Chair of Astronautics
Prof. Prof. h.c. Dr. Dr. h.c.
Ulrich Walter

Technical University
of Munich

DEPARTMENT OF INFORMATICS

Master's Thesis
in
Robotics, Cognition, Intelligence

Development of a Simulation-Based Evaluation Framework for Mechatronic Systems

Entwicklung eines Tools zur simulationsbasierten Auswertung Mechatronischer Systeme

RT-MA 2020/08

Author: Michael Hermann Dyck

Submission date: 15 July, 2020

Supervisors:
M.Sc. Jonis Kiesbye
Chair of Astronautics
Technical University of Munich

M.Sc. Jonas Wittmann
Chair of Applied Mechanics
Technical University of Munich

External Supervisor:
Dr. Martin Bischoff
Siemens AG, Corporate Technology
Mechatronic Systems
München Perlach

Erklärung

Ich erkläre, dass ich alle Einrichtungen, Anlagen, Geräte und Programme, die mir im Rahmen meiner Semester- oder Masterarbeit von der TU München bzw. vom Lehrstuhl für Raumfahrttechnik zur Verfügung gestellt werden, entsprechend dem vorgesehenen Zweck, den gültigen Richtlinien, Benutzerordnungen oder Gebrauchsanleitungen und soweit nötig erst nach erfolgter Einweisung und mit aller Sorgfalt benutze. Insbesondere werde ich Programme ohne besondere Anweisung durch den Betreuer weder kopieren noch für andere als für meine Tätigkeit am Lehrstuhl vorgesehene Zwecke verwenden.

Mir als vertraulich genannte Informationen, Unterlagen und Erkenntnisse werde ich weder während noch nach meiner Tätigkeit am Lehrstuhl an Dritte weitergeben.

Ich erkläre mich außerdem damit einverstanden, dass meine Master- oder Semesterarbeit vom Lehrstuhl auf Anfrage fachlich interessierten Personen, auch über eine Bibliothek, zugänglich gemacht wird, und dass darin enthaltene Ergebnisse sowie dabei entstandene Entwicklungen und Programme vom Lehrstuhl für Raumfahrttechnik uneingeschränkt genutzt werden dürfen. (Rechte an evtl. entstehenden Programmen und Erfindungen müssen im Vorfeld geklärt werden.)

Ich erkläre außerdem, dass ich diese Masterarbeit ohne fremde Hilfe angefertigt und nur die in dem Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, den _____

_____

Unterschrift

Name: Michael Hermann Dyck

Matrikelnummer: 03706694

Zusammenfassung

Die sich ständig ändernde Nachfrage am Markt für mechatronische Systeme stellt Ingenieursfirmen vor die Herausforderung, Markteinführungszeit und Produktionskosten zu reduzieren, und gleichzeitig präzise und flexible Produktion zu gewährleisten [1]. Durch das stetige Testen verschiedener Produktdesigns und die kontinuierliche Bewertung der Systemperformance virtueller Prototypen, hilft das sogenannte *Virtuelle Prototyping* [2, 3] Ingenieursfirmen diesen Herausforderungen gerecht zu werden. Ziel dieser Arbeit ist die Entwicklung eines generischen Auswertungsframeworks, das es Produktdesignern und Domänenexperten erlaubt, virtuelle Prototypen über eine Vielzahl von Modellparametern für bestimmte Zielfunktionen effizient auszuwerten. Darüber hinaus wurden zwei Beispielanwendungen mit dem *Franka Emika Panda* Roboterarm implementiert, die die Fähigkeiten des Auswertungsframeworks demonstrieren. Im ersten der beiden Szenarien wird die Energieeffizienz verschiedener Trajektorien in einer Pick and Place Anwendung evaluiert. Das zweite Szenario modelliert einen robotischen Bin Picking Prozess in einer industriellen Anlage mit dem Ziel, kriterien-spezifische optimale Roboterpositionen zu identifizieren.

Entsprechend der klassischen Top-Down Vorgehensweise wurde das Auswertungsframework in drei Komponenten aufgeteilt: (1) Die Definition der zu testenden Designalternativen und auszuwertenden Zielfunktionen eines existierenden Simulationsmodells ist in *C#* und der Game Engine *Unity* implementiert. (2) Die zweite Komponente des Frameworks ist als *Microsoft .NET Core* Anwendung realisiert und nutzt Parallelisierungsmethoden, um das Simulationsmodell in *Unity* für alle definierten Designalternativen effizient zu evaluieren. (3) Mithilfe von *Python*'s *tkinter* Bibliothek visualisiert die dritte Frameworkkomponente die Puntkwolken im Auswertungsraum in einer interaktiven graphischen Benutzeroberfläche.

Die Evaluierungsergebnisse der Pick and Place Anwendung zeigen, dass für eine energieoptimale Trajektorie so wenig Bewegung im Gelenkraum wie möglich stattfinden muss. Dies zeigt, dass die Länge der Distanz, die im Gelenkraum zurückgelegt wird, den größten Einfluss auf den Energieverbrauch des Roboters hat.
Die Auswertung des Bin Picking Szenarios für verschiedene Roboterpositionen in der industriellen Anlage zeigt, dass nur ein sehr kleiner Spielraum für die Positionierung des Roboters zur Verfügung steht. Dies liegt daran, dass die Bewegunsreichweite des *Panda* Roboterarms im Vergleich zum Arbeitsraum in der Industrieanlage relativ klein ist. Darüber hinaus führen bestimmte Roboterpositionen gleichzeitig zu geringer Ausführungsdauer und Distanz im Gelenkraum.
Beide Szenarien demonstrieren die erfolgreiche Anwendung des kompletten Auswertungsframeworks und verdeutlichen die Möglichkeiten und Stärken des Frameworks in breiter und tiefgreifender Evaluierung virtueller Prototypen.

Aufgrund der generischen Umsetzung kommen in Zukunft unzählige weitere Anwendungsbereiche für das Auswertungsframework in Frage. Darüber hinaus ist es möglich, das Framework durch Integration weiterer Simulationssoftware und Implementierung von multi-kriteriellen Optimierungsalgorithmen zu erweitern.

Abstract

Quickly changing trends in market demand of mechatronic systems require engineering companies to reduce production costs and time-to-market and to increase the production flexibility and precision of product design [1]. *Virtual prototyping* [2, 3] aims at fulfilling these requirements, by utilizing virtual prototypes of mechatronic systems, in order to test system design and performance in any phase of the engineering development process. The main objective of this thesis is to develop a generic evaluation framework, that allows product designers and engineering domain experts to efficiently and meaningfully evaluate multiple criteria of virtual prototypes for a variety of model parameters in simulation. Furthermore, two exemplary applications involving the *Franka Emika Panda* robot arm are implemented, in order to demonstrate the capabilities of the developed framework. In the first scenario, different trajectories for robotic pick and place use cases are evaluated for their energy efficiency. The second scenario models a bin picking process of the *Panda* robot in an industrial plant and aims at identifying valid and objective-specific optimal robot positions inside the plant.

The evaluation framework was designed and implemented by following a top-down approach. Evaluating a mechatronic system was separated into three distinct steps and correspondingly the evaluation framework comprises three main components: (1) Defining the *evaluation parameters* and *objective values* of the evaluation from an existing simulation model, i.e. choosing the design alternatives to be tested and the corresponding decision criteria to be evaluated, is implemented in *Unity* in the *C#* programming language. (2) The evaluation framework's second component is implemented in *C#* as a *Microsoft .NET Core* solution and uses *.NET* parallelization methods to efficiently evaluate the executable of the *Unity* simulation model for all combinations of *evaluation parameters*. (3) Using *Python*'s *tkinter* library, the third component visualizes the resulting point clouds of all decision criteria for all design alternatives in an interactive graphical user interface (GUI).

The evaluation results of the pick and place application show that the energy-optimal robot trajectories in the simulated scenario correspond to those, where the robot moves as few joints as possible, showing that the amount of distance covered in joint space constitutes the major fraction of electric energy consumption on a trajectory.
Evaluating bin picking with the *Panda* robot in an industrial plant reveals, there is only very little scope available for positioning the robot such that it is able to successfully perform bin picking. This lies in the fact that the *Panda* robot's range of motion is relatively small compared to its workspace in the industrial plant. Furthermore, the results show, that robot positions resulting in trajectories with low execution time concurrently result in short joint distance.
In both exemplary scenarios, the complete evaluation framework was applied successfully. The applications showcase the capabilities of the evaluation framework and demonstrate its strength in performing basic, extensive evaluation of virtual prototypes.

For future work, the generic nature of the evaluation framework makes it possible to apply it in many different application areas and easily extend it to combine multiple simulation software and integrate multi-objective optimization algorithms.

Contents

List of Figures

List of Tables

Symbols and Formulas

Symbol	Unit	Description	Symbol	Unit	Description
$\mathcal{C}$	-	configuration space	q_i	$^\circ$	waypoint i on trajectory
n	-	number of waypoints	t_f	s	duration of trajectory
v_{max}	$^\circ/s$	joint velocity limits	a_{max}	$^\circ/s^2$	joint acceleration limits
ΔT_i	s	time between q_i, q_{i+1}	t_i^b	s	blend phase at q_i
v_i	$^\circ/s$	joint velocity at q_i	a_i	$^\circ/s^2$	joint acceleration at t_i^b
T_i	s	time at q_i	$q(t)$	$^\circ$	joint trajectory function
$q_i[j]$	$^\circ$	j-th component of q_i	$v_i[j]$	$^\circ/s$	j-th component of v_i
τ	Nm	joint-side torque	q	$^\circ$	joint position
$M(q)$	kgm^2	mass matrix	$\dot{q}$	$^\circ/s$	joint velocity
$C(q,\dot{q})$	$kg\frac{m}{s}$	Coriolis matrix	$\ddot{q}$	$^\circ/s^2$	joint acceleration
$g(q)$	Nm	gravity torque vector			
P_{el}	W	electric power	P_{loss}	W	power loss
P_{mech}	W	mechanical power	P_{motor}	W	motor power
R_T	Ω	terminal resistance	I_{motor}	A	motor current
τ^j	Nm	j-th component of τ	ω^j	$1/s$	j-th component of $\dot{q}$
R	-	gear ratio	η_{gear}	-	gear efficiency
η_{motor}	-	motor efficiency	n_{motor}	$\frac{1}{min}$	motor speed
τ_{motor}	Nm	motor-side torque	k_M	$\frac{Nm}{A}$	torque constant
Δt	s	time step	ΔE_{el}	Ws	electric energy change

Acronyms

1D one-dimensional

2D two-dimensional

3D three-dimensional

AEC architecture, engineering and construction

API application programming interface

APM *Asynchronous Programming Model*

AR augmented reality

BLDC brushless direct current

CAD computer-aided design

CLI *Common Language Infrastructure*

COM center of mass

DOF degree of freedom

EAP *Event-based Asynchronous Pattern*

GUI graphical user interface

IDE integrated development environment

MADM multi-attribute decision making

MCDA multi-criteria decision analysis

MCDM multi-criteria decision making

MODM multi-objective decision making

OS operating system

PLINQ *Parallel Language-Integrated Query*

RNEA *Recursive Newton-Euler Algorithm*

ROS *Robot Operating System*

rpm revolutions per minute

SAINT *Supervised Autonomous Interaction in Unknown Territories*

TAP *Task-based Asynchronous Pattern*

Tcl *Tool command language*

TPL *Task Parallel Library*

TUM *Technical University of Munich*

UI user interface

URDF *Unified Robot Description Format*

VR virtual reality

1 Motivation

Mechatronic systems refer to the family of all industrial products and processes where mechanics, electronics and control are integrated in a synergetic way and interact with each other to support the overall system performance [4]. The market demand of such systems today, with its quickly changing trends, price fluctuations, arise of new technologies and global economic competition, poses new challenges to companies. Simultaneously, the demands for high planning accuracy and quality, precise product design and short manufacturing timeframes increase. [1, 2, 5, 6] To this end, in the past couple of years the traditional product engineering process, in which experts from mechanics, electronics and control only interacted sparely, experienced an enhancement in the form of the integration of modelling and simulation into the development process [7]. Based on this development, during the course of the last two to three decades the concepts of *virtual commissioning* and *virtual prototyping*, arose. The idea of *virtual commissioning* [8, 9] and *virtual prototyping* [2, 3] is to test mechanical and electronic design choices alongside control software in simulation, before or during the development process of the physical product itself. This allows the designers of mechatronic systems to examine whether design specifications are met in the system, while enabling a comprehensive exploration of choices in the system design. Furthermore, *virtual commissioning* and *prototyping* make it possible to gain a more detailed knowledge of the system's quality and performance, be it mechanical-, electronic- or control-specific information, without the explicit need for a physical prototype. Thereby, costs of production and time-to-market can be reduced, thus posing a time- and cost-efficient instrument for companies. At the same time, designing and evaluating mechatronic systems in a simulation-based environment allows companies to refine product design, increase production flexibility and react quicker on changes in market demand. [1, 6, 8, 9]

