variable $X_{i,j}$.
$C(u_1, u_2, \ldots, u_d)$	Copula function.
$u_i = F_{X_{i,j}}(X_{i,j})$	Uniform variable on $[0, 1]$.
Φ^{-1}	Inverse CDF of standard normal distribution.
Φ_Σ	Multivariate normal CDF with covariance matrix Σ.
Σ	Covariance matrix encoding dependency structure.
$\mathbf{R}$	Correlation matrix.
ρ_{ij}	Pearson correlation coefficient.
$\mathbf{Z} = (Z_1, Z_2, \ldots, Z_d)$	Sampled vector from multivariate normal distribution.
$U_i = \Phi(Z_i)$	Transformed uniform variable.
$X_{i,j} = F_{X_{i,j}}^{-1}(U_i)$	Synthetic data point.
G	Generator function in GAN.
D	Discriminator function in GAN.
$Z \sim p_Z$	Noise vector sampled from prior distribution.
$G(Z)$	Synthetic data generated by G.
$X \sim p_{\text{data}}$	Real data sample.
$\mathcal{L}$	WGAN-GP loss function.
$\hat{X}$	Interpolated data point between real and synthetic.
λ	Regularization parameter for gradient penalty.
$X_{i,j}^{(m)} = \frac{X_{i,j} - \mu_m}{\sigma_m}$	Mode-specific normalized value.
μ_m	Mean of mode m.
σ_m	Standard deviation of mode m.
$Z \sim \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2))$	Sampled latent variable.
$\boldsymbol{\mu}$	Mean vector of latent distribution.
$\boldsymbol{\sigma}$	Standard deviation vector of latent distribution.
$\mathcal{L}_{\text{ELBO}}$	Evidence lower bound loss function.
$q(Z\|X)$	Approximate posterior distribution.
$p(Z)$	Prior distribution over latent space.
$p(X\|Z)$	Likelihood of data given latent variable.
$D_{\text{KL}}(q(Z\|X)\|p(Z))$	Kullback-Leibler divergence.
$P(y_i = j\|Z) = \frac{\exp(l_{ij})}{\sum_{k=1}^{K}\exp(l_{ik})}$	Softmax probability for categorical variable.
$\mathbf{l}_i$	Predicted logits vector.
l_{ij}	j-th logit for i-th variable.
C_{pair}	Column Pair Score.

Continued on next page

Table 5.1 – continued from previous page

Symbol	Description
$P_{\text{real}}(X_{i,j}, X_{:,j})$	Joint distribution in real data.
$P_{\text{synthetic}}(X_{i,j}, X_{:,j})$	Joint distribution in synthetic data.
D_{JS}	Jensen-Shannon divergence.
N_{pairs}	Number of column pairs.
C_{shape}	Column Shape Score.
$P_{\text{real}}(X_{:,j})$	Marginal distribution in real data.
$P_{\text{synthetic}}(X_{:,j})$	Marginal distribution in synthetic data.
N_{columns}	Number of columns.
Δt	Time between two data points.
$\omega_i \forall i \in [1, \cdots, 6]$	Acceleration of each axis.
$\Delta x, \Delta y, \Delta z$	Change in linear position of TCP.
$\Delta\phi, \Delta\theta, \Delta\psi$	Change in angle of TCP.
$\Delta\dot{x}, \Delta\dot{y}, \Delta\dot{z}$	Linear components of TCP speed and acceleration.
$\Delta\dot{\phi}, \Delta\dot{\theta}, \Delta\dot{\psi}$	Angular speeds and accelerations.
t_1	Time of the first data point.
t_2	Time of the second data point.
$\Delta t = t_2 - t_1$	Measurement time difference.
$\tau_i \forall i \in [1, \cdots, 6]$	Individual joint torque.
X	Dataset.
$X_{:,j}$	Variable (column) vector from dataset.
$X_{:,j} = [x_{1,j}, X_{2,j}, \cdots X_{N,j}]^T$	Vector of N elements in variable $X_{:,j}$.
N	Number of elements in each feature.
$X = [X_{i,1}, X_{i,2}, \cdots X_{i,m}]$	Dataset with m variables.
m	Number of variables in dataset.
$\mu_{X^d_{:,j}}$	Mean value of variable $X^d_{:,j}$.
M_{X^d}	Median of dataset X^d.
m	Mode of variable (value with highest frequency).
$f_{X^d_{i,:}}$	Frequency of value $X^d_{i,:}$.
$V_{X^d_{:,j}}$	Variance of variable $X^d_{:,j}$.
$\sigma_{X^d_{:,j}}$	Standard deviation of variable $X^d_{:,j}$.
$\lozenge_{X^d_{:,j}}$	Range of variable $X^d_{:,j}$.
$\max(X^d_{:,j}), \min(X^d_{:,j})$	Maximum and minimum values of $X^d_{:,j}$.
$Sk_{X^d_{:,j}}$	Skewness of variable $X^d_{:,j}$.
$K_{X^d_{:,j}}$	Kurtosis of variable $X^d_{:,j}$.

Curriculum Vitae

Sandi Baressi Šegota mag. ing. comp. is a teaching assistant at the Department of Automation and Electronics, Faculty of Engineering – University of Rijeka and a member of the Chair of Electronics, robotics and automation and Laboratory for Automation and Robotics. After finishing Secondary Technical School Pula in 2013 he graduated from Faculty of Engineering University of Rijeka with a bachelors degree (cum laude) in computer science in 2017 and the master's degree (cum laude) in the same field in 2019. He enrolled as a candidate for doctoral studies at Faculty of Engineering Rijeka in the same year. He was employed at the University as a junior researcher – expert associate in 2019, as a collaborator on CEKOM SmartCity.4DII (KK.01.2.2.03.0004) and Scientific Center of Research Excellence in Data Science "DATACROSS" (KK.01.1.1.01.0009). He has also collaborated on multiple other projects. He has coauthored over 50 papers Web of Science Core Collection, 30% of which as the first author. He has also coauthored 22 papers at scientific conferences and 1 book chapter. He was awarded for the Exceptional contribution to the development and promotion of technical culture by Community of Technical Culture Pula in 2021. and the reward for the best paper in the field of Artificial Intelligence in Medical science at SICAAI conference in 2023. In addition to his work, he is a member of Astronomical Society "Istra" Pula (ADIP) since 2008, serving as a member of its directorate since 2021; Institute of Electrical and Electronic Engineers (IEEE) since 2021 and Global Meteor Network (GMN).

Selected publications

Scientific papers published in journals

1. Baressi Šegota S., et al. Dynamics Modeling of Industrial Robotic Manipulators: A Machine Learning Approach Based on Synthetic Data. Mathematics. 2022 Apr 4;10(7):1174.

2. Baressi Šegota S., et al.. Path planning optimization of six-degree-of-freedom robotic manipulators using evolutionary algorithms. International journal of advanced robotic systems. 2020 Mar 19;17(2): 1729881420908076.

3. Baressi Šegota S., et al. Utilization of multilayer perceptron for determining the inverse kinematics of an industrial robotic manipulator. International Journal of Advanced Robotic Systems. 2021 Aug 13;18(4):1729881420925283.

Scientific papers published in conferences

1. Car Z., Baressi Šegota S., et al. Determining Inverse Kinematics of a Serial Robotic Manipulator Through the Use of Genetic Programming Algorithm. In 8th International Congress of Serbian Society of Mechanics. 2021.

2. Baressi Šegota S., et al. Determining normalized friction torque of an industrial robotic manipulator using the symbolic regression method. Industry 4.0. 2023;8(1):21-4.