Being able to test mechanical and electronic design in simulation requires sophisticated and highly accurate virtual models of the underlying mechatronic system. Such models are frequently referred to as *digital twins* [1]. The authors of [7] provide a comprehensive and clear definition of a *digital twin*, by stating that "[t]he vision of the Digital Twin itself refers to a comprehensive physical and functional description of a component, product or system, which includes more or less all information which could be useful in all—the current and subsequent—lifecycle phases" [7]. Following this definition, a *digital twin* is a representation of a mechatronic system as a simulation model, which aims at mirroring mechanical and electronic structure and design, control systems and other properties of its physical image [1]. There exist different, more specific definitions of *digital twins*, building upon the definition stated above. What most of these definitions have in common, is that they refer to a *digital twin* being a virtual model used alongside a physical prototype. Keeping that in mind, these definitions divide a *digital twin* into three main components [1]: the physical product, the virtual simulation model and the data being exchanged between the two. However, a *digital twin* does not necessarily require an already existing physical prototype. A simulation model can also be utilized very early in the engineering process where a physical system is not existing yet [1].

The importance of *digital twins* in engineering companies, both today and in the future, is also shown in a study commissioned by *Forrester Consulting* in October 2019. In this study, 358 architecture, engineering and construction (AEC) and manufacturing companies filled in an online questionnaire regarding real-time three-dimensional (3D) content and visualization [10]. In the context of this study, real-time 3D refers to the process of representing real-world assets, such as mechanical components of products or newly designed buildings, by realistic digital copies, i.e. *digital twins*. The study reveals that, while today the incorporation of real-time 3D visualization into production and design process is relatively low, over half of the participating companies plan on adopting 3D techniques in the upcoming years, indicating a significant rise of industrial utilization of the concept of *digital twins* in the future. Furthermore, companies already using *digital twins* in their business see substantial advantages like cost reduction, sales increase and reduced time-to-market. [10]

To this end, being able to automatically evaluate *digital twins*, independent from the type of mechatronic system being simulated, can be of high value for the industry. In order for production and design experts to be able to examine the simulation model in any way requires them to frame the context in which the system will be tested. This includes defining specifications against which the model will be tested, i.e. defining decision criteria whose evaluation allows the experts to draw conclusions from the current system design. Furthermore, specifying the parameters for which the *digital twin* shall be investigated is an important necessity for extensively testing a mechatronic system in the context of *virtual commissioning* and *virtual prototyping*. This refers to defining mechanical-, electronic- or controller-related properties of the mechatronic system that comprise the space of design choices available at the current phase of system development.

A framework providing tools for intuitively defining this evaluation space directly from the *digital twin*, efficiently testing the simulation model in this context and visualizing the test results in an informative manner can help companies to increase the value gained from implementing *virtual commissioning* or *virtual prototyping* in their business, by enabling experts to test design choices of all kinds in a simple, efficient and in-depth manner, independent from the current phase of the engineering process. Additionally, simulation of system models in general, and in combination with such an evaluation framework, makes it possible for domain experts (electricians, mechanics, embedded software developers) to test their sub-component concepts in the whole system. This enables them to better grasp the impact of design modifications on the overall system performance and behaviour.
It is for these reasons, that this thesis deals with exactly such kind of a framework for simulation-based evaluation of mechatronic systems.

2 Theoretical Context and State of the Art

This chapter summarizes theoretical background information and state of the art techniques, approaches and software environments applied in the industry in the context of this thesis. This includes state of the art approaches to *virtual prototyping* and model-based design, techniques implemented and realized in the industry for simulating virtual prototypes, as well as a summary on state of the art methods in the field of multi-criteria optimization.

Virtual Prototyping and Simulation-Based Design Generally, a product development scenario comprises the following steps, usually arranged in a v-shape, thus commonly referred to as the *V-Model* of systems engineering [11, 12]:

1. Analysis of system requirements
2. System design
   (a) Modelling and model analysis continuously support the whole further development process
3. Domain-specific design
   - Mechanical engineering
   - Electrical engineering
   - Information technology
4. System integration
   (a) During system integration, iteratively verifying and validating system requirements and assuring system properties in the system design
5. Product commissioning

Any issues identified during or in between any of the above development stages requires to revise design specifications and system properties and repeat all, or a subset of the steps mentioned above (see step $4\ a)$ in the description of the *V-Model* above. Usually, prototypes provide means for product designers to answer questions that arise during the product development cycle [3]. Advancements in computer technology and digitalization lead to an increase in the integration of virtual prototypes in the development process, thereby complementing, and even fully replacing physical prototypes. The most important advantages of virtual over physical prototypes are the increase in design and production flexibility and the reduced time and cost for creating and evaluating the prototype, thereby reducing the time-to-market. Additionally, *virtual prototyping* helps to reduce the number of iterations required in the development cycle, and aims at identifying errors and potentials for improvement as early as possible. [3]
Virtual prototypes are generally classified according to their modelling purpose in the context of the product development cycle. Five partially overlapping classes of virtual

prototypes can be distinguished, with each class fulfilling one of the following purposes [3]:

- **Visualization**: Such virtual models can be used to examine product appearance and form.
- **Mechanical fit**: Some virtual prototypes are used to evaluate the state of the products assembly and the fit of mechanical components in the whole product.
- **Testing of functions and performance**: In *virtual prototyping*, testing and verifying functions and properties of the product, as well as observing the prototype's performance in different scenarios, plays a central role.
- **Manufacturing evaluation**: These prototypes allow experts of different engineering disciplines to explore the current state of the product and collaborate on further design and manufacturing.
- **Human factors analysis**: Special kinds of virtual prototypes make it possible to evaluate safety and ergonomics of the product and test how intuitively user interaction is realised in the current product design.

Determining the class of virtual prototype suitable for the current stage of the development cycle is a necessity for designers to choose a proper modelling approach to realize the virtual prototype.

Simulation Technology and Software for Virtual Prototypes There exist various approaches and software environments for simulating mechatronic systems using *digital twins*. The software that is best suited for the respective application depends on the type of system being simulated, i.e. the type of virtual prototype according to their aforementioned classification, as well as on the context and objectives of the simulation, be it of mechanical-, electronic- or control-related nature.
Discrete event system simulation [13] is a simulation technology, in which the simulation progresses by processing a sequence of events. The model of a system in such a simulation can be represented by finite automaton, a list of current and future events in the simulation, as well as specifications of event timings and the logic behind the execution of events [13]. Discrete event system simulation is a technology often used in the context of computer networks. *ns-3* [14] and *OMNet++* [15] are two examples of software tools implementing discrete event system simulation techniques.
MATLAB/Simulink [16] is a model development environment for equation-based/signal oriented simulation approaches. It is based on the programming language *MATLAB* and realizes modelling and simulation in the form of graphic blocks. In the context of mechatronic systems, the technique of signal oriented simulation in *MATLAB/Simulink* is mainly used for one-dimensional (1D) simulations of virtual models of control systems, but has difficulties when simulating 3D mechanics and collision detection.
Game engines are another toolkit for modelling and simulating mechatronic systems. Game engines like *Unreal Engine* [17] and *Unity* [18] experience a great increase in their application in industries other than entertainment. They can be used as model development environments and for 3D real-time simulation methods of systems in

a physically realistic environment. Thus, they pose simulation environments for 3D mechanics, with libraries and extension possibilities in various other relevant physical domains.
Dassault Systèmes' CATIA [19] and *DELMIA* [20] software solutions provide toolkits for performing ergonomics analysis with virtual prototypes. These software environments provide various multi-domain simulation possibilities and support multiple system modelling approaches and simulation techniques. Thereby, these environments allow users to simulate a variety of different virtual prototypes within one development environment. Commercial packages, such as *ADAMS* [21], *Altair*'s *HyperWorks* [22] and *MATLAB*'s *Simscape Multibody* [23] are great tools for modelling and simulating the mechanics of multi-body systems, ranging from mechanical dimensioning to vibration studies.
Modular physical modelling is a modelling method originating in the 1970's and has evolved into numerous different modelling languages and simulation environments. [2] *Modelica* [24] is probably the most famous modelling language in this context. In modular physical modelling, complex physical systems are formulated as a set of ordinary and algebraic differential equations. Using modular physical modelling is especially useful when interdisciplinary characteristics of mechatronic systems have to be addressed in simulation, since it allows to combine thermal, electronic, hydraulic, control and other system components into one simulation model.

Multi-Criteria Decision Making Multi-criteria decision making (MCDM), also called multi-criteria decision analysis (MCDA), refers to situations in which decisions are influenced by multiple, often conflicting, criteria [25, 26]. MCDM has experienced a drastic increase in application in the past few years, and is seen as one of the most well known disciplines of general decision making. Generally, in MCDM, the actions available to the decision maker, i.e. the different choices of parametrization of the underlying decision problem, are referred to as alternatives. Furthermore, the criteria, i.e. the attributes or decision criteria, of the problem represent the perspectives and dimensions from which the alternatives of the problem are examined. Usually, this means that the decision problem is defined by alternatives representing different problem scenarios, for each of which the attributes of the MCDM are evaluated. [25, 26] In general, decision making is very problem-specific. Nonetheless, optimization in the context of MCDM problems comprises the following characteristics [25]:

- **Multiple attributes:** Every MCDM problem consists of multiple attributes. These are problem-specific and usually defined by the decision maker.
- **Conflicting attributes:** Usually, some, or all, attributes of the underlying problem conflict with each other. This means that there is often a trade off between different dimensions the decision maker has to consider.
- **Incomparable units:** Since each attribute of the MCDM problem represents a different dimension, their units differ and cannot be related in a straightforward manner.
- **Solution:** In the end, the decision maker aims at finding the best-possible, problem-specific compromise solution for all considered decision criteria. This

often requires the decision maker to identify *Pareto* fronts and find the *Pareto*-optimal alternative therein. Here, *Pareto*-optimal refers to the alternative leading to the best possible result in all considered decision criteria, i.e. the optimal trade off between all attributes.

Following many authors, e.g. [25, 26], MCDM can be divided into two main disciplines: (1) multi-objective decision making (MODM) applied for decision problems with continuous decision criteria, such as mathematical objective functions; (2) multi-attribute decision making (MADM) focuses on discrete decision spaces with a finite set of alternatives.
For both these disciplines of MCDM, there exist multiple distinct methods, with different application-related advantages and disadvantages. In [27], the most common and state of the art methods for both MODM and MADM, as well as several other approaches are summarized and reviewed. These methods include *Multi-Attribute Utility Theory, Analytic Hierarchy Process, Fuzzy Set Theory, Case-based Reasoning, ELECTRE* and *PROMETHEE*, just to name a few. Additionally, [25, 26] provide the basics and details for each of these methods. Without going into detail, the ultimate goal of every MCDM method is to support the decision maker in defining the decision problem's alternatives and attribute space. Besides that, such methods help the decision maker to find criteria to weight, rank and evaluate the problem's attributes and to evaluate and identify conflicting alternatives and decision criteria. An important concept therein is meaningful treatment and visualization of the data. [25, 26, 27]

Visualization Techniques in MCDM Meaningfully treating and visualizing the data of any MCDM problem plays a major role in decision making techniques and is an important feature of most state of the art MCDM software solutions [25, 26, 27, 28, 29, 30]. Amongst many others, current state of the art software includes *1000Minds, Analytica, Decerns, Diviz*, just to name a few. Concerning visualization, the complexity of an MCDM problem is determined by the number of alternatives and the number of attributes, i.e. decision criteria or objectives. In general, any graphical representation of a problem's decision space should aim at providing the information of the data in an informative and clear way, thus enabling the decision maker to intuitively explore the problem data, evaluate alternatives and identify objective trade offs. [29, 30] The technique used to visualize the data is thereby depending on the main purpose of the graphical representation. The following list presents some common visualization techniques and the situations in which they might be most appropriate [29]:

- **Single alternative snapshot:** When only looking at single alternatives, i.e. a snapshot of the values of all decision criteria for a specific solution, bar charts are usually the way to go. In this context, a snapshot refers to a user-specified selection, or subset of decision alternatives and decision criteria. Another option is the so-called *Chernoff* face, which represents a snapshot of an alternative by a human face, comprised of primitive geometric shapes, whose properties are determined by the values of the decision criteria. [31]
- **Evaluating multiple alternatives:** If the set of alternatives is finite and very small, again a bar chart can be used to visualize the attributes for each alternative in the

decision space, with one bar for each alternative and attribute. If there are too many alternatives, however, or the decision maker wants to compare alternatives directly, bar charts are not sufficient. Alternatively, line graphs are commonly used to visualize decision problems with small sets of data. Each line usually represents an attribute, and its values for the corresponding alternatives on the x-axis are connected by straight lines. This allows the decision maker to directly compare alternatives. Switching this representation, such that objectives are organized on the x-axis and each alternative is represented by a line, results in a so-called score profile.

In [28], over 20 different MCDM software solutions are reviewed, evaluated and compared. One important aspect in the evaluation of these software tools was the technique used to represent the decision space. Besides the aforementioned methods, some of these tools utilize more advanced graph representations, such as radar, spider, hierarchical tornado and bubble graphs, and even combine multiple visualization techniques to enable the decision maker to grasp the information of the decision problem more easily. What most of these software solutions have in common, is their interactive interface. Almost all graphical representations are extended by interactive user interface (UI) elements, that allow the decision maker to modify the visualization, rank and weigh decision criteria, select preferred alternatives or focus on explicitly selected decision criteria and their relations.

Unity Simulation In September 2019, *Unity Technologies*, creator of the aforementioned *Unity* game engine [18] announced *Unity Simulation*. *Unity Simulation* is a product, implementing the idea of running multiple simulations of a *Unity* project at scale, similar to the idea of the *Execution and Evaluation* component of the evaluation framework presented in this thesis (see Section 6.3). [32] With *Unity Simulation*, *Unity Technologies* introduces a cloud-based solution that allows users to run millions of simulations at the same time. As mentioned similarly in Chapter 1, *Unity Simulation* can be applied to *virtual prototyping* approaches, since it enables companies to test multiple scenarios in simulation, prior to the costly manufacturing of physical products. The primary idea of *Unity Simulation*—according to *Unity Technologies*—is to apply the new solution to computer vision, AI and machine learning problems in automotive, gaming and robotics industries. In this context, *Unity Simulation* simplifies and accelerates the training of machine learning algorithms, the testing of multiple scenarios in simulation, and the validation of product performance before going into production. [32]

3 Objectives

The main objective of this thesis is to develop and implement an independent evaluation framework capable of evaluating simulation models of mechatronic systems. The idea is to provide an evaluation framework as a set of standalone tools that enable the user to easily evaluate simulation models in a user-defined evaluation space. In this context, the evaluation framework must be capable of providing means to define the parameters and decision criteria for which the simulation model should be evaluated, automatically and efficiently carry out multiple sequences of simulations of the underlying model and visualize the results from these simulations in a meaningful and informative manner. Visualizing the point clouds resulting from the decision criteria being evaluated for a large sequence of simulations should enable the user to obtain an overview of the multi-dimensional evaluation space and the corresponding scope of designing and planning the mechatronic system. Furthermore, another objective of the thesis is to implement two exemplary applications demonstrating the capabilities of the evaluation framework. The framework shall then be used to evaluate the simulation model of both applications and identify optimal parameters in the context of the two scenarios.

Going into more detail, the fundamental requirement of this thesis is to develop the evaluation framework building upon the game engine *Unity*, and implementing the functionality of the framework in the *C#* programming language. Furthermore, one central demand during the implementation of the evaluation framework is to increase the framework's efficiency and execution speed by concentrating on parallelizing the *C#* code, in order to be able to execute simulations in parallel.
In reference to the aforementioned exemplary applications demonstrated and evaluated in this thesis, the following details must be considered. The first scenario reproduces a robotic picking process, i.e. robotic motion along different trajectories, in simulation. Here, the goal is to use the evaluation framework to identifying application-specific, optimal geometric and dynamic parameters of the simulated robot arm. The objective of the second scenario is to determine the optimal position of the *Franka Emika Panda* robot arm inside an industrial plant in the context of a bin picking process for the *Supervised Autonomous Interaction in Unknown Territories* (*SAINT*) project, in which the *Chair of Astronautics* and *Chair of Applied Mechanics* at *Technical University of Munich* (*TUM*) participate. Initially, the idea was to use the evaluation framework to evaluate the two applications and discuss their results, while falling back on already existing simulation models and environments. However, during the work on the two scenarios, an additional focus was laid upon modelling them in physically realistic simulations in *Unity*.

4 Approach

Implementing the objectives explained in Chapter 3 and satisfying all of the specified requirements was achieved through multiple distinct, but closely related steps. Obviously, the initial focus of the work was laid upon building a first, fully-functional version of an evaluation framework, already containing all main components required to evaluate a mechatronic system in a simulation-based environment. On completion of this first version, a simple simulation acting as a test environment for the evaluation framework was developed and used to debug errors in the code. Subsequently, the focus of the work shifted towards modelling the two aforementioned scenarios and representing them in a *Unity* simulation. Continuously applying the evaluation framework on these two applications pointed out missing features in the current state of the evaluation framework that prove helpful for the process of evaluating a simulation model and improve the framework's usability, efficiency and robustness. To this end, the evaluation framework was extended and improved, while, in parallel, the two applications were finalised and, in the end, evaluated with the completed evaluation framework. One major focus of implementing the two application examples and extend the evaluation framework was the integration of the *Robot Operating System* (*ROS*) into the framework's architecture. In this context, the evaluation framework alongside *ROS* was used to simulate realistic, complex robot trajectories.

Designing and implementing the core components of the evaluation framework was achieved by following, to some extend, a top-down approach. Initially, the overall idea and concept of evaluating a mechatronic system in simulation was framed and then divided into several distinct components, by separating independent functions the evaluation framework has to carry out. With this main concept and the different features of the evaluation framework identified, the design and functionality of each separate part of the framework could be delineated from the other components and formulated as part of the whole architecture in more detail. Consequently, all components of the framework could be implemented separately. In fact, the order in which each of the framework's components was implemented corresponds to the earlier identified sequence of steps required to evaluate a simulation model of a mechatronic system.

The content of this thesis is organized similar to the development approach outlined above. Chapter 5 introduces the basic concepts of all development environments utilized during the implementation of the evaluation framework. Subsequently, Chapter 6 presents details regarding architecture, design choices, functionality and code implementations of the evaluation framework and its components. With the evaluation framework being completed, Chapter 7 dives into the two scenarios implemented in this work. The chapter presents the simulation model of the mechatronic system being evaluated, explains the two applications in detail and showcases how the evaluation framework can be utilized in this context. Afterwards Chapters 8 and 9 present and discuss the evaluation results of the two exemplary applications. Chapter 10 concludes

the thesis and summarizes the results obtained during this work. Lastly, Chapter 11 puts the results into perspective and provides ideas for future work with and extensions of the evaluation framework.

5 Development Environments

The evaluation framework developed in this thesis is based on several existing development environments, code libraries and toolkits. The focus of this chapter is to give a brief introduction into the main concepts behind these software tools used during development. Section 5.1 explains the basics of *Unity*, originally a game engine and today widely used in other industries as well. *Microsoft .NET*, a cross-platform software development platform, and its fundamental components in developing *.NET* solutions are summarized in Section 5.2. Subsequently, Section 5.3 dives into the basics of *tkinter*, a *Python* library for creating interactive UIs. These three development environments are the basis of the three main components of the evaluation framework developed during this work, respectively. In addition to that, Section 5.4 and 5.5 provide information about *ROS* and the open-source library *ROS#*, which build the software foundation for applications of the developed evaluation framework discussed in Chapter 7.
All software and development environment versions used to implement and test the evaluation framework of this thesis are listed in Appendix A.

5.1 Unity

Approximately 50 % of all computer, console and mobile games are powered by *Unity*, thus reaching nearly three billion devices worldwide and making *Unity* primarily known to be the world's leading game engine [33]. *Unity* is a software environment that enables users to create two-dimensional (2D) and 3D simulations, as well as augmented reality (AR) and virtual reality (VR) solutions [34]. As stated by *Unity*'s former CEO David Helgason the company's mission is to "[...] democratize game development" [35]. While initially being released for Mac OS X in 2005, *Unity* is now available on 30 different platforms, including Windows, iOS and Android. By providing tools that empower developers to create visual content while maximizing ease-of-use, *Unity* opened up the broad field of game development to both experts and beginners. [36]
Today, *Unity* plays an important role in cross-industry software development and is widely used for projects that go beyond entertainment applications. In 2014, *Unity* was already applied in industries like architecture and art to create photo-realistic, interactive 3D visualizations and experiences. *Unity* was also established as an important tool in medical education and training, enabling students and staff to run through virtual, realistic medical situations and practice without the danger of causing any harm. AR solutions for interactive virtual tours through museums and art galleries constitute another part of industry in which *Unity* took root. [37] In 2018, *Unity* broadened its target audience by officially introducing development bundles and software extensions for, amongst others, automotive, film, construction and engineering [38, 39, 40, 41, 42]. The ten leading car manufacturers already use *Unity*'s real-time 3D platform for several crucial operations in the design, construction and marketing of cars, thereby showing *Unity*'s success in industrial sectors [43].

Unity can be divided into two main parts: the *Unity* editor itself alongside its main

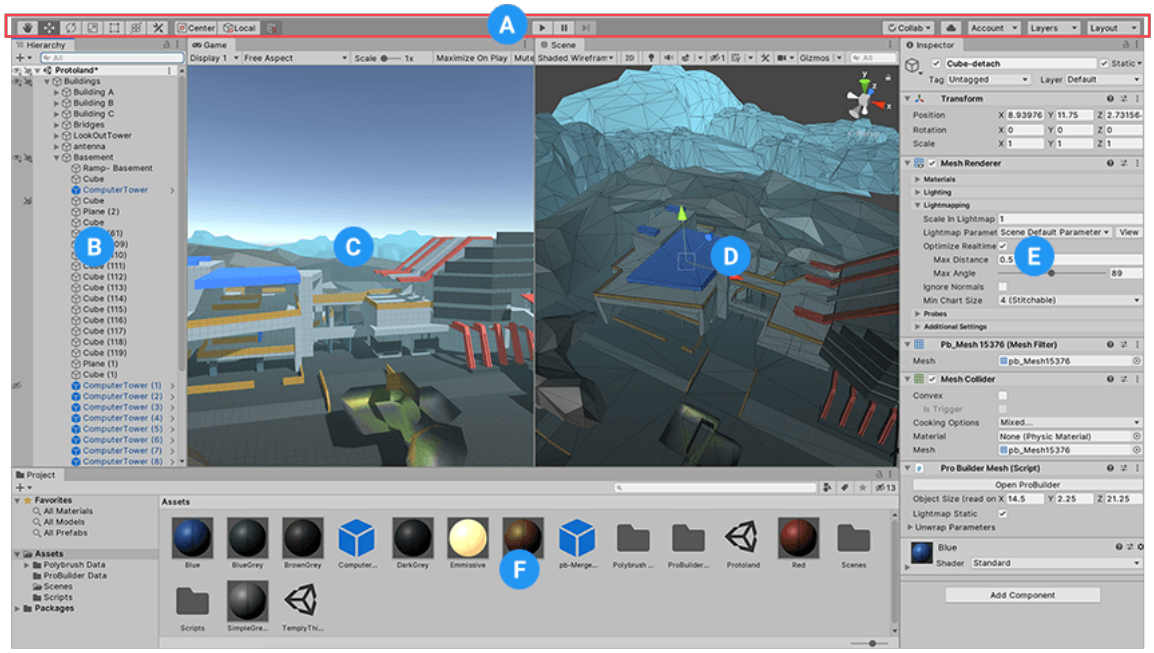

Fig. 5–1: *Unity*'s editor window. (A): Toolbar, (B): Hierarchy window, (C): Game view, (D): Scene view, (E): Inspector window, (F): Project window. (Source: [44])

components and the *Unity* scripting application programming interface (API). The basic concepts of those two parts when creating *Unity* solutions will be briefly introduced in the following sections.