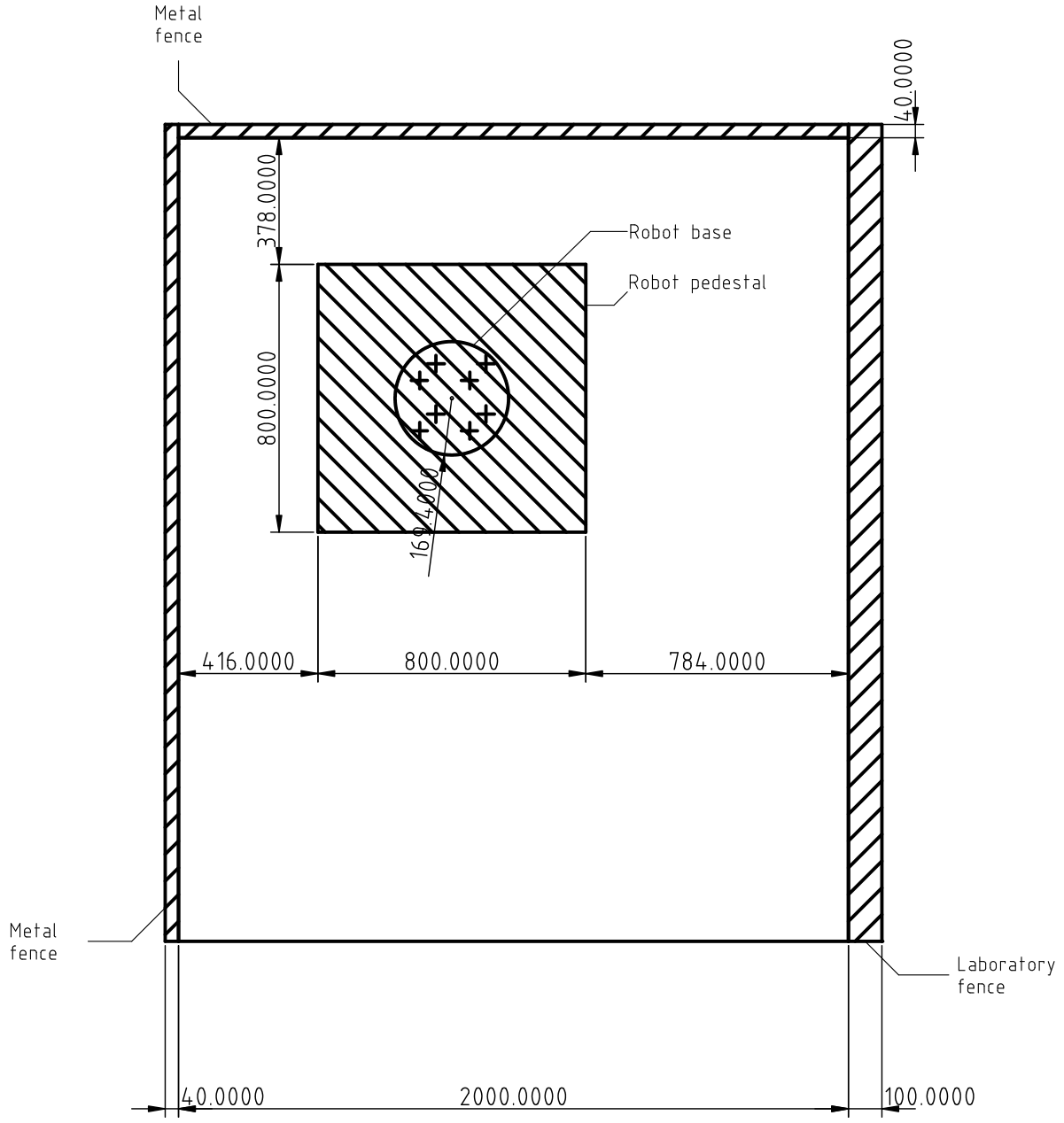
Acknowledgments

This research was partly supported by:

- CEEPUS network CIII-HR-0108,
- Erasmus+ project WICT under the grant 2021-1-HR01-KA220-HED-000031177,
- Erasmus+ project AISE under grant 2023-1-EL01-KA220-SCH-000157157,
- European Regional Development Fund under the grant KK.01.1.1.01.0009 (DAT-ACROSS),
- project Center of Competences for Smart Cities under the grant KK.01.2.2.03.0004,
- CEI project COVIDAi (305.6019-20),
- project Metalska jezgra Čakovec (KK.01.1.1.02.0023), and
- University of Rijeka scientific grants:
  - uniri-tehnic-18-275-1447 and
  - uniri-mladi-technic-22-61.

APPENDIX A

Laboratory floor plan with dimensions



Floor plan of the L8 laboratory	
Doctoral dissertation - Appendix A	
S. Baressi Šegota	M20:1
RITEH	
06/03/2024	

APPENDIX B

Python implementation of the LE algorithm

```
import numpy as np
from sympy import *
from xml.dom import minidom
import numpy as np
from numba import jit

q1 = Symbol('q1')
q2 = Symbol('q2')
q3 = Symbol('q3')
q4 = Symbol('q4')
q5 = Symbol('q5')
q6 = Symbol('q6')

dq1 = Symbol('dq1')
dq2 = Symbol('dq2')
dq3 = Symbol('dq3')
dq4 = Symbol('dq4')
dq5 = Symbol('dq5')
dq6 = Symbol('dq6')

ddq1 = Symbol('ddq1')
ddq2 = Symbol('ddq2')
ddq3 = Symbol('ddq3')
ddq4 = Symbol('ddq4')
ddq5 = Symbol('ddq5')
```

```
    ddq6 = Symbol('ddq6')

    a1 = Symbol('a1')
    a2 = Symbol('a2')
    a3 = Symbol('a3')
    a4 = Symbol('a4')
    a5 = Symbol('a5')
    a6 = Symbol('a6')

    m1 = Symbol('m1')
    m2 = Symbol('m2')
    m3 = Symbol('m3')
    m4 = Symbol('m4')
    m5 = Symbol('m5')
    m6 = Symbol('m6')

    d1 = Symbol('d1')
    d2 = Symbol('d2')
    d3 = Symbol('d3')
    d4 = Symbol('d4')
    d5 = Symbol('d5')
    d6 = Symbol('d6')

    f1 = Symbol('f1')
    f2 = Symbol('f2')
    f3 = Symbol('f3')
    f4 = Symbol('f4')
    f5 = Symbol('f5')
    f6 = Symbol('f6')


    q__ = [q1, q2, q3, q4, q5, q6]
    dq__ = [dq1, dq2, dq3, dq4, dq5, dq6]
    ddq__ = [ddq1, ddq2, ddq3, ddq4, ddq5, ddq6]
    a__ = [a1, a2, a3, a4, a5, a6]
```

```
    m__ = [m1, m2, m3, m4, m5, m6]
    d__ = [d1, d2, d3, d4, d5, d6]
    f__ = [f1, f2, f3, f4, f5, f6]

    robot = minidom.parse("robot.irml")

    DEBUG = False
    import sys

    # get general values
    name = robot.getElementsByTagName("name")[0].childNodes[0].data
    manufacturer = robot.getElementsByTagName("manufacturer")[0].childNodes
[0].data
    joints = len(robot.getElementsByTagName("joint"))
    capacity = robot.getElementsByTagName("carrying-capacity")[0].
childNodes[0].data
    reach = robot.getElementsByTagName("reach")[0].childNodes[0].data
    configuration = robot.getElementsByTagName("configuration")[0].
childNodes[0].data
    if DEBUG:
        print("MANUFACTURER=",manufacturer)
        print("NAME=", name)
        print("JOINTS=",joints)
        print("CAPACITY=",capacity,"kg")
        print("REACH=", reach, "m")
        print("TYPE=", configuration)

    #variables for storage
    T = []
    R = []
    c = []
    D = []
    m = []
    a_s = []
    xi = []
```

```
93      # get individual joint values
94      count = 0
95      for joint in robot.getElementsByTagName("joint"):
96          xi.append(int(joint.getElementsByTagName("xi")[0].childNodes[0].
    data))
97          for kinematics_parameter in joint.getElementsByTagName("kinematics"
    ):
98              # get denavit-hartenberg parameters, convert them to symbols
99              q = Symbol(kinematics_parameter.getElementsByTagName("q")[0].
    childNodes[0].data)
100             d = Symbol(kinematics_parameter.getElementsByTagName("d")[0].
    childNodes[0].data)
101             r = Symbol(kinematics_parameter.getElementsByTagName("r")[0].
    childNodes[0].data)
102             a = Symbol(kinematics_parameter.getElementsByTagName("a")[0].
    childNodes[0].data)
103             print(r)
104
105             # calculate joints transformation matrix
106             T_ = np.array([(cos(q__[count]), -sin(q__[count])*cos(a), sin(
    q__[count])*sin(a), r*cos(q__[count])),
107                            (sin(q__[count]), cos(q__[count])*cos(a), -cos(q__[
    count])*sin(a), r*sin(q__[count])),
108                            (0, sin(a), cos(a), d),
109                            (0,0,0,1)])
110             # store value to matrix
111             T.append(T_)
112
113             if DEBUG:
114                 print("T=",T_)
115                 print("R=",T_[0:3,0:3])
116
117         # Get dynamics-related parameters
118         # get mass of the joint
119         for dynamics_parameter in joint.getElementsByTagName("dynamics"):
```

```
120             mass = dynamics_parameter.getElementsByTagName("link-mass")[0].
    childNodes[0].data
121             if DEBUG:
122                 print("m=", mass)
123             m.append(float(mass))
124             # get joint mass center coordinates
125             center_coordinates = dynamics_parameter.getElementsByTagName("
    center")
126             for coordinate in center_coordinates:
127                 x = coordinate.getElementsByTagName("x")[0].childNodes[0].
    data
128                 y = coordinate.getElementsByTagName("y")[0].childNodes[0].
    data
129                 z = coordinate.getElementsByTagName("z")[0].childNodes[0].
    data
130                 c_ = np.array([[x], [y], [z], [1]]).astype(float)
131                 c.append(c_)
132                 if DEBUG:
133                     print("c=", c_)
134
135             # get tensor of inertia from the XML
136             tensor_elements = dynamics_parameter.getElementsByTagName("
    inertia")
137             for element in tensor_elements:
138                 xx = element.getElementsByTagName("xx")[0].childNodes[0].
    data
139                 xy = element.getElementsByTagName("xy")[0].childNodes[0].
    data
140                 xz = element.getElementsByTagName("xz")[0].childNodes[0].
    data
141                 yx = element.getElementsByTagName("yx")[0].childNodes[0].
    data
142                 yy = element.getElementsByTagName("yy")[0].childNodes[0].
    data
```

```
143                 yz = element.getElementsByTagName("yz")[0].childNodes[0].
    data
144                 zx = element.getElementsByTagName("zx")[0].childNodes[0].
    data
145                 zy = element.getElementsByTagName("zy")[0].childNodes[0].
    data
146                 zz = element.getElementsByTagName("zz")[0].childNodes[0].
    data
147
148                 D_ = np.array([[xx, xy, xz], [yx, yy, yz], [zx, zy, zz]]).
    astype(float)
149
150                 D.append(D_)
151                 if DEBUG:
152                     print("D=", D_)
153                     print(10*'----------')
154
155     # Get R values of tranformation matrices
156     # first joint is I
157     R.append(np.eye(3,3))
158     # second is from base to joint 1
159     R.append(T[0][:3, :3])
160     # third is from base to joint two, hence the multiplication
161     R.append(T[0].dot(T[1])[:3, :3])
162     #and so on for the rest of the joints
163     R.append(T[0].dot(T[1]).dot(T[2])[:3, :3])
164     R.append(T[0].dot(T[1]).dot(T[2]).dot(T[3])[:3, :3])
165     R.append(T[0].dot(T[1]).dot(T[2]).dot(T[3]).dot(T[4])[:3, :3])
166     R.append(T[0].dot(T[1]).dot(T[2]).dot(T[3]).dot(T[4]).dot(T[5])[:3,
    :3])
167
168     ### first iteration
169     N=joints
170     z_arr = []
171     c_arr = []
```

```
    D_arr = []
    A_q_arr = []
    B_q_arr = []
    D_q_arr = []
    # flipped because of shaping and order of dims in numpy
    H1 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, 0]])
    for i in range(N):
        print("First iterative part, joint ", i)
    # 5
        z_ = R[i]@np.array([0, 0, 1]).T
        T_0_i = T[0]
        for j in range(0, i):
            T_0_i=T_0_i@T[j]

        c_ = c[i].T@(T_0_i@H1)
        D_ = R[i]@D[i]@(R[i].T)

        z_arr.append(z_)
        c_arr.append(c_)
        D_arr.append(D_)
        # 6
        #set the initial A_q_
        #all A_q_'s have the first
        A_q_ = np.array(diff(c_arr[0], q__[0])).T.reshape(3,1)
        B_q_ = np.array(xi[0]*z_arr[0]).T.reshape(3,1)
        # fill it with the rest of the differentials
        for j in range(1, i+1):
            A_q_=np.hstack((A_q_, np.array(diff(c_arr[j], q__[j])).T.
    reshape(3,1)))
            B_q_=np.hstack((B_q_, np.array(xi[j]*z_arr[j]).T.reshape(3,1)))

        # pad the remainder
        A_q_ = np.hstack((A_q_, np.zeros((3, N-A_q_.shape[1]))))
        B_q_ = np.hstack((B_q_, np.zeros((3, N-B_q_.shape[1]))))