5.1.1 Unity Editor

The editor window is the main interface *Unity* users work with during development. Figure 5–1 depicts this window in its general form, with distinct sub-windows, tagged with different letters. It is important to note that the layout of the editor window can be fully customized by the user. All windows can be rearranged, docked, resized, duplicated and removed, while additional windows and functionalities can be added to exactly match the creator's needs. In *Unity* objects present in the scene are referred to as *GameObjects* (see below). [44] The components highlighted in Figure 5–1 compose the most basic windows required for almost any solution created with *Unity* and serve the following purposes [44]:

(A) **Toolbar:** The toolbar is the only window inside the editor that cannot be rearranged or changed. It provides the basic features for changing the editor window layout, manipulating *GameObjects* and navigating in the scene view. Buttons for starting, stopping or pausing the execution of the project are also part of the toolbar. Starting the application is referred to as entering *Play Mode*, whereas its counterpart is denoted as *Edit Mode*.

(B) **Hierarchy window:** All *GameObjects* existing in the application scene are listed in the hierarchy window. The default order of *GameObjects* inside the hierarchy window results from their order of creation and can be changed via drag-and-drop. The concept of parenting is solved by introducing a tree-like structure for parent

and child *GameObjects*, with the parent *GameObject* being the topmost layer and all children hierarchically grouped underneath. In order to increase the clarity of the scene view and maximize ease-of-use when manipulating the scene, the visibility of *GameObjects* can be modified in the hierarchy window.

(C) **Game view:** Each *Unity* project contains one *Unity* camera by default. The game view is rendered from exactly this (and all other present) camera(s) and represents the view of the player using the application. Using the play, pause and stop buttons of the toolbar allows the creator to switch from developing the project to the player's view of the current state of the application.

(D) **Scene view:** Every virtual world created in *Unity* is displayed in the so-called scene view. Using mouse and keyboard, the user can navigate in the scene and manipulate cameras, lights and all other objects per drag-and-drop.

(E) **Inspector window:** Selecting a *GameObject* displays its associated information in the inspector window. Besides *GameObjects*, basically all project *assets* can be selected and inspected in the inspector window. Additionally, the window allows the user to modify the properties of the selected *asset* or *GameObject*.

(F) **Project window:** *Unity*'s project window contains information about the organization of the entire project. The window lists all files present in the project, arranged in folders in a tree-like structure. In *Unity*, all items existing as files in the project window, such as textures, meshes and scripts, are referred to as *assets*, which is why the top folder in the project window containing them is always named *Assets*. A search bar enables the user to navigate inside the project and quickly find and/or reorder *assets* and other project-related files. To the right of the hierarchical depiction of the project content, another window exists showing a visual representation of the files inside the currently selected folder.

By default another tab is activated besides the project window enabling the *Unity* console. The console window displays errors and warnings generated by *Unity*. The console is furthermore frequently used as a debugging tool for developers, since it allows them to print specific messages at any point in their application. [45]

GameObjects and Components As already mentioned, all scene objects in *Unity* are called *GameObjects*. *GameObjects*, however, are not *Unity assets*, since they are not represented by files stored in the *assets* folder, but only exist in the scene and hierarchy window. Every *GameObject* consists of *Components* describing a functional part of the object. [46] Thus, *GameObjects* are in some sense containers for *Unity*'s *Components*. Without any *Components*, a *GameObject* is basically just a visual representation of an object, without any functionality. *Unity* supports a variety of built-in *Components* that can be used, while additional customized *Components* can be added by the user [46]. Each *Component* itself comprises multiple *Properties* determining the functional characteristics of the respective *Component*. By default, any *GameObject* has one *Transform Component*, representing position, orientation and scale of the object within the scene. [46] Looking at the concrete example of simulating a person in *Unity*, the person itself could be represented by one or several connected *GameObjects*, a

Component could be the hat worn by the person, and the hat's *Property* would be its material and color. In this example, the hat's material *Property* would usually be stored as an *asset* in the project window.
The main *Components* relevant to this thesis are summarized below:

- **Colliders:** Physical collision between *GameObjects* in *Unity* require the objects to have a *Collider Component*. *Colliders* define the collision shape of a *GameObject* that must not necessarily match the shape of the object's mesh (a *mesh* is the main graphics primitive in *Unity*). A simple *Collider* can be of primitive shape, such as *Box, Sphere* and *Capsule Colliders*, making its usage in gameplay very efficient. More complex collision shapes can be represented by *Compound Colliders*, combining multiple primitive *Colliders*. Even more complex shapes have to be realized by *Mesh Colliders*, which match the shape of the *GameObject* exactly, thus providing accurate collision control. However, these Mesh *Colliders* are processor-intensive and hence slow-down the performance of the application drastically. [47]

- **Rigidbody:** *GameObjects* can only behave in a physically realistic way if they have a *Rigidbody Component* attached. A *Rigidbody* allows any *GameObject* to receive forces and torques, react to gravitational influences or experience collisions with other objects, in order to be able to realistically accelerate and move in the scene. The *GameObject*'s mass, as well as drag and angular drag coefficients can be specified as *Properties* of the *Rigidbody Component*. Besides that, the body's inertia is also managed by the *Rigidbody Component*. *Unity* provides real-world physical behaviour through the *NVIDIA PhysX* physics engine. [48]

- **HingeJoint:** Connecting two *Rigidbodies* in *Unity* can be achieved in multiple ways. Two objects can be connected by *FixedJoints, HingeJoints, ConfigurableJoints* and others. *Unity*'s *HingeJoint Component* provides a specifically convenient way to simulate a hinge connecting two bodies, as it is often required in robotic applications when simulating revolute joints with one angular degree of freedom (DOF). Table 5–1 summarizes the most important *Properties* of a *HingeJoint*. [49]

- **Scripting:** Scripting is the process of creating customized *Components* through code. In *Unity*, *C#*-scripts can be attached to *GameObjects* just like any other built-in *Component*, thus appearing in the *GameObject*'s inspector window. In Section 5.1.2 detailed information on *Unity*'s Scripting API is provided. [50]

5.1.2 Unity Scripting API

Creating customized *Components* in *Unity* is done via scripts. A script is a piece of code that allows the developer to trigger events, respond to input from the player, modify *Properties* of built-in *Components*, control the behaviour of objects in the scene and even modify the *Unity* editor itself. *C#* is the natively supported programming language and *Visual Studio* is the default editor used to implement scripts, that can be automatically downloaded alongside *Unity*. However, many other languages and editors can be used within *Unity*, as long as they fulfil some specific requirements. [51]

Scripts, which are one particular type of *Unity assets*, can generally be implemented in any code editor. However, creating and opening a script in *Unity* opens *Unity*'s default *Visual Studio* code editor. Every script implements its own class. To provide functionalities in *Play Mode*, a script normally inherits from *Unity*'s built-in *MonoBehaviour* class, thus connecting the scripted *Component* to *Unity*'s internal workings. As a result, a script inheriting from *MonoBehaviour* is not activated until it is attached to a *GameObject*, which can be achieved the same way as any other *Component* is attached. *Properties* of built-in *Components* are now represented by variables in the script. Scripts attached to a *GameObject* furthermore enable the developer to modify other attached *Components* within the script at runtime, in order to e.g. move the object or change its *Rigidbody*'s mass, as well as spawn and destroy other objects inside the scene. [51]

Execution Flow Specific event functions included in the *MonoBehaviour* class control the execution of scripts before, during and after *Play Mode*, rather than just running the code line-by-line in an endless loop. If a script *Component* needs to react to any of those events, a function with the corresponding event-name has to be defined in code, thus executing the body of the function whenever the event is triggered. By default, *MonoBehaviour* scripts are only executed in *Play Mode*. However, some specific events are triggered even while working in the editor window. Figure 5–2 displays the order of execution of *Unity*'s most common event functions listed below [52]:

- **Initialization events:** *Unity*'s *Awake* function is the first function called when entering *Play Mode*, being triggered on scene load. Straight after the *Awake* function, a *Start* function is executed. Both of these functions can be used for initialization purposes, since they are executed before *Unity* enters the main execution loop.
- **Update events:** These are events that are repeatedly called during *Play Mode* of the game. A game is simply a sequence of animation frames and in *Unity*,

Tab. 5–1: Main *Properties* of *Unity*'s *HingeJoint* (Source: [49])

Property	Function
Connected Body	*Rigidbody* of the *GameObject* the object is connected to.
Anchor	Position of the axis of rotation.
Axis	Axis of rotation.
Limits	Minimum and maximum rotation angle of the joint.
Spring	A spring stretched between the joint's current and target angle. The specified spring force pushes the joint towards its target angle, while its damping force dampens the angular velocity during the movement.
Motor	A simulated motor moving the joint. The motor tries to reach a given target velocity without exceeding its specified maximum torque.

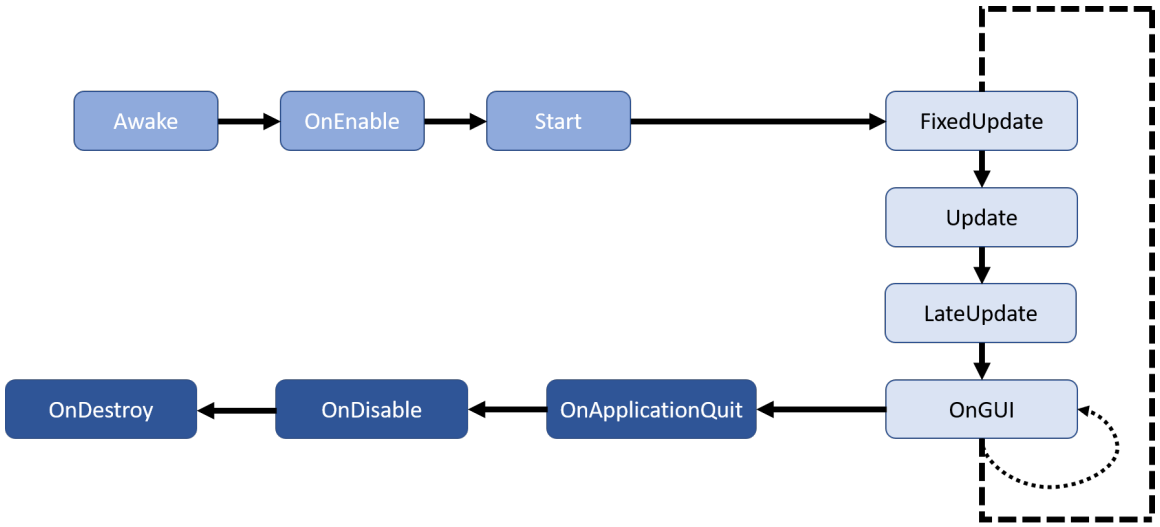

Fig. 5–2: Execution order of *Unity*'s event functions. (Adapted from [53])

the *Update* function is called just before the next animation frame is rendered. Hence, changing the position or behaviour of objects is mostly done inside the Update function. It is important to note that the Update function is not called in constant time-intervals, but rather depends on the varying frame rate of the game and the hardware capabilities. Therefore, *Unity* provides a *FixedUpdate* function. The event triggering the execution of this function is elicited in constant, user-specified time intervals right before the next physics engine update. The physics engine requires a constant time step. Thus, all physics-related calculations must be performed inside the *FixedUpdate* function. Additionally, the *LateUpdate* function is called after the Update and *FixedUpdate* functions have been called on all *GameObjects*.