```

```
206         A_q_arr.append(A_q_)
207         B_q_arr.append(B_q_)
208     # 7
209         D_q_ = A_q_arr[0].T@(m[0]*A_q_arr[0]) + B_q_arr[0].T@D_arr[0]
    @B_q_arr[0]
210         for j in range(1, i+1):
211             D_q_+= A_q_arr[j].T@(m[j]*A_q_arr[j]) + B_q_arr[j].T@D_arr[j]
    @B_q_arr[j]
212         D_q_arr.append(D_q_)
213
214     #9
215     # create an empty placeholder for filling up the speed connectivity
216     C = [[[0 for i in range(N)] for i in range(N)] for i in range(N)]
217     g = np.array([[0], [-9.81], [0]])
218     taus = []
219     for i in range(N):
220         print("Second iterative part, joint ", i+1, "step 1 - speed
    connectivity tensor")
221         for j in range(N):
222             for k in range(N):
223                 C[i][k][j] = diff(D_q_arr[-1][i][j], q__[k])-(1/2)*diff(
    D_q_arr[-1][k][j], q__[i])
224         print("Second iterative part, joint ", i+1, "step 2 - gravity
    influence")
225         h=0
226         for k in range(3):
227             for j in range(i, N):
228                 h += g[k]*m[i]+A_q_arr[i][k][j]
229         print("Second iterative part, joint ", i+1, "step 3 - friction")
230
231
232         mu_k = 0.504 # 0.5--1.0
233         mu_s = 0.302 # 0.3 -- 0.8
234         b = mu_k * sign(dq__[i]) * (1 - exp(-(abs(dq__[i]) - mu_s) / mu_k))
235
```

```
236
237         # placeholder variables for calculating tau subsums
238         D_SUM = 0
239         C_SUM = 0
240         print("Second iterative part, joint ", i+1, "step 4a - inertia sum"
    )
241         for j in range(N):
242             D_SUM += D_q_arr[-1][i][j]*ddq__[j]
243         print("Second iterative part, joint ", i+1, "step 4b - speed sum")
244         for k in range(3):
245             for j in range(i, N):
246                 C_SUM+=C[i][k][j]*dq__[k]*dq__[j]
247
248         print("Second iterative part, joint ", i+1, "step 4c - summing tau"
    )
249
250         print(50*'-'+"\n"+str(abs(b))+"\n"+50*'-'+"\n")
251
252         taus.append(abs(D_SUM+C_SUM+h+abs(b)))
253
254     print("Writing taus to file:")
255     from threading import Thread
256     def write_tau(id):
257         file=open("b"+str(b_type)+"tau_"+str(id+1)+"_le_"".py", "w")
258         expr = str(taus[id])
259         file.write("from numpy import sin, cos, tan, pi, sign, exp\n")
260         file.write("from numpy import absolute as Abs\n")
261         file.write("def tau_"+str(id+1)+"(q1, q2, q3, q4, q5, q6, dq1, dq2,
     dq3, dq4, dq5, dq6, ddq1, ddq2, ddq3, ddq4, ddq5, ddq6):\n")
262         file.write("\t return "+expr)
263         file.close()
264     threads=[]
265     threads=[]
266     for i in range(len(taus)):
267         t = Thread(target=write_tau, args=(i,))
```

```
268             threads.append(t)
269             t.start()
270 
271         for t in threads:
272             t.join()
```

Listing B.1: Implementation of the LE algorithm into the Python code

APPENDIX C

IRML example for ABB IRB 120 IRM

```
<?irml version = "0.1" encoding = "UTF-8" ?>
<robot>
    <!--General information-->
    <name>IRB 120</name>
    <manufacturer>ABB</manufacturer>
    <carrying-capacity>3.0</carrying-capacity>
    <reach>0.6</reach>
    <configuration>Articulated</configuration>
    <ir-type>Serial</ir-type>
    <manipulator-mass>25</manipulator-mass>
    <spec-sheet>https://search.abb.com/library/Download.aspx?DocumentID
=3HAC035960-001</spec-sheet>
    <iec590>IP30</iec590>
    <color>ABB Orange</color>
    <power-consumption>
        <iso-cube-max-speed>0.240</iso-cube-max-speed>
        <zero-position-brakes-off>0.095</zero-position-brakes-off>
        <zero-position-brakes-on>0.175</zero-position-brakes-on>
    </power-consumption>
    <ambient>
        <minimal-operating-temperature>5</minimal-operating-temperature
>
        <maximal-operating-temperature>45</maximal-operating-
temperature>
        <minimal-storage-temperature>-25</minimal-storage-temperature>
        <maximal-storage-temperature>55</maximal-storage-temperature>
        <maximal-short-period-temperature>70</maximal-short-period-
temperature>
```

```
25              <maximal-humidity>95</maximal-humidity>
26          </ambient>
27          <loads>
28              <endurance-loads>
29                  <Fx>265</Fx>
30                  <Fy>264</Fy>
31                  <Fz>200</Fz>
32                  <Tx>195</Tx>
33                  <Ty>195</Ty>
34                  <Tz>85</Tz>
35              </endurance-loads>
36              <maximal-loads>
37                  <Fx>515</Fx>
38                  <Fy>515</Fy>
39                  <Fz>365</Fz>
40                  <Tx>400</Tx>
41                  <Ty>400</Ty>
42                  <Tz>155</Tz>
43              </maximal-loads>
44          </loads>
45          <joint ID="1">
46              <type>T</type>
47              <xi>1</xi>
48              <kinematics>
49                  <q>q1</q>
50                  <d>0.29</d>
51                  <r>0</r>
52                  <a>-pi/2</a>
53              </kinematics>
54              <dynamics>
55                  <link-mass>3.06700626</link-mass>
56                  <center>
57                      <x>0.00009765</x>
58                      <y>0.23841163</y>
59                      <z>0.00011925</z>
```

```
                </center>
                <inertia>
                    <xx>-0.00615871</xx>
                    <xy>0.99996896</xy>
                    <xz>-0.00491487</xz>
                    <yx>-0.99767761</yx>
                    <yy>-0.00647786</yy>
                    <yz>-0.06780436</yz>
                    <zx>-0.06783409</zx>
                    <zy>0.00448587</zy>
                    <zz>0.99768653</zz>
                </inertia>
            </dynamics>
        </joint>
        <joint ID="2">
            <type>R</type>
            <xi>1</xi>
            <kinematics>
                <q>q2</q>
                <d>0</d>
                <r>0.27</r>
                <a>0</a>
            </kinematics>
            <dynamics>
                <link-mass>3.90863939</link-mass>
                <center>
                    <x>0.00077828</x>
                    <y>0.39124254</y>
                    <z>0.00211700</z>
                </center>
                <inertia>
                    <xx>0.00165516</xx>
                    <xy>0.99947723</xy>
                    <xz>0.03228827</xz>
                    <yx>-0.00062529</yx>
```

```
                    <yy>-0.03228727</yy>
                    <yz>0.99947843</yz>
                    <zx>0.99999843</zx>
                    <zy>-0.00167449</zy>
                    <zz>0.00057152</zz>
                </inertia>
            </dynamics>
        </joint>
        <joint ID="3">
            <type>R</type>
            <xi>1</xi>
            <kinematics>
                <q>q3</q>
                <d>0</d>
                <r>0.07</r>
                <a>-pi/2</a>
            </kinematics>
            <dynamics>
                <link-mass>2.94370258</link-mass>
                <center>
                    <x>-0.02280820</x>
                    <y>0.61791059</y>
                    <z>-0.00843547</z>
                </center>
                <inertia>
                    <xx>0.95772239</xx>
                    <xy>-0.28731840</xy>
                    <xz>0.01469572</xz>
                    <yx>0.28768727</yx>
                    <yy>0.95679871</yy>
                    <yz>-0.04209820</yz>
                    <zx>-0.00196526</zx>
                    <zy>0.04454616</zy>
                    <zz>0.99900539</zz>
                </inertia>
```

```
130             </dynamics>
131         </joint>
132         <joint ID="4">
133             <type>T</type>
134             <xi>1</xi>
135             <kinematics>
136                 <q>q4</q>
137                 <d>0.302</d>
138                 <r>0</r>
139                 <a>pi/2</a>
140             </kinematics>
141             <dynamics>
142                 <link-mass>1.44756525</link-mass>
143                 <center>
144                     <x>-0.22617555</x>
145                     <y>0.63037204</y>
146                     <z>-0.00456824</z>
147                 </center>
148                 <inertia>
149                     <xx>0.99316721</xx>
150                     <xy>0.00703297</xy>
151                     <xz>0.11648791</xz>
152                     <yx>0.11654819</yx>
153                     <yy>-0.00887405</yy>
154                     <yz>-0.99314539</yz>
155                     <zx>-0.00595104</zx>
156                     <zy>0.99993589</zy>
157                     <zz>-0.00963309</zz>
158                 </inertia>
159             </dynamics>
160         </joint>
161         <joint ID="5">
162             <type>R</type>
163             <xi>1</xi>
164             <kinematics>
```

```
                <q>q5</q>
                <d>0</d>
                <r>0</r>
                <a>-pi/2</a>
            </kinematics>
            <dynamics>
            <link-mass>0.01367842</link-mass>
                <center>
                    <x>-0.36693791</x>
                    <y>0.62999868</y>
                    <z>-0.00016958</z>
                </center>
                <inertia>
                    <xx>0.00006718</xx>
                    <xy>1.00000000</xy>
                    <xz>-0.00006971</xz>
                    <yx>0.00597413</yx>
                    <yy>0.00006930</yy>
                    <yz>0.99998215</yz>
                    <zx>0.99998215</zx>
                    <zy>-0.00006759</zy>
                    <zz>-0.00597412</zz>
                </inertia>
            </dynamics>
        </joint>
        <joint ID="6">
            <type>T</type>
            <xi>1</xi>
            <kinematics>
                <q>q6</q>
                <d>0.072</d>
                <r>0</r>
                <a>0</a>
            </kinematics>
            <dynamics>
```

```
                <link-mass>0.01367842</link-mass>
                <center>
                    <x>-0.36693791</x>
                    <y>0.62999868</y>
                    <z>-0.00016958</z>
                </center>
                <inertia>
                    <xx>0.00006718</xx>
                    <xy>1.00000000</xy>
                    <xz>-0.00006971</xz>
                    <yx>0.00597413</yx>
                    <yy>0.00006930</yy>
                    <yz>0.99998215</yz>
                    <zx>0.99998215</zx>
                    <zy>-0.00006759</zy>
                    <zz>-0.00597412</zz>
                </inertia>
            </dynamics>
        </joint>
    </robot>
```

Listing C.1: An IRML example for ABB IRB 120 IRM.

APPENDIX D

RAPID modules used for performing measurements

D.1 RAPID code for randomized path generation

```
MODULE Module1

!Define the upper and lower bounds of joints, grabbed by observing
kinematics
CONST num J1_LO := -120;
CONST num J1_HI := 120;
CONST num J2_LO := -55;
CONST num J2_HI := 70;
CONST num J3_LO := -100;
CONST num J3_HI := 40;
CONST num J4_LO := -160;
CONST num J4_HI := 160;
CONST num J5_LO := -120;
CONST num J5_HI := 120;
CONST num J6_LO := -400;
CONST num J6_HI := 400;

!Possible number of zones and speeds
CONST num SPEED_NUMBER:=17;
CONST num ZONE_NUMBER:=14;

!Possible speed and zone values
```

```
22      CONST speeddata SPEED_ARR{SPEED_NUMBER}:=[v100, v150, v200,
23              v300, v400, v500,
24              v600, v800, v1000,
25              v1500, v2000, v2500,
26              v3000, v4000, v5000,
27              v6000, v7000];
28
29
30      CONST zonedata ZONE_ARR{ZONE_NUMBER}:=[fine, z0, z1, z5,
31              z10, z20, z30, z40,
32              z50, z60, z80, z100,
33              z150, z200];
34
35      !For normalizing RAND() to range <0,1>
36      CONST num RAND_MAX:=32767;
37
38      !Define variables for storing generated random values
39      VAR num J1;
40      VAR num J2;
41      VAR num J3;
42      VAR num J4;
43      VAR num J5;
44      VAR num J6;
45      VAR speeddata V;
46      VAR zonedata Z;
47
48      VAR string starttime;
49      VAR string endtime;
50
51      VAR num position;
52      VAR num speed;
53      VAR num torque;
54      VAR num exttorque;
55
56
```

```
57
58      !Define number of joint positions to be tested
59      CONST NUM SIMULATION_COUNT:=1000;
60
61      PROC main()
62          starttime := CTime();
63
64          FOR i FROM 0 TO SIMULATION_COUNT DO
65
66              !Write the simulation step, for info
67              TPWrite("INFO Simulation count=" + NumToStr(i,0) + "/" +
    NumToStr(SIMULATION_COUNT,0));
68
69              !Randomly select joint 1--6, speed and zone
70              J1 := ((RAND()/RAND_MAX)*(J1_HI-J1_LO))+J1_LO;
71              J2 := ((RAND()/RAND_MAX)*(J2_HI-J2_LO))+J2_LO;
72              J3 := ((RAND()/RAND_MAX)*(J3_HI-J3_LO))+J3_LO;
73              J4 := ((RAND()/RAND_MAX)*(J4_HI-J4_LO))+J4_LO;
74              J5 := ((RAND()/RAND_MAX)*(J5_HI-J5_LO))+J5_LO;
75              J6 := ((RAND()/RAND_MAX)*(J6_HI-J6_LO))+J6_LO;
76
77              V := SPEED_ARR{1+ROUND((RAND()/RAND_MAX)*(SPEED_NUMBER-1))};
78              Z := ZONE_ARR{1+ROUND((RAND()/RAND_MAX)*(ZONE_NUMBER-1))};
79
80              !Print the selected values for monitoring
81              !TPWrite(NumtoStr(J1,2)+","+NumtoStr(J2,2)+","+NumtoStr(J3,2)
    +","+NumtoStr(J4,2)+","+NumtoStr(J5,2)+","+NumtoStr(J6,2));
82
83
84              MoveAbsJ [[J1,J2,J3,J4,J5,J6],[9E9,9E9,9E9,9E9,9E9,9E9]],V,Z,
    tool0;
85
86              ! TPWrite(NumtoStr(speed,2)+","+NumToStr(torque,2)+","+NumToStr
    (torque*speed,2));
87
```

```
88          ENDFOR
89          TPWrite("Start time: " + starttime);
90          TPWrite("End time: " + CTime());
91          EXIT;
92      ENDPROC
93  ENDMODULE
```