- **GUI events:** The so-called *OnGUI* function is called periodically and should contain any code that needs to react on changes and clicks on GUI controls inside the *Unity* editor windows. It is also used to implement code responsible for changing the visual appearance of editor windows. Other GUI event-functions include *OnEnable* and *OnDisable*, called whenever a *GameObject* becomes enabled and active or disabled and inactive, respectively. In contrast to the update event functions, GUI events are triggered even outside the *Play Mode*, thus reacting on changes in *Unity*'s GUI. Usage of the *OnGUI* function is explained in more detail below.
- **Decommissioning events:** When the application is quit or, in the editor, *Play Mode* is quit, or when a *GameObject* is destroyed, the *OnApplicationQuit* and *OnDestroy* functions are called, respectively.

Editor Scripting When the developed application is not in *Play Mode*, *Unity* is in *Edit* Mode. Scripts inheriting from the previously described *MonoBehaviour* class are normally not executed in *Edit Mode*. However, there is a way in *Unity* to implement code that executes independently from *Play Mode* and hence affects the *Unity* editor windows.

As already mentioned, the complete UI of *Unity*'s editor with all of its windows and tabs can be customized and rearranged to match the developer's needs. Besides that, it is possible to create new editor windows through scripts. Basis of this editor scripting is to derive the script-class from *Unity*'s *EditorWindow* class instead of the default *MonoBehaviour* (which is used to attach scripts as *Components* to *GameObjects*). All code defining the appearance and behaviour of the newly created window is packed into the body of the aforementioned *OnGUI* event function. Code inside this *OnGUI* function executes periodically, whether *Unity* is in *Play Mode* or not. [54]
Further customization of *Unity*'s editor can be achieved through *CustomEditors*. By default, any *Component* attached to an object has a standard appearance in *Unity*'s inspector window when selecting the corresponding object. Hence, all *Properties* of each *Component* can be edited in the inspector. However, it might be convenient to customize the appearance of frequently re-used *Components* to speed up development. A *CustomEditor* achieves exactly that by replacing the default interface of *Unity*'s inspector for a specified *Component*. Implementing such a *CustomEditor* is carried out by inheriting the script class from *Unity*'s *Editor* class and adding a reference to the type of *Component* the *CustomEditor* is modifying. Similar to the *OnGUI* function mentioned above, code to customize a *Component*'s visual behaviour through *CustomEditors* has to be placed into the so-called *OnInspectorGUI* function. [54]

Scriptable Objects As opposed to *GameObjects*, *ScriptableObjects* are *assets* and hence are stored in *Unity*'s *assets* folder and cannot be attached to *GameObjects* as a customized *Component*. The primary use case for *ScriptableObjects* is the storage of large amounts of data. *ScriptableObjects* are usually used to either store data that is shared by various objects and used at runtime, or to save data accumulating in *Edit Mode*. This can be particularly useful when information about application, scene or specific objects can be gathered during *Edit Mode* without requiring the application to actually execute. [55] Specific utilization of *ScriptableObjects* in the context of this thesis will be discussed in Section 6.2.

5.2 Microsoft .NET

Microsoft .NET (pronounced *dotnet*) is a an open source, cross-platform software tool for developers to create and build multiple distinct types of programs and applications [56]. It is a developer platform that allows for the usage of different programming languages alongside libraries that provide the necessary functionality. *Microsoft .NET* currently supports *C#*, *Visual Basic* and *F#* as programming languages, as well as three different libraries called *platforms* themselves [56]: (1) *.NET Core* as cross-platform development tool running on Windows, MacOS and Linux; (2) *.NET Framework* used to, among others, create websites, running solely on Windows; (3) *Mono* as *.NET* platform for mobile applications. All of these platforms share a set of standard libraries combined in the so-called *.NET Standard* platform. By default, *.NET* code is created using *Microsoft*'s *Visual Studio* integrated development environment (IDE). All *.NET* platforms implement the *Common Language Infrastructure* (*CLI*), which is a set of standardized specifications that allow the developer to choose between the aforementioned programming languages when developing *.NET* solutions. Furthermore,

all supported *.NET* programming languages are type-safe, meaning that each object in code is an instance of a specific type. Objects of each type can only call methods associated with their type. Type definitions in *.NET* are object-oriented and thus work with the concept of base classes and class inheritance. Each type in *.NET* languages inherits from the overall base class *object*. [56] *Microsoft*'s *.NET* Documentation [57] provides useful information on the main concepts of *.NET* implementations. Important concepts in the context of this thesis are summarized in more detail below.

5.2.1 Asynchronous and Parallel Programming

Writing asynchronous or parallel code in *.NET* can be achieved through a wide range of methods and libraries. This section deals with the most common and best-practice approaches to achieve asynchronous and/or parallel program execution. In general, asynchronous programming is required to keep applications responsive for the user while - asynchronously - the program is processing in the background to e.g. download files from a server. Parallelization aims at speeding up the the execution of a program by efficiently using different threads to fully exploit the computer's hardware resources.

Asynchronous Code in .NET *.NET* provides three templates for implementing asynchronous code: *Task-based Asynchronous Pattern* (*TAP*), *Event-based Asynchronous Pattern* (*EAP*) and *Asynchronous Programming Model* (*APM*). All three of them will be briefly introduced, even though *TAP* is clearly stated to be best-practice when it comes to asynchronous programming and should be preferred over *EAP* and *APM* in most use cases. Among these methods, *TAP* is the most high-level, easy-to-use and scalable method for creating asynchronous code, without explicitly having to deal with threading operations and memory locks. [58]

- ***TAP*:** This pattern only uses one method for initiating, monitoring and disposing an asynchronous operation and is the preferred method for most *.NET* applications. *TAP* is based on the *Task*-namespace of *.NET*, that is implemented in the *Task Parallel Library* (*TPL*), and *C#*'s *async* and *await* keywords. *TAP* was first available in *.NET Framework 4*. [58]
- ***EAP*:** As opposed to *TAP*, *EAP* requires one method performing the asynchronous operation alongside one event handler or delegate, which triggers on completion of the asynchronous operation. Hence, it requires already more lines of code in comparison to *.NET*'s *TAP* method. According to [58], *EAP* was introduced in *.NET Framework 2*.
- ***APM*:** The third option is *APM*, which requires the definition of two separate methods to *Begin* and *End* the asynchronous operation. [58]

Parallelizing .NET Solutions *Microsoft .NET*'s *TPL* is the library behind the *Task*-namespace used by *TAP* to create asynchronous code. Besides the *TPL*, *Parallel Language-Integrated Query* (*PLINQ*) is a common toolkit for parallelizing *.NET* code. Both *TPL* and *PLINQ* serve different purposes. [59] Since parallelization of the code implemented in this thesis solely builds upon *TPL*, it will be briefly introduced below.

The main benefit of creating parallel code using *TPL* is that the developer does not have to deal with any low-level parallelization concepts and can focus solely on implementing the actual program, while still optimizing code performance. *TPL* automatically adjusts the degree of parallelism to fully exploit the computer's CPU resources. Besides that, *TPL* plans the distribution of threads, supports cancellation handling and deals with many other low-level concepts. However, parallelizing code, especially when dealing with short code sections requiring only very few processing power, can also affect code performance negatively, since the additional effort through parallelization slows down the performance of the program. Hence, the developers have to chose carefully, when and to what extent they fall back to parallel programming concepts. [60]
.NET further specifies various data structures useful for improving and stabilizing parallel code. These structures can be used in combination with any available parallelization library. Nonetheless, *TAP*, or rather *TPL*, remain best-practice when dealing with parallel or asynchronous program execution in *.NET*. The data structures provided by *.NET* apply to the following areas of parallel programming [61]:

- **Concurrent collection classes:** This namespace provides classes handling thread-safety when dealing with collections, such as arrays or lists, being modified concurrently from within different threads. Adding- and removing-operations do not require the developer to implement locks specifically. By using the concurrent collection classes, *.NET* takes care of avoiding locks as often as possible, while still providing fine-grained locking whenever necessary.
- **Synchronization primitives:** These classes implement different mechanisms that allow for efficient, high-performance synchronization methods. *.NET* provides various synchronization primitives, enabling synchronized, concurrent access of shared resources, as well as coordination of thread execution and completion and restriction of the degree of parallelization.
- **Lazy initialization:** This term refers to the concept of allocating memory for an object only when it is needed, that is, accessed, in code. In doing so, memory allocation for large objects can be spread evenly across the full execution of a program and, in some cases, can lead to significantly increased performance.

5.3 Tcl, Tk, tkinter

When aiming to develop a desktop UI application, one easily stumbles across *Tool command language* (*Tcl*) and *Tk*. *Tcl* is a dynamic programming language developed in the 1980's by John Ousterhout, a computer science professor at University of California, Berkley at that time [62]. *Tcl* was developed with the objective to provide an easily extensible, simple and generic language with good facilities for integration. While working on *Tcl*, John Ousterhout initiated the emergence of a toolkit called *Tk*, providing the necessary tools to build GUIs in desktop applications, Unix-only at first. Since both *Tcl* and *Tk* were developed and extended in parallel, back in the 1980's *Tk* was only meant to be an extension to *Tcl* that enables the developer to implement UI solutions by reusing as many components as possible and assembling them with *Tcl*. [62]
Today, *Tk* is a cross-platform toolkit, working on a higher-level than most conventional

GUI toolkits. As *Tk* is connected to the dynamic programming language *Tcl* since its early development, it is now also supported by most dynamic programming languages available, such as *Python*, *Ruby* or *Perl*. Due to its high-level functionality, *Tk* is not only accessible to experienced developers, but offers easily to use tools for people without computer science background. *Python*'s interface to the *Tk* GUI Toolkit is called *tkinter*. [63]

Widgets Every component of a GUI created with *Tk* is a *widget*, also referred to as *controls* and *windows*. Every *widget* is an object, i.e. an instance of a class representing a UI-component, such as buttons, sliders and so on. That is, when adding a *widget* to the *Tk* application, the corresponding class has to be identified first. While there exist many different *widget* classes, some of the most common are *Frame, Label, Button, Panedwindow, Canvas* and *Checkbutton*. Furthermore, each *widget* follows a specific window hierarchy, in which each *widget* is child of a window, or *Frame*, and at the top of the whole application there is one *root* window, representing the main UI window. The hierarchy, however, can be arbitrarily deep, such that *Frames* can be children of other *Frames*. Furthermore, each *widget* has additional configuration parameters, that modify the appearance of the *widget* and make the GUI customizable, such as font size, background color and border size. [64]

Geometry Management Creating *widgets* does not automatically add them to the GUI window. Each *widget* has to be placed individually into its parent *Frame*. This process is called geometry management. *Tk* offers three different geometry managers: *Grid*, *Pack* and *Place*. The *Grid* geometry manager is the most useful and straightforward to use manager of these three, which is why the other three will not be discussed here. Geometry management in *Tk* follows the principle of master and slave. The master is any toplevel *Frame* or window, while its child *widgets* represent the slaves. In some sense, the geometry manager controls how each slave *widget* is aligned and arranged in the master window. It does so by capturing each slave's natural and desired (specified) size, as well as using each slave's set of parameters provided in code specifying how to arrange the slave inside the window. [64]
In the case of the *Grid* geometry manager, arranging *widgets* is achieved by separating each master window into rows and columns, thus spanning a grid over the total space available. Each *widget* thus specifies a row and column number (starting from index 0) to determine its relative position inside the parent *Frame*. [65]

Event Handling Handling events in *Tk*, as in many other UI libraries, is handled in an event loop. *Tk* internally manages common events, such as changing the background color of a button when hovering over the *widget* with the mouse. For each *widget* particular events can be caught by specifying a callback (in *Tk* referred to as *command*), which simply is a function executed when e.g. a button is pressed. *Widgets* without default events associated to them can still react to any event triggered, by declaring a corresponding callback function to define the reaction to that event. [64]