Listing D.1: RAPID code for random path generation

D.2 RAPID measurement of relevant physical values of IRM during operation

```
1       MODULE measure
2       ! Define variables for storing position, speed, and torques
3
4       VAR NUM POS1;
5       VAR NUM POS2;
6       VAR NUM POS3;
7       VAR NUM POS4;
8       VAR NUM POS5;
9       VAR NUM POS6;
10
11      VAR NUM SPD1;
12      VAR NUM SPD2;
13      VAR NUM SPD3;
14      VAR NUM SPD4;
15      VAR NUM SPD5;
16      VAR NUM SPD6;
17
18      VAR NUM TOR1;
19      VAR NUM TOR2;
20      VAR NUM TOR3;
21      VAR NUM TOR4;
22      VAR NUM TOR5;
```

```
23      VAR NUM TOR6;
24
25      VAR NUM ETOR1;
26      VAR NUM ETOR2;
27      VAR NUM ETOR3;
28      VAR NUM ETOR4;
29      VAR NUM ETOR5;
30      VAR NUM ETOR6;
31
32
33      VAR IODEV LOGFILE;
34
35      VAR robtarget TCP;
36      VAR clock sim_clock;
37      VAR num time;
38
39
40
41      PROC main()
42          Open "HOME:" \File:=CDate()+"-"+NumToStr(GetTime(\Hour),0)+"-"+
        NumToStr(GetTime(\Min),0)+"-"+NumToStr(GetTime(\Sec),0)+"_MEASUREMENT.
        CSV", logfile \Write;
43          Write logfile, "t,q1,dq1,tau1,q2,dq2,tau2,"\NoNewLine;
44          Write logfile, "q3,dq3,tau3,q4,dq4,tau4,q5,dq5,tau5,"\NoNewLine;
45          Write logfile, "q6,dq6,tau6,x,y,z,e1,e2,e3,e4,psi,theta,phi";
46          ClkReset sim_clock;
47          ClkStart sim_clock;
48          WHILE TRUE DO
49              GetJointData \MechUnit:=ROB_1, 1 \Position:=POS1 \Speed:=SPD1 \
        Torque:=TOR1 \ExtTorque:=ETOR1;
50              GetJointData \MechUnit:=ROB_1, 2 \Position:=POS2 \Speed:=SPD2 \
        Torque:=TOR2 \ExtTorque:=ETOR2;
51              GetJointData \MechUnit:=ROB_1, 3 \Position:=POS3 \Speed:=SPD3 \
        Torque:=TOR3 \ExtTorque:=ETOR3;
```

```
52          GetJointData \MechUnit:=ROB_1, 4 \Position:=POS4 \Speed:=SPD4 \
    Torque:=TOR4 \ExtTorque:=ETOR4;
53          GetJointData \MechUnit:=ROB_1, 5 \Position:=POS5 \Speed:=SPD5 \
    Torque:=TOR5 \ExtTorque:=ETOR5;
54          GetJointData \MechUnit:=ROB_1, 6 \Position:=POS6 \Speed:=SPD6 \
    Torque:=TOR6 \ExtTorque:=ETOR6;
55
56          ! Read TCP position, x, y, z CPos, CTool
57          TCP := CRobT(\Tool:=tool0 \WObj:=wobj0);
58
59          time := ClkRead(sim_clock);
60
61          Write logfile, NumToStr(time,5)+","+NumtoStr(POS1,5)+","+
    NumtoStr(SPD1,5)+","+NumToStr(TOR1,5)+","\NoNewLine;
62          Write logfile, NumtoStr(POS2,5)+","+NumtoStr(SPD2,5)+","+
    NumToStr(TOR2,5)+","\NoNewLine;
63          Write logfile, NumtoStr(POS3,5)+","+NumtoStr(SPD3,5)+","+
    NumToStr(TOR3,5)+","\NoNewLine;
64          Write logfile, NumtoStr(POS4,5)+","+NumtoStr(SPD4,5)+","+
    NumToStr(TOR4,5)+","\NoNewLine;
65          Write logfile, NumtoStr(POS5,5)+","+NumtoStr(SPD5,5)+","+
    NumToStr(TOR5,5)+","\NoNewLine;
66          Write logfile, NumtoStr(POS6,5)+","+NumtoStr(SPD6,5)+","+
    NumToStr(TOR6,5)+","\NoNewLine;
67          Write logfile, NumToStr(TCP.trans.x,5)+","+NumToStr(TCP.trans.y
    ,5)+","+NumToStr(TCP.trans.z,5)+","\NoNewLine;
68          Write logfile, NumToStr(TCP.rot.q1, 5)+","+NumToStr(TCP.rot.q2,
     5)+","+NumToStr(TCP.rot.q3, 5)+","+NumToStr(TCP.rot.q4, 5)+","\
    NoNewLine;
69          Write logfile, NumToStr(EulerZYX(\X, TCP.rot),5)+","+NumToStr(
    EulerZYX(\Y, TCP.rot),5)+","+NumToStr(EulerZYX(\Z, TCP.rot),5);
70          ! Pause for 1/40 s
71          WaitTime 0.025;
72      ENDWHILE
73      Close logfile;
```

```
74      ENDPROC
75  ENDMODULE
```

Listing D.2: RAPID code for measurement

APPENDIX E

Developed models

E.1 Equations obtained with LE

LE equations link 1 – https://bit.ly/4byV4vs

LE equations link 2 – https://bit.ly/4bykIWA

LE equations link 3 – https://bit.ly/4hAh0HN

LE equations link 4 – https://bit.ly/4bugFoT

LE equations link 5 – https://bit.ly/4ixLGuu

LE equations link 6 – https://bit.ly/3Ftnqvh

E.2 Selected best performing ML-based models

MLP model E – https://bit.ly/4hAuPGf

MLP model E1 – https://bit.ly/4hwy7KB

MLP model E2 – https://bit.ly/4koUN2h

MLP model E3 – https://bit.ly/4bSEQxk

MLP model E4 – https://bit.ly/43VoNwC

MLP model E5 – https://bit.ly/43VoP7I

MLP model E6 – https://bit.ly/41QY1mP

Data scaler – https://bit.ly/3DLbtk4

APPENDIX F

MA algorithm code

```
import numpy as np
import time
import pickle
import warnings
import pandas as pd
import uuid
import sys
import itertools
POPULATION_SIZE = int(sys.argv[1])
GENERATIONS = int(sys.argv[2])
CROSSOVER_RATE = float(sys.argv[3])
MUTATION_RATE = float(sys.argv[4])

STARTING_ANGLES =  np.random.choice(np.arange(-1, 1, 0.05), 6)
ENDING_ANGLES = np.random.choice(np.arange(2, 3, 0.05), 6)
# set random value between 1 and 3
TIME = 1

MODEL = pickle.load(open(r"./fitness_models/E.pickle", 'rb'))
SCALER = pickle.load(open(r"./fitness_models/scaler_E.pickle", 'rb'))
class Gene():
    def __init__(self):
        self.gene = np.random.choice(np.arange(-10, 10.05, 0.05), (6,
6))
        self.fitness = 0
        self.probability = 0
        self.cumulative_probability = 0