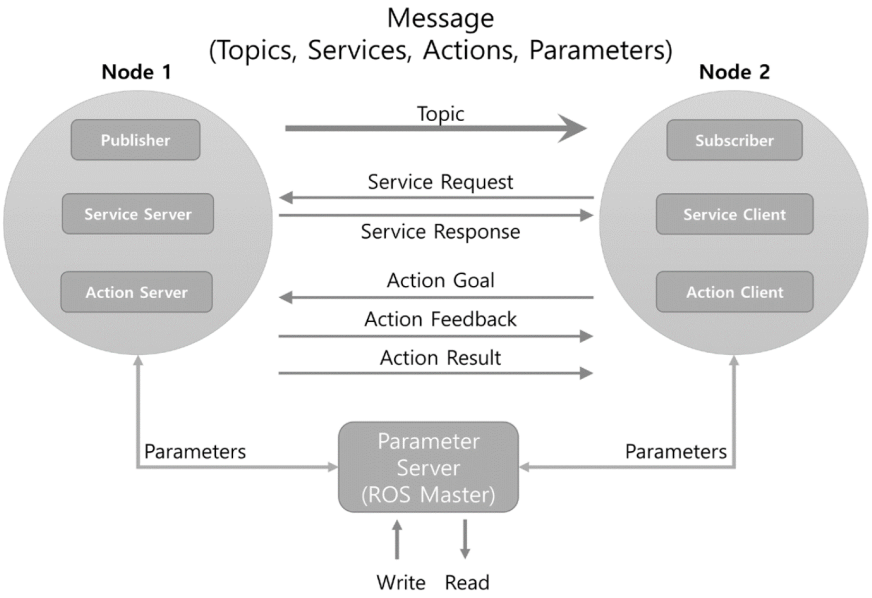

Fig. 5–3: Message communication in *ROS*. (Source: [66])

5.4 Robot Operating System

ROS is a meta-operating system, meaning that *ROS* provides functionality similar to a real operating system (OS), such as Windows, MacOS and Linux. This refers to the fact that *ROS* includes hardware abstraction layers and utilizes computing resources to perform scheduling, monitoring, message communication, error handling an many other processes. However, *ROS* cannot be installed on a computer as a standalone system, but rather runs on top of any existing Linux OS, such as Ubuntu. Besides that, *ROS* serves as a development environment, providing libraries for creating and running code specialized for robot applications. In doing so, *ROS* aims at maximizing ease-of-use and code re-usage. [66]

Main Concepts Figure 5–3 depicts the main concept of message communication in *ROS*. The figure shows two *nodes* communicating with eath other via *ROS* messages. A *node* is *ROS*' smallest unit and can be thought of a small executable program itself. Usually, one individual *node* is created for each separate function in the *ROS* application. As an example, in a robotic use-case one *node* would be implemented for driving the motors of a robot, reading its sensor data and recognizing objects, respectively. [67] Every *node* in the system is connected to a single *ROS master*, which serves as a common server for all connections and message communications within the *ROS* application. In order to connect a *node* to a *master*, the *node* has to specify a set of parameters. The *master* uses a HTTP-based protocol to communicate with its slaves (*nodes*), while the slaves themselves fall back to specific TCP/IP protocols for inter-*node* communication. [68]
Each *node* can serve as one or multiple of six types of message source or destination. Which type(s) the *node* actually represents, depends on the types of messages the *node* uses to communicate. Table 5–2 summarizes relations between the type of

node and type of message in *ROS* applications. In addition to those concepts, each *node*'s parameters can be modified from outside the *node*. Each message type and the respective concept of communication are summarized in more detail below [66]:

- **Topic:** In order to communicate via *topics*, both *publisher* (*node* sending the message) and *subscriber* (*node* receiving the message) have to be connected by specifying the exact same *topic* name as parameter. One *node* can be *publisher* and *subscriber* at the same time. *Topics* work uni-directional, meaning the *topic publisher* does not receive any feedback by the *subscriber*. That being said, *topics* are most convenient for transmitting sensor data that is continuously gathered by the robot. *Publishers* and *subscribers* cannot only be connected one by one, but also support 1:N, N:1 and N:N connections.

- **Service:** Communicating with *services* requires a *service client* requesting and a *service server* responding to the request. As opposed to *topics*, *services* are a one-time communication method. The server responds only when a client is sending a request, and as soon as both request and response communication are completed, both *nodes* are disconnected. *Services* in *ROS* are often used to command specific operations and in some cases can be used to replace *topics*, thereby reducing the communication load in the program.

- **Action:** Similar to *services*, *actions* work bi-directional. However, an *action* is composed of a goal sent by the *action client* and a result answered by the *action server* on completion. Both *action* goal and *action* result correspond to the *service* request and response, respectively. Since *actions* are used to replace *services* whenever execution of a command takes a very long time, such as moving a robotic arm to a specific position or making it follow a trajectory, *ROS actions* additionally define a feedback provided by the action server at any time. This allows for the client to monitor the progress of the execution of an *action* and tells it how close the goal is. Other than that, the *action client* can cancel the execution of an *action* at any time.

Tab. 5–2: *ROS* message communication methods and their applications. (Adapted from: [66])

Node type	Message type	Features	Description
Subscriber, *Publisher*	*Topic*	Asynchronous, Unidirectional	Continuously transfer data between *nodes*
Service Client, *Service Server*	*Service*	Synchronous, Bi-directional	Used when message communication requires request-response principle
Action Client, *Action Server*	*Action*	Asynchronous, Bi-directional	Provides immediate feedback during long response times, alternative to server concept when dealing with long-running requests

```
<link name="base_link">
    <visual>
        <geometry>
            <cylinder length="0.6" radius="0.2"/>
        </geometry>
        <material name="blue">
            <color rgba="0 0 .8 1"/>
        </material>
    </visual>
    <collision>
        <geometry>
            <cylinder length="0.6" radius="0.2"/>
        </geometry>
    </collision>
    <inertial>
        <mass value="10"/>
        <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>
    </inertial>
</link>
```

Fig. 5–4: Example URDF robot link written in XML. (Source: [69])

***Unified Robot Description Format* (*URDF*)** In *ROS*, the model of a robot is represented by an *URDF* file [69]. *URDF* is formatted and written in XML. *URDF* allows to define a multi-body system model of the robot. Some of the most common and in robot models often reused URDF specifications are depicted in Figure 5–4. The figure shows an example on how to create a robotic link, tagged with a unique name, and several additional components. The visual appearance of the link is specified first. This can be done by combining one or multiple geometric primitives and assign a material and color, or by specifying a path to a computer-aided design (CAD) file (in *dae* or *stl* format) containing a mesh representation of the link. Similarly, collision shapes of the link are defined. Physical properties like mass and inertia are configured inside the *inertial* tag. Joints are added to a robot model in a very similar fashion. However, they comprise different parameters that have to be set, including the parent and child object the joint is connecting, its axis of rotation, the joint position, velocity and effort limits and many more. Defining a robot in this way facilitates the modular creation of the robot model, since each component, in other words each of the robot links and joints, can be added individually and parametrized according to its real-world specifications. [69]

5.4.1 MoveIt! Motion Planning Framework

MoveIt! is the primary open-source robotics toolkit for motion planning and robot manipulation in *ROS*. Many of the standard robot models available in *ROS* already support the integration of *MoveIt!* into the *ROS* application work flow. *MoveIt!* is a software framework written entirely in *C++*, with *Python* implementations for high-level scripting. Its main purpose is to provide easily usable motion planning algorithms in *ROS*. Therefore, *MoveIt!* uses *ROS*' messaging and communication system, and defines its core functionalities in plugins from three different areas: motion planning, collision detection and kinematics. *MoveIt!* is thus widely used by the *ROS* community to calculate forward and inverse kinematics of a robot model, detect and avoid collisions with objects, plan trajectories in joint or Cartesian space and many more. [70]

5.5 ROS#

ROS# [71] is an open-source library developed and maintained by Siemens, providing tools written in *C#* that enable communication between *ROS* and *.NET* applications. Primarily, *ROS#* was developed to communicate between *Unity* simulations and *ROS* systems. While *ROS#* was implemented for Windows, the software has successfully been deployed on various other platforms by community members. Above all, *ROS#* aims at providing a simple tool for using standard *ROS* communication concepts, such as *topics*, *services* and *actions* within *.NET* programs. Furthermore, *ROS#* also offers an *URDF* importer, which allows the user to import a *URDF* file into *Unity* in a straightforward manner. Thus, *ROS#* makes it possible to e.g. visualize and simulate the robot in *Unity* (detached from *ROS*), control the real robot via *Unity*, visualize the state of the real robot in *Unity*, simulate in *Unity* and visualize in *ROS*, or even simulate the robot in both *Unity* and *ROS* and compare the results. [71]
ROS# provides a *Unity* package containing the required project *assets* to import *ROS#* into *Unity* applications for direct usage. However, all libraries are also available as independent *.NET* solutions and hence can be used in *.NET* projects independent from *Unity*, as well. Three main libraries compose the complete open-source library: (1) *RosBridgeClient* provides the *.NET* communication interface to *ROS* via *rosbridge_suite* (a *ROS* package to communicate with *ROS* from non-*ROS* applications); (2) *UrdfImporter* contains a file parser to import *URDF* files in *.NET* applications; (3) *MessageGeneration* is the *.NET* library that automatically generates *C#* source code from *ROS* messages, *services* and *actions*. [71]

6 Evaluation Framework

The evaluation framework for evaluating simulation models, i.e. *digital twins*, of mechatronic systems developed in this thesis is presented in this chapter. During the development of this framework, decisions concerning architecture, design, code structure, data storage and parallelization had to be made. While explaining the functionality of each part of the framework, this chapter also focusses on these decisions, the reasoning behind them and their realization. Specific information on these development choices are arranged in separate subsections, providing details about the underlying context.
The chapter starts in Section 6.1 by explaining the complete architecture of the framework on a high level, introducing its separate, independently usable components and the data and files communicated between these components. Subsequently, each component is discussed in detail. The sequence of actions each component performs and which main classes and functions it uses are described thoroughly. Section 6.2 illustrates how the *evaluation space* can be defined from inside the *Unity* editor, and explains concepts of the code providing this functionality. Afterwards, Section 6.3 summarizes how the code developed in *.NET* repeatedly calls an executable and evaluates the simulation. Finally, Section 6.4 outlines the idea of the third component of the framework, which reads in a *.txt* file and produces a visualization of the evaluation results alongside a UI for the user.

6.1 Framework Architecture

Figure 6–1 shows a scheme of the general architecture of the evaluation framework, split into three main segments visualized as blue pieces. In the remainder of the thesis, these segments will be referred to as components of the framework. The naming of the three components reflects their respective role in the process of evaluating a simulation model: (1) *Definition of Evaluation Space*; (2) *Execution and Evaluation*; (3) *Point Cloud Visualization*. The architecture highlights that the components can be executed in a sequence.
Besides its main components, the evaluation framework includes three segments required to transfer information, which are represented by gray pieces. Hereinafter, these bridges between components will be referred to as inputs.