```

```
28          def set_gene(self, gene):
29              self.gene = gene
30
31          def get_gene(self):
32              return self.gene
33
34          def set_fitness(self, fitness):
35              self.fitness = fitness
36
37          def get_fitness(self):
38              return self.fitness
39
40          def set_probability(self, probability):
41              self.probability = probability
42
43          def get_probability(self):
44              return self.probability
45
46          def set_cumulative_probability(self, cumulative_probability):
47              self.cumulative_probability = cumulative_probability
48
49          def get_cumulative_probability(self):
50              return self.cumulative_probability
51
52
53
54      def fitness(gene, time=TIME, y_starts=STARTING_ANGLES, y_ends=
    ENDING_ANGLES):
55
56          #iterate over the genes and make sure they are within the limits
    -10, 10
57          for i in range(6):
58              for j in range(6):
59                  if gene.get_gene()[i, j] < -10:
60                      gene.get_gene()[i, j] = -10
```

```
61                elif gene.get_gene()[i, j] > 10:
62                    gene.get_gene()[i, j] = 10
63
64        a1_1 = gene.get_gene()[0, 0]
65        a1_2 = gene.get_gene()[0, 1]
66        a1_3 = gene.get_gene()[0, 2]
67        a1_4 = gene.get_gene()[0, 3]
68        a1_5 = gene.get_gene()[0, 4]
69        a1_6 = gene.get_gene()[0, 5]
70        a2_1 = gene.get_gene()[1, 0]
71        a2_2 = gene.get_gene()[1, 1]
72        a2_3 = gene.get_gene()[1, 2]
73        a2_4 = gene.get_gene()[1, 3]
74        a2_5 = gene.get_gene()[1, 4]
75        a2_6 = gene.get_gene()[1, 5]
76        a3_1 = gene.get_gene()[2, 0]
77        a3_2 = gene.get_gene()[2, 1]
78        a3_3 = gene.get_gene()[2, 2]
79        a3_4 = gene.get_gene()[2, 3]
80        a3_5 = gene.get_gene()[2, 4]
81        a3_6 = gene.get_gene()[2, 5]
82        a4_1 = gene.get_gene()[3, 0]
83        a4_2 = gene.get_gene()[3, 1]
84        a4_3 = gene.get_gene()[3, 2]
85        a4_4 = gene.get_gene()[3, 3]
86        a4_5 = gene.get_gene()[3, 4]
87        a4_6 = gene.get_gene()[3, 5]
88        a5_1 = gene.get_gene()[4, 0]
89        a5_2 = gene.get_gene()[4, 1]
90        a5_3 = gene.get_gene()[4, 2]
91        a5_4 = gene.get_gene()[4, 3]
92        a5_5 = gene.get_gene()[4, 4]
93        a5_6 = gene.get_gene()[4, 5]
94        a6_1 = gene.get_gene()[5, 0]
95        a6_2 = gene.get_gene()[5, 1]
```

```
96          a6_3 = gene.get_gene()[5, 2]
97          a6_4 = gene.get_gene()[5, 3]
98          a6_5 = gene.get_gene()[5, 4]
99          a6_6 = gene.get_gene()[5, 5]
100
101         x_values=np.linspace(0, time, 10)
102
103             # calcule the individual paths
104
105         q1 = a1_6*x_values**5 + a1_5*x_values**4 + a1_4*x_values**3 + a1_3*
    x_values**2 + a1_2*x_values + a1_1
106         q2 = a2_6*x_values**5 + a2_5*x_values**4 + a2_4*x_values**3 + a2_3*
    x_values**2 + a2_2*x_values + a2_1
107         q3 = a3_6*x_values**5 + a3_5*x_values**4 + a3_4*x_values**3 + a3_3*
    x_values**2 + a3_2*x_values + a3_1
108         q4 = a4_6*x_values**5 + a4_5*x_values**4 + a4_4*x_values**3 + a4_3*
    x_values**2 + a4_2*x_values + a4_1
109         q5 = a5_6*x_values**5 + a5_5*x_values**4 + a5_4*x_values**3 + a5_3*
    x_values**2 + a5_2*x_values + a5_1
110         q6 = a6_6*x_values**5 + a6_5*x_values**4 + a6_4*x_values**3 + a6_3*
    x_values**2 + a6_2*x_values + a6_1
111
112         q1_min, q1_max = q1[0], q1[-1]
113         q1 = (q1 - q1_min) / (q1_max - q1_min) * (y_ends[0] - y_starts[0])
    + y_starts[0]
114         q2_min, q2_max = q2[0], q2[-1]
115         q2 = (q2 - q2_min) / (q2_max - q2_min) * (y_ends[1] - y_starts[1])
    + y_starts[1]
116         q3_min, q3_max = q3[0], q3[-1]
117         q3 = (q3 - q3_min) / (q3_max - q3_min) * (y_ends[2] - y_starts[2])
    + y_starts[2]
118         q4_min, q4_max = q4[0], q4[-1]
119         q4 = (q4 - q4_min) / (q4_max - q4_min) * (y_ends[3] - y_starts[3])
    + y_starts[3]
120         q5_min, q5_max = q5[0], q5[-1]
```

```
121         q5 = (q5 - q5_min) / (q5_max - q5_min) * (y_ends[4] - y_starts[4])
    + y_starts[4]
122         q6_min, q6_max = q6[0], q6[-1]
123         q6 = (q6 - q6_min) / (q6_max - q6_min) * (y_ends[5] - y_starts[5])
    + y_starts   [5]
124
125         dq1 = 5*a1_6*x_values**4 + 4*a1_5*x_values**3 + 3*a1_4*x_values**2
    + 2*a1_3*x_values + a1_2
126         dq2 = 5*a2_6*x_values**4 + 4*a2_5*x_values**3 + 3*a2_4*x_values**2
    + 2*a2_3*x_values + a2_2
127         dq3 = 5*a3_6*x_values**4 + 4*a3_5*x_values**3 + 3*a3_4*x_values**2
    + 2*a3_3*x_values + a3_2
128         dq4 = 5*a4_6*x_values**4 + 4*a4_5*x_values**3 + 3*a4_4*x_values**2
    + 2*a4_3*x_values + a4_2
129         dq5 = 5*a5_6*x_values**4 + 4*a5_5*x_values**3 + 3*a5_4*x_values**2
    + 2*a5_3*x_values + a5_2
130         dq6 = 5*a6_6*x_values**4 + 4*a6_5*x_values**3 + 3*a6_4*x_values**2
    + 2*a6_3*x_values + a6_2
131
132         ddq1 = 20*a1_6*x_values**3 + 12*a1_5*x_values**2 + 6*a1_4*x_values
    + 2*a1_3
133         ddq2 = 20*a2_6*x_values**3 + 12*a2_5*x_values**2 + 6*a2_4*x_values
    + 2*a2_3
134         ddq3 = 20*a3_6*x_values**3 + 12*a3_5*x_values**2 + 6*a3_4*x_values
    + 2*a3_3
135         ddq4 = 20*a4_6*x_values**3 + 12*a4_5*x_values**2 + 6*a4_4*x_values
    + 2*a4_3
136         ddq5 = 20*a5_6*x_values**3 + 12*a5_5*x_values**2 + 6*a5_4*x_values
    + 2*a5_3
137         ddq6 = 20*a6_6*x_values**3 + 12*a6_5*x_values**2 + 6*a6_4*x_values
    + 2*a6_3
138
139         x__ = []
140         y__ = []
141         z__ = []
```

```
142         pitch__ = []
143         roll__ = []
144         yaw__ = []
145         qx__ = []
146         qy__ = []
147         qz__ = []
148         qw__ = []
149 
150         for x_point in range(len(x_values)):
151             q_1 = q1[x_point]
152             q_2 = q2[x_point]
153             q_3 = q3[x_point]
154             q_4 = q4[x_point]
155             q_5 = q5[x_point]
156             q_6 = q6[x_point]
157             #print(q1, q2, q3, q4, q5, q6)
158             T = np.array([[((np.sin(q_1)*np.sin(q_4) + np.cos(q_1)*np.cos(
    q_4)*np.cos(q_2 + q_3))*np.cos(q_5) - np.sin(q_5)*np.sin(q_2 + q_3)*np.
    cos(q_1))*np.cos(q_6) + (np.sin(q_1)*np.cos(q_4) - np.sin(q_4)*np.cos(
    q_1)*np.cos(q_2 + q_3))*np.sin(q_6), -((np.sin(q_1)*np.sin(q_4) + np.cos
    (q_1)*np.cos(q_4)*np.cos(q_2 + q_3))*np.cos(q_5) - np.sin(q_5)*np.sin(
    q_2 + q_3)*np.cos(q_1))*np.sin(q_6) + (np.sin(q_1)*np.cos(q_4) - np.sin(
    q_4)*np.cos(q_1)*np.cos(q_2 + q_3))*np.cos(q_6), -(np.sin(q_1)*np.sin(
    q_4) + np.cos(q_1)*np.cos(q_4)*np.cos(q_2 + q_3))*np.sin(q_5) - np.sin(
    q_2 + q_3)*np.cos(q_1)*np.cos(q_5), -0.147*np.sin(q_1)*np.sin(q_4)*np.
    sin(q_5) - 0.147*np.sin(q_5)*np.cos(q_1)*np.cos(q_4)*np.cos(q_2 + q_3) -
     0.147*np.sin(q_2 + q_3)*np.cos(q_1)*np.cos(q_5) - 0.302*np.sin(q_2 +
    q_3)*np.cos(q_1) + 0.27*np.cos(q_1)*np.cos(q_2) + 0.07*np.cos(q_1)*np.
    cos(q_2 + q_3)], [((np.sin(q_1)*np.cos(q_4)*np.cos(q_2 + q_3) - np.sin(
    q_4)*np.cos(q_1))*np.cos(q_5) - np.sin(q_1)*np.sin(q_5)*np.sin(q_2 + q_3
    ))*np.cos(q_6) - (np.sin(q_1)*np.sin(q_4)*np.cos(q_2 + q_3) + np.cos(q_1
    )*np.cos(q_4))*np.sin(q_6), -((np.sin(q_1)*np.cos(q_4)*np.cos(q_2 + q_3)
     - np.sin(q_4)*np.cos(q_1))*np.cos(q_5) - np.sin(q_1)*np.sin(q_5)*np.sin
    (q_2 + q_3))*np.sin(q_6) - (np.sin(q_1)*np.sin(q_4)*np.cos(q_2 + q_3) +
    np.cos(q_1)*np.cos(q_4))*np.cos(q_6), -(np.sin(q_1)*np.cos(q_4)*np.cos(
```

```
    q_2 + q_3) - np.sin(q_4)*np.cos(q_1))*np.sin(q_5) - np.sin(q_1)*np.sin(
    q_2 + q_3)*np.cos(q_5), -0.147*np.sin(q_1)*np.sin(q_5)*np.cos(q_4)*np.
    cos(q_2 + q_3) - 0.147*np.sin(q_1)*np.sin(q_2 + q_3)*np.cos(q_5) -
    0.302*np.sin(q_1)*np.sin(q_2 + q_3) + 0.27*np.sin(q_1)*np.cos(q_2) +
    0.07*np.sin(q_1)*np.cos(q_2 + q_3) + 0.147*np.sin(q_4)*np.sin(q_5)*np.
    cos(q_1)], [-(np.sin(q_5)*np.cos(q_2 + q_3) + np.sin(q_2 + q_3)*np.cos(
    q_4)*np.cos(q_5))*np.cos(q_6) + np.sin(q_4)*np.sin(q_6)*np.sin(q_2 + q_3
    ), (np.sin(q_5)*np.cos(q_2 + q_3) + np.sin(q_2 + q_3)*np.cos(q_4)*np.cos
    (q_5))*np.sin(q_6) + np.sin(q_4)*np.sin(q_2 + q_3)*np.cos(q_6), np.sin(
    q_5)*np.sin(q_2 + q_3)*np.cos(q_4) - np.cos(q_5)*np.cos(q_2 + q_3),
    -0.27*np.sin(q_2) + 0.147*np.sin(q_5)*np.sin(q_2 + q_3)*np.cos(q_4) -
    0.07*np.sin(q_2 + q_3) - 0.147*np.cos(q_5)*np.cos(q_2 + q_3) - 0.302*np.
    cos(q_2 + q_3) + 0.29], [0, 0, 0, 1]])
159
160
161
162         x, y, z = T[:3, 3]
163
164         R = T[:3, :3]
165
166         if np.abs(R[2, 0]) != 1:
167             pitch = -np.arcsin(R[2, 0])
168             roll = np.arctan2(R[2, 1] / np.cos(pitch), R[2, 2] / np.cos
    (pitch))
169             yaw = np.arctan2(R[1, 0] / np.cos(pitch), R[0, 0] / np.cos(
    pitch))
170         else:
171             # Gimbal lock case
172             yaw = 0  # Default yaw
173             if R[2, 0] == -1:
174                 pitch = np.pi / 2