Evaluating a Mechatronic System in Simulation Evaluation of a mechatronic system based on its representation in simulation requires multiple distinct, but closely connected steps. In general, these steps can be directly linked to the concept of virtual prototyping presented in Chapter 2. At first, the user has to define the *evaluation space* in which the *digital twin* should be evaluated. This includes the specification of a set of parameters which should be varied in the simulation model of the mechatronic system. Such a parameter set usually relates to properties of components of the inspected mechatronic system, whose variation is expected to have impact on things like appearance, behaviour, functionality, efficiency or safety of the system. In the implemented software architecture, these parameters are entitled *evaluation parameters*. Generally,

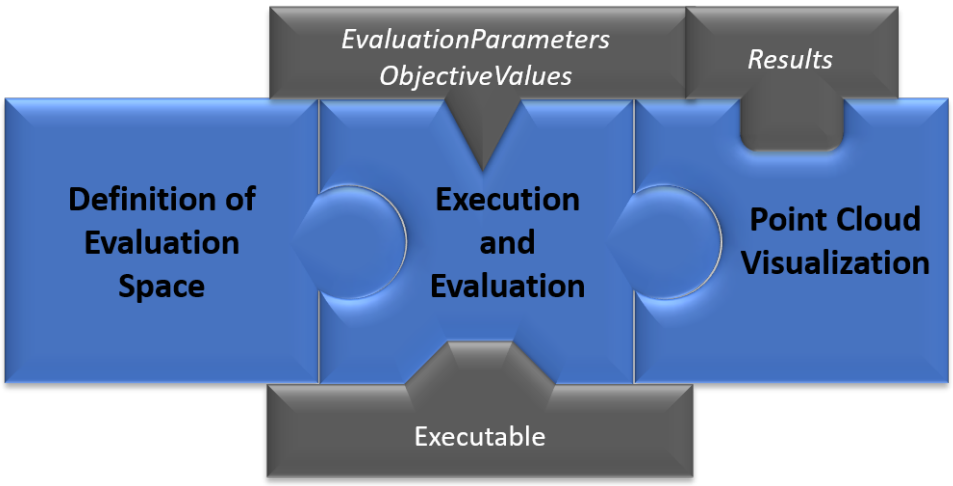

Fig. 6–1: Complete architecture of the evaluation framework. *Blue*: three main components of the framework, *gray:* inputs of the three main components.

the user wants to identify optimal settings for the *evaluation parameters* and thus is interested in simulating the mechatronic system over a range of different values. Hence, the user chooses an *evaluation parameter* and specifies a range of values, in which the parameter should be varied, together with a step size. This process is repeated until all desired *evaluation parameters* and their range of variation are defined.
Objective values comprise the second part of the *evaluation space*. These values represent decision criteria the user wants to be evaluated during simulation. This means that *objective values* are typically properties of the mechatronic system, whose values the user is interested in when simulating the system with different *evaluation parameters*. Below, two exemplary mechatronic systems with possible sets of *evaluation parameters* and *objective values* are listed.

- Autonomous vehicle
  - **Evaluation parameters:** Scanning resolution of LIDAR sensor; Controller parameters for steering
  - **Objective values:** Number of obstacles detected in test environment; Ability to autonomously follow a line
- Robotic arm
  - **Evaluation parameters:** Type of motor used in joints; Length of robot links; Intermediate joint configuration on trajectory
  - **Objective values:** Degree of realizability of example robot motions with each type of motor; Production costs of robot links; Energy efficiency during motion on trajectories

The total *evaluation space* then comprises a grid spanned over all *evaluation parameters*, thus including all possible combinations of *evaluation parameters*, as well as the user-defined *objective values* as functions to be evaluated for each set of *evaluation parameters*.

Secondly, for each *evaluation parameter* combination, the mechatronic system is simulated (execution) and the predefined *objective values* are calculated (evaluation). One step of execution and evaluation is hereinafter referred to as one *evaluation step*. Running simulations and retrieving *objective values* one by one does not allow the user to intuitively evaluate the model by recognizing relations between *evaluation parameters* and *objective values*. Therefore, the third step is the visualization of the point clouds generated by repeatedly running an *evaluation step* for each combination of *evaluation parameters*. A good visualization in combination with intuitively designed UI elements enables the user to inspect the *evaluation space* from different perspectives, identify interdependencies and effects of *evaluation parameters* on *objective values*, select optimized sets of parameters (optimal in the sense of application-specific objectives) and find the best trade off solution in conflicting objectives. This is closely connected to the MCDM problems introduced in Chapter 2. In this context, the *evaluation parameters* of the simulation correspond to the alternatives of the decision analysis problem, the *objective values* refer to the attributes, that is, decision criteria in MCDM.

The evaluation framework as a whole is broken down to its individual components and their interconnections in Figure 6–2. The three components of the evaluation framework use three different development environments: simulating the mechatronic system and defining the *evaluation space* is done in *Unity*, execution and evaluation of each executable is implemented in Mircrosoft *.NET* and the point cloud visualization is carried out with the help of *Python*'s *tkinter* library. Besides that, this figure depicts which component is responsible for providing which input. *Unity* is responsible for creating an executable (*.exe* file) of the simulation. Additionally, the *evaluation space* is defined inside the *Unity* editor and exported into two *.json* files. The *.NET* implementation of the evaluation framework reads these two *.json* files and then periodically calls the *Unity* executable, sends a combination of *evaluation parameters* and receives the set of *objective values*. After all parameter combinations have been simulated and evaluated, the results are stored into a *.txt* file. This file in turn is the input for the final component of the framework: the point cloud visualization.

6.1.1 Software Architecture

The architecture of the evaluation framework developed in this thesis is closely linked to the aforementioned process of evaluating a simulation model of a mechatronic system as easily and intuitively as possible. As a result, the user of the framework immediately recognizes each component and understands its purpose in the context of the whole software. Besides that, the main motivation behind this software design is the goal of developing the framework as generic as possible, thus providing an easily re-usable and transferable set of software components. Even though all three components of the framework are closely linked in the process of evaluating a mechatronic system in simulation (see Figure 6–1), each component can also be used individually, detached from its connected components. As a result of the chosen architecture, providing the *Execution and Evaluation* component with two sets of parameters, *evaluation parameters* and *objective values*, respectively, and an executable build from the simulation of the mechatronic system, this component runs all evaluation steps automatically. It is not dependent on a specific development environment used to define the *evaluation space*

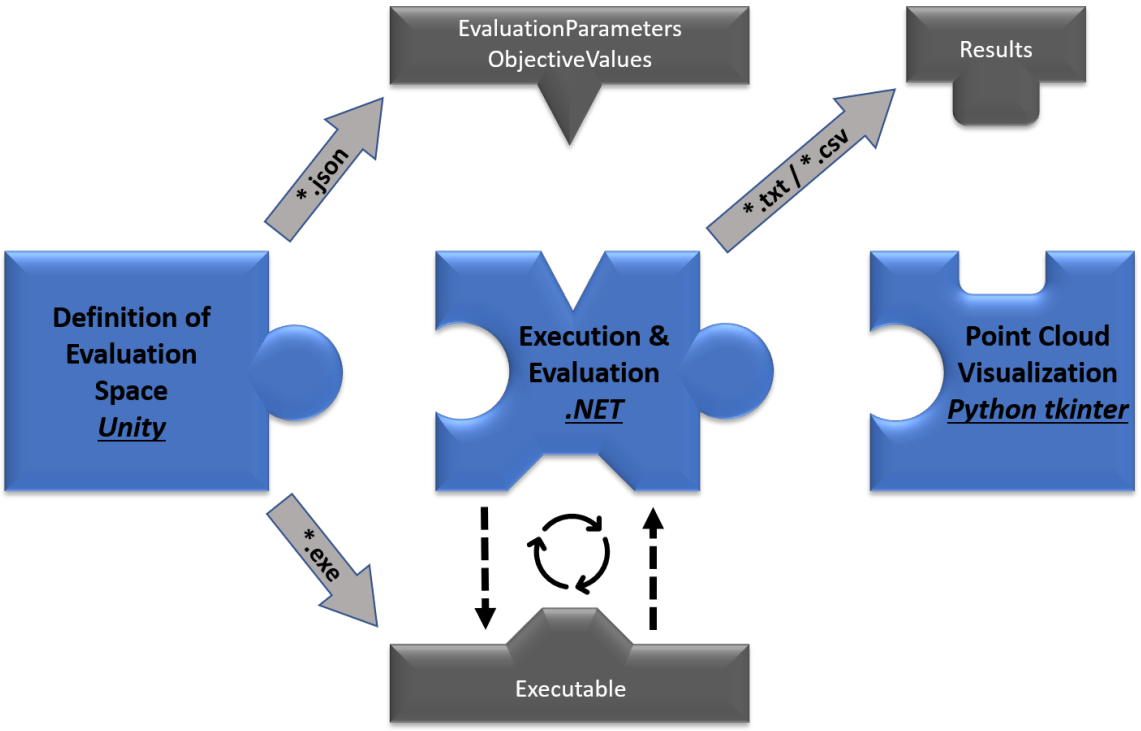

Fig. 6–2: Complete architecture of evaluation framework broken down to each component.

and to simulate in. Furthermore, the GUI visualizing point clouds and providing a UI for interaction does not depend on the user having defined the *evaluation space* and simulated and evaluated the mechatronic system with the evaluation framework. This third framework component can be also used as a standalone GUI to inspect point clouds and results obtained from completely different applications. Hence, if any of the three components is provided with its respective input, it works independently, can be separated from the remaining evaluation framework and used as a standalone software tool.

6.1.2 Choice of Development Environments

When designing the evaluation framework, the development environments for all three components had to be chosen. The game engine *Unity* served as a simulation environment during the course of this thesis. *Unity* was selected due to its easy-to-use and intuitive UI, allowing for a quick familiarization with its concepts. Aside from that, *Unity* is already widely used in many industries for real-time 3D visualization and simulation. Hence, it is a great tool for building *digital twins* and representing mechatronic systems in simulation. Apart from that, both the *Chair of Astronautics* of *TUM* and the *Siemens AG* already deploy *Unity* in some of their projects. That made it possible to use lots of already existing know-how to support the development process. As already mentioned, *Unity* also provides lots of options to customize the appearance of the editor and add scripted editor windows for specific purposes. This is especially helpful when defining the *evaluation space* by selecting properties of the simulated mechatronic system (see Section 6.2). Furthermore, *Unity* offers a simple way of building any simulation into

an executable and call it as a standalone program. This feature made it possible to further generalize the evaluation framework by implementing the *.NET* component in such a way that it only needs the path to an executable, without explicitly depending on a specific simulation environment for the mechatronic system.
Scripting in *Unity* is mostly done in *C#*, which is the reason why *Microsoft*'s development platform *.NET* alongside the *C#* programming language were utilized to implement the *Execution and Evaluation* component of the framework. This ensures a smooth transition from implementing scripts in *Unity* to building a *.NET* solution. With *TPL*, (see Section 5.2.1) *.NET* also provides a great library to easily implement parallel code, which is definitely an important requirement when running thousands, or even more, simulations. Beyond that, *.NET* is a cross-platform development environment, which makes it easier to implement device agnostic code and port the framework to other operating systems. Furthermore, *ROS#*, as described in Section 5.5, is a set of *C#* libraries implemented as *.NET* solutions. This results in an easy integration of *ROS#* into the *Execution and Evaluation* component of the evaluation framework. An application of such a software integration will be presented in Section 7.3.
For developing the visualization component of the framework, *Mathworks' MATLAB* and *Python* came into question. Both provide great and easy-to-use visualization techniques and GUI development tools. In order to retain the possibility of making the evaluation framework publicly available as an open-source project, the point cloud visualization was implemented in *Python*, since *MATLAB* does not offer any open-source solutions. *tkinter* as a toolkit for creating the GUI was chosen due to its popularity in the *Python* community, being one of the primarily used libraries for developing GUIs, and its easily expandable and customizable UI *widgets*.