175                 roll = np.arctan2(R[0, 1], R[0, 2])
176             else:
177                 pitch = -np.pi / 2
178                 roll = np.arctan2(-R[0, 1], -R[0, 2])
```

```
179
180             euler_angles = (roll, pitch, yaw)  # Output in radians
181
182             # Convert Rotation Matrix to Quaternion
183             trace = np.trace(R)
184             if trace > 0:
185                 S = 2.0 * np.sqrt(trace + 1.0)
186                 qw = 0.25 * S
187                 qx = (R[2, 1] - R[1, 2]) / S
188                 qy = (R[0, 2] - R[2, 0]) / S
189                 qz = (R[1, 0] - R[0, 1]) / S
190             elif (R[0, 0] > R[1, 1]) and (R[0, 0] > R[2, 2]):
191                 S = 2.0 * np.sqrt(1.0 + R[0, 0] - R[1, 1] - R[2, 2])
192                 qw = (R[2, 1] - R[1, 2]) / S
193                 qx = 0.25 * S
194                 qy = (R[0, 1] + R[1, 0]) / S
195                 qz = (R[0, 2] + R[2, 0]) / S
196             elif R[1, 1] > R[2, 2]:
197                 S = 2.0 * np.sqrt(1.0 + R[1, 1] - R[0, 0] - R[2, 2])
198                 qw = (R[0, 2] - R[2, 0]) / S
199                 qx = (R[0, 1] + R[1, 0]) / S
200                 qy = 0.25 * S
201                 qz = (R[1, 2] + R[2, 1]) / S
202             else:
203                 S = 2.0 * np.sqrt(1.0 + R[2, 2] - R[0, 0] - R[1, 1])
204                 qw = (R[1, 0] - R[0, 1]) / S
205                 qx = (R[0, 2] + R[2, 0]) / S
206                 qy = (R[1, 2] + R[2, 1]) / S
207                 qz = 0.25 * S
208
209             x__.append(x)
210             y__.append(y)
211             z__.append(z)
212             pitch__.append(pitch)
213             roll__.append(roll)
```

```
214             yaw__.append(yaw)
215             qx__.append(qx)
216             qy__.append(qy)
217             qz__.append(qz)
218             qw__.append(qw)
219
220
221
222         dx = np.gradient(x__)
223         dy = np.gradient(y__)
224         dz = np.gradient(z__)
225         dphi = np.gradient(pitch__)
226         dtheta = np.gradient(roll__)
227         dpsi = np.gradient(yaw__)
228         ddx = np.gradient(dx)
229         ddy = np.gradient(dy)
230         ddz = np.gradient(dz)
231         ddphi = np.gradient(dphi)
232         ddtheta = np.gradient(dtheta)
233         ddpsi = np.gradient(dpsi)
234
235         E_path = 0
236         for i in range(10):
237             input_vector = SCALER.transform(np.array([q1[i], dq1[i], q2[i],
    dq2[i],
238             q3[i], dq3[i], q4[i], dq4[i], q5[i],
239             dq5[i], q6[i], dq6[i], x__[i], y__[i],
240             z__[i], qx__[i], qy__[i], qz__[i],
241             qw__[i], yaw__[i], pitch__[i], roll__[i],
242             ddq1[i], ddq2[i], ddq3[i], ddq4[i],
243             ddq5[i], ddq6[i], dx[i], dy[i], dz[i],
244             dphi[i], dtheta[i], dpsi[i], ddx[i], ddy[i],
245             ddz[i], ddphi[i], ddtheta[i], ddpsi[i]]).reshape(1, -1))
246             E = MODEL.predict(input_vector)
247             E_path += abs(E)
```

```
248
249         return E_path
250
251     def selection(pop, total_fitness):
252         # generate a random number between 0 and 1
253         r = np.random.rand()
254         for gene in pop:
255             if r <= gene.get_cumulative_probability():
256                 return gene
257
258     def crossover_random(gene1, gene2):
259         # itearete over the genes
260         for i in range(6):
261             for j in range(6):
262                 # generate a random number between 0 and 1
263                 r = np.random.rand()
264                 if r < 0.5:
265                     gene1.get_gene()[i, j] = gene2.get_gene()[i, j]
266         return gene1
267
268     def crossover_average(gene1, gene2):
269         for i in range(6):
270             for j in range(6):
271                 gene1.get_gene()[i, j] = (gene1.get_gene()[i, j] + gene2.
    get_gene()[i, j])/2
272         return gene1
273
274     def crossover_differential(gene1, gene2, gene3, f):
275
276         for i in range(6):
277             for j in range(6):
278                 gene1.get_gene()[i, j] = gene1.get_gene()[i, j] + f*(gene2.
    get_gene()[i, j] - gene3.get_gene()[i, j])
279         return gene1
280
```

```
    def mutation(gene1):
        # select a random value between 1 and 10
        n = np.random.randint(1, 10)
        # select n random i,j locations
        for i in range(n):
            i = np.random.randint(6)
            j = np.random.randint(6)
            gene1.get_gene()[i, j] = np.random.choice(np.arange(-10, 10.05,
     0.05))
        return gene1

    def randomized_local_search(gene, samples=100):
        modifications = [-0.25, -0.2, -0.15, -0.1, -0.05, 0, 0.05, 0.1,
    0.15, 0.2, 0.25]
        original_matrix = gene.get_gene()
        orig_fitness = gene.get_fitness()
        # Generate all possible modifications for the 36 elements
        for _ in range(samples):
            random_modifications = np.random.choice(modifications, size
    =(6,6))
            modified_matrix = original_matrix + random_modifications

            temp_gene = Gene()
            temp_gene.set_gene(modified_matrix)
            fitness_ = fitness(temp_gene)

            if fitness(temp_gene) < gene.get_fitness():
                gene.set_gene(modified_matrix)
                gene.set_fitness(fitness_)
        return gene

    def informed_local_search(gene, steps=20):
        original_matrix = gene.get_gene()
        original_fitness = gene.get_fitness()
        # get first and second row
```

```
313
314         first_row = gene.get_gene()[0, :]
315         second_row = gene.get_gene()[1, :]
316         third_row = gene.get_gene()[2, :]
317         fourth_row = gene.get_gene()[3, :]
318         fifth_row = gene.get_gene()[4, :]
319         sixth_row = gene.get_gene()[5, :]
320         # modify those values so they tend closer to zero
321         set_flag=False
322         for i in range(steps):
323             for i in range(6):
324                 first_row[i] = first_row[i] * 0.9
325                 second_row[i] = second_row[i] * 0.9
326             modified_matrix = np.vstack((first_row, second_row, third_row,
    fourth_row, fifth_row, sixth_row))
327             temp_gene = Gene()
328             temp_gene.set_gene(modified_matrix)
329
330             fitness_ = fitness(gene)
331             if fitness(temp_gene) < gene.get_fitness():
332                 gene.set_gene(modified_matrix)
333                 gene.set_fitness(fitness_)
334                 set_flag=True
335
336         if not(set_flag):
337             gene.set_gene(original_matrix)
338
339         return gene
340
341     alg_time_start = time.time()
342     # crete a list of genes
343     pop = [Gene() for i in range(POPULATION_SIZE)]
344
345     for gene in pop:
346         gene.set_fitness(fitness(gene))
```

```
347
348     improvement = []
349     for t in range(GENERATIONS):
350         generation_time_start = time.time()
351         # calculate the total fitness of the population
352         total_fitness = sum([1.0/gene.get_fitness() for gene in pop])
353         for gene in pop:
354             gene.set_probability(1.0/gene.get_fitness()/total_fitness)
355         cumulative_probability = 0
356         for gene in pop:
357             cumulative_probability += gene.get_probability()
358             gene.set_cumulative_probability(cumulative_probability)
359         # select one gene
360
361         for gene in pop:
362
363             gene1 = selection(pop, total_fitness)
364             r = np.random.rand()
365
366             if r < CROSSOVER_RATE:
367                 gene2 = selection(pop, total_fitness)
368                 gene_out = crossover_random(gene1, gene2)
369             elif r < CROSSOVER_RATE+MUTATION_RATE:
370                 gene_out = mutation(gene1)
371             else:
372                 gene_out = gene1
373
374             new_fitness = fitness(gene_out)
375
376             if new_fitness < gene.get_fitness():
377                 gene.set_gene(gene_out.get_gene())
378                 gene.set_fitness(new_fitness)
379
380         # sort the population by fitness, from lowest to highest
381
```

```
382         pop.sort(key=lambda x: x.get_fitness())
383         pop[0] = informed_local_search(pop[0])
384         improvement.append(pop[0].get_fitness())
385         print(pop[0].get_fitness(), " in ", time.time() -
    generation_time_start)
386
387     print('Algorithm time:', time.time() - alg_time_start)
388
389     # save the improvement list to a csv file
390     uuid__ = uuid.uuid4()
391     np.savetxt(str(uuid__)+'_improvement.csv', improvement, delimiter=',')
392
393     np.savetxt(str(uuid__)+'_best_gene.csv', pop[0].get_gene(), delimiter='
    ,')
394     np.savetxt(str(uuid__)+'_angles_in_'+str(TIME)+'.csv', np.vstack((
    STARTING_ANGLES, ENDING_ANGLES)), delimiter=',')
395
396     print(np.round(pop[0].get_gene(),2))
397     print(pop[0].get_fitness())
```

Listing F.1: MA algorithm implemented in Python

APPENDIX G

RAPID code for final path comparison

```
1
2     MODULE MainModule
3     VAR robtarget p1;
4     VAR robtarget p2;
5     CONST num X := 10;  ! Number of repetitions
6
7     VAR jointtarget e1:=[[5.73,11.46,22.92,68.75,68.75,45.83],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
8     VAR jointtarget e2:=[[8.12,12.06,20.82,69.08,69.10,50.17],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
9     VAR jointtarget e3:=[[10.59,12.68,19.54,69.47,69.78,53.93],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
10    VAR jointtarget e4:=[[12.99,13.32,19.25,70.00,70.81,56.98],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
11    VAR jointtarget e5:=[[15.45,13.99,20.12,70.85,72.15,59.27],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
12    VAR jointtarget e6:=[[17.98,14.68,22.34,72.28,73.74,60.96],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
13    VAR jointtarget e7:=[[20.59,15.39,26.17,74.59,75.48,62.59],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
14    VAR jointtarget e8:=[[23.27,16.08,31.97,78.23,77.25,65.14],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
15    VAR jointtarget e9:=[[25.98,16.71,40.22,83.72,78.88,70.23],[9E9,9E9,9E9
      ,9E9,9E9,9E9]];
16    VAR jointtarget e10:=[[28.65,17.19,51.56,91.67,80.21,80.21],[9E9,9E9,9
      E9,9E9,9E9,9E9]];
17    PROC main()
18        ! Convert joint positions to Cartesian positions
```

```
        p1 := CalcRobT(e1, tool0);
        p2 := CalcRobT(e1, tool0);

        ! Move to the initial position
        MoveAbsJ e1, v100, z10, tool0;
        WaitTime 2.000;
        ! Repeat movement X times
        FOR i FROM 1 TO X DO
            MoveL p2, v100, z10, tool0;  ! Move in a straight line to
    position 2
            MoveL p1, v100, z10, tool0;  ! Move back to position 1
        ENDFOR


        MoveAbsJ e1, v100, z10, tool0;
        WaitTime 2.000;

        FOR i FROM 1 TO X DO
            MoveAbsJ e1, v100, z10, tool0;
            MoveAbsJ e2, v100, z10, tool0;
            MoveAbsJ e3, v100, z10, tool0;
            MoveAbsJ e4, v100, z10, tool0;
            MoveAbsJ e5, v100, z10, tool0;
            MoveAbsJ e6, v100, z10, tool0;
            MoveAbsJ e7, v100, z10, tool0;
            MoveAbsJ e8, v100, z10, tool0;
            MoveAbsJ e9, v100, z10, tool0;
            MoveAbsJ e10, v100, z10, tool0;
        ENDFOR
    ENDPROC
ENDMODULE
```

Listing G.1: RAPID code for comparing paths