6.1.3 Data Storage and File System

Transferring data between the evaluation framework's components is essential to ensure a smooth program sequence. As can be seen in Figure 6–2, *evaluation parameters* and *objective values* are stored in *.json* files, while the point clouds are saved into a *.txt* file. For the definition of the *evaluation space* and its transfer into readable files, the json file system was preferred over *.yaml* and *.txt* files. The reason behind that is the fact that *C#* offers straightforward serialization of data into *.json* files through built-in serialization methods and external packages. The *Newtonsoft Json.NET* [72] framework is a popular package containing many different serialization libraries for storing data into *.json* files. An implementation of *Newtonsoft Json.NET* is also available as an extension to *Unity*. Alternative software for json serialization include *Microsoft .NET*'s *System.Text.Json* namespace [73], or *Unity*'s *Json Serialization* [74]. However, the former *.NET* toolkit does not support serialization from within *Unity*, while the latter *Unity* class only provides serialization for basic data types like strings and integer variables. Consequently, the *Newtonsoft Json.NET* package was used to both store *evaluation parameters* and *objective values* from *Unity* into *.json* files, as well as to read those files in the framework's *Execution and Evaluation* component. Furthermore, the json file format fits to the structure of data obtained from defining the *evaluation space*. Data in *.json* files is structured into separate fields, where in this context each field represents an *evaluation parameter* or *objective value*. Additionally, the data in those files can be

easily extended by adding new fields, that is, parameters, while existing fields can be modified independently.
Keeping in mind that the visualization of the simulation results falls back on *Python*'s *tkinter* library, it was decided to use a *.txt* file to transmit the point cloud data between the latter two components of the framework. *Python* offers many tools to read csv-formatted *.txt* files (or plain *.csv* files) in a simple and efficient manner. Additionally, using the *.txt* file format makes it easy to organize data into tables, which is the most convenient way of formatting the *evaluation space*: one column per *evaluation parameter*, one column per *objective value*, and, correspondingly, one row for each combination of *evaluation parameters* and the resulting *objective values*. On these grounds, the point cloud data is serialized into a *.txt* file and used as input by the *Point Cloud Visualization* component. Along with the reasoning behind these choices of data storage, the aforementioned generalization of the evaluation framework plays a central role. All the file systems used in the evaluation framework are widely used and supported tools for data storage and transmission. Hence, data in the format of these files can easily be provided by other software tools as well, making it easier to incorporate other development environments into the evaluation framework or use each component as a standalone program.

6.2 Definition of Evaluation Space

The *evaluation space* can be defined from inside the *Unity* editor. In order to make this possible, a concept had to be developed that allows the user to choose *evaluation parameters* and *objective values* in an intuitive manner while *Unity* itself is in *Edit Mode*. The idea of *Unity*'s editor scripting as introduced in Section 5.1.2 plays the main role in this context. Figure 6–3 shows a scheme of the code structure providing the functionality for selecting *evaluation parameters* in *Unity*. Please note that the same concept is used for *objective values* as well. The three concepts of customizing the *Unity* editor through *C#* scripts introduced in Section 5.1.2 are utilized in this part of the evaluation framework:

1. Adding additional editor windows through *Unity*'s *EditorWindow* class
2. Using *CustomEditor* as a tool for changing the appearance of a *Component* in the inspector window
3. Utilization of *Unity*'s *ScriptableObject* as data storage object

To this end, most of the code implemented in *Unity* is packed into the already introduced *OnGUI* and *OnInspectorGUI* functions.

The *Unity* component of the evaluation framework can be initiated by selecting a menu item in the *Unity* editor specifically added in the context of this component. This results in two things. Firstly, an empty *GameObject* named *EvaluationFramework* is added to *Unity*'s hierarchy. Secondly, a customized *EditorWindow* pops up, referred to as the *EvaluationParameterManager* (see Figure 6–3). This window provides buttons to add and remove *evaluation parameters*. Besides that, it lists information of the already selected *evaluation parameters* inside the window. Now, adding an *evaluation parameter* initiates a sequence of actions inside the *Unity* editor:

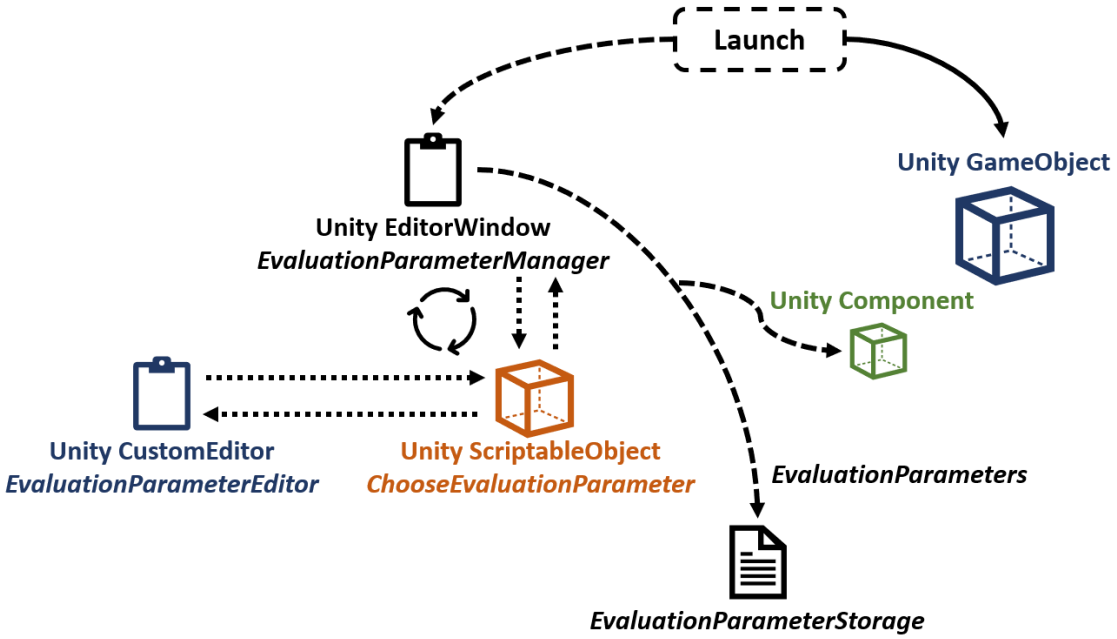

Fig. 6–3: Selection of *evaluation parameters* in the *Unity* editor.

1. A *ScriptableObject* is created and stored in the *Unity* assets folder.
2. The appearance of said *ScriptableObject* in the inspector window is modified by a *CustomEditor* referred to as *EvaluationParameterEditor*.
3. Through UI elements in the *ScriptableObject*'s inspector, the user can now select a *GameObject*, one of its *Components* and finally a variable, i.e. a *Property*, of the *Component* as *evaluation parameter*. The *Component* has to be a *C#* script *Component*, and cannot be one of *Unity*'s built-in *Components*.
4. On selection of a variable, the user is prompted with fields to enter the settings for the selected *evaluation parameter*. The framework supports the selection of integer, float and boolean variables, as well as integer and float arrays. For integer and float variables, this includes the definition of a minimum value, a maximum value and a step size at which the parameter should be varied. For array variables, the size of the array has to be specified additionally. Selecting a boolean variable as *evaluation parameter* does not require any further specification.
5. Finally, the user optionally can set the unit of the selected *evaluation parameter*. This makes it easier to understand and draw conclusions from the visualization of the point clouds later on.

When the selection process is complete, the information about the *evaluation parameter* pops up inside the *EvaluationParameterManager* editor window. The information displays the name of the selected *GameObject*, *Component* and variable, as well as the defined minimum and maximum values and step size.
Hence, for each *evaluation parameter* a separate *ScriptableObject* is created. Removing an *evaluation parameter* from the *evaluation space* simply deletes the corresponding

ScriptableObject. As soon as the user is satisfied with the selection of *evaluation parameters*, a button in the *EvaluationParameterManager* editor window allows the user to save all information contained in the different *ScriptableObjects* as an array of *evaluation parameters* into a separately implemented *C#* class entitled *EvaluationParameterStorage*. Additionally, a unique script *Component* is added to the initially created *GameObject EvaluationFramework*. This *Component* stores the same array of *evaluation parameters* for later usage. Subsequently, this whole process is repeated analogously for the definition of all *objective values*.
Finally, the user initiates the serialization of the parameter arrays from both *EvaluationParameterStorage* and *ObjectiveValueStorage* (not depicted in Figure 6–3) into two *.json* files, respectively, with the help of the aforementioned *Newtonsoft Json.NET* package. Additionally, the whole simulation scene in *Unity* is built into a standalone executable, which is used later on by the *Execution and Evaluation* component of the evaluation framework.

6.2.1 Intuitive, Reusable and Modular Selection of Parameters

The design of this component of the evaluation framework is motivated by making its usage as intuitive and easy as possible, even for users with only basic knowledge of *Unity*'s editor and the concepts of *GameObjects* and *Components*. With the architecture mentioned above, the user is able to quickly understand the process of selecting an *evaluation parameter*. Once understood, the same steps are repeated for the complete process of defining the *evaluation space*.
While moving through the process, the user can review the *evaluation space* selected so far by either looking at the information provided in the two *EvaluationParameterManager* editor windows, by inspecting the information saved in each distinct *ScriptableObject* listed inside the assets folder, or by looking at the saved *.json* files. Besides that, all UI elements in the editor and inspector windows are straightforward to use. Removing mistakenly configured parameters is as easy as clicking a button. Selecting a *GameObject* from the hierarchy window is possible through drag-and-drop or a built-in *Unity* selection window. Choosing *Components* and variables thereafter is realised by simple drop down menus. Besides that, some UI elements are locked until special events in the *Unity* editor are triggered. As an example, a new *evaluation parameter* can only be added as soon as the previously selected parameter is saved into a *ScriptableObject*. The button inside the *EvaluationParameterManager* window for building the executable is only activated after both *evaluation parameters* and *objective values* were safely stored into the *.json* files. This ensures a simple and reliable definition of the *evaluation space*.
Apart from that, the outlined design of this component in *Unity* is very modular. The implemented *EvaluationParameterManager* editor windows for selecting *evaluation parameters* and *objective values* are almost identical. Likewise, the CustomEditors for *ScriptableObjects*, whether it stores an *evaluation parameter* or an *objective value*, are nearly the same. Lastly, the implementation of the functionality of providing the user with a list of all *Components* and their variables for a selected *GameObject* is independent from what *GameObject* is actually selected. All of this results in the fact that this component of the developed evaluation framework is completely detached from the *Unity* simulation itself and can be used for any type of application developed in

Unity, since it only runs in *Edit Mode* and does not require any information from the simulation's behaviour.

6.2.2 Data Handling in Unity

Saving data during the course of defining the *evaluation space* is crucial. When deciding on how to handle the storage of *evaluation parameters* and *objective values* in *Unity*, the fact that the code fully executes in *Edit Mode* played a central role. As described in Section 5.1.2, *ScriptableObjects* are great for storing data gathered during *Unity*'s *Edit Mode*. Besides that, using *ScriptableObjects* made it easy to assign one *ScriptableObject* for each defined parameter, thus keeping the selection process clear and traceable. In contrast, storing all selected parameters in one single file would result in the opposite. Firstly, it would be cumbersome and inefficient to remove any already selected parameter from the file. Secondly, the user would have a much harder time to maintain an overview of the selected *evaluation space*.
However, *ScriptableObjects* are saved as assets and stored in the project's assets folder. When building a standalone executable from any *Unity* simulation, the executable cannot access any *ScriptableObject* asset. As a result, the already gathered information on the selected *GameObject*, *Component* and variable for each parameter would be lost. In order to circumvent this issue, the sets of *evaluation parameters* and *objective values* are additionally packed into the *EvaluationFramework GameObject*. By adding one dedicated *Component* to the *GameObject* for each set of parameters, respectively, the information about the *evaluation space* is also available inside the executable and can be used for communication with the framework's *Execution and Evaluation* component. This bypasses the necessity to communicate information about the *evaluation parameters* to the executable each time it is called from the *.NET* solution, and reduces the communication between these two programs to just the values of the parameters.

6.3 Execution and Evaluation

In order to obtain an overview on the solution space and to identify optimized system configurations, many evaluations must be performed. The *Execution and Evaluation* component of the evaluation framework developed in this thesis carries out this large number of simulations and stores the resulting point clouds in a format that can then be read by other visualization tools for spreadsheets for further analysis. The component initially reads the *.json* file specifying all information about the *evaluation parameters*. Since each parameter is given with a range in which it should vary, as well as a step size along this range, each *evaluation parameter* in *evaluation space* can be represented by an array, or in case the *evaluation parameter* already is an array, it is represented by a matrix in *evaluation space*. The goal of the evaluation framework is to simulate the whole *evaluation space* in a brute-force fashion. This means that all *evaluation parameters* are combined with each other to form a multi-dimensional point cloud containing all possible combinations of *evaluation parameters* in *evaluation space*. What remains for the *Execution and Evaluation* component to carry out, is to call the simulation executable for each of these *evaluation parameter* combinations and retrieve the predefined *objective values* as results. Figure 6–4 summarizes this process of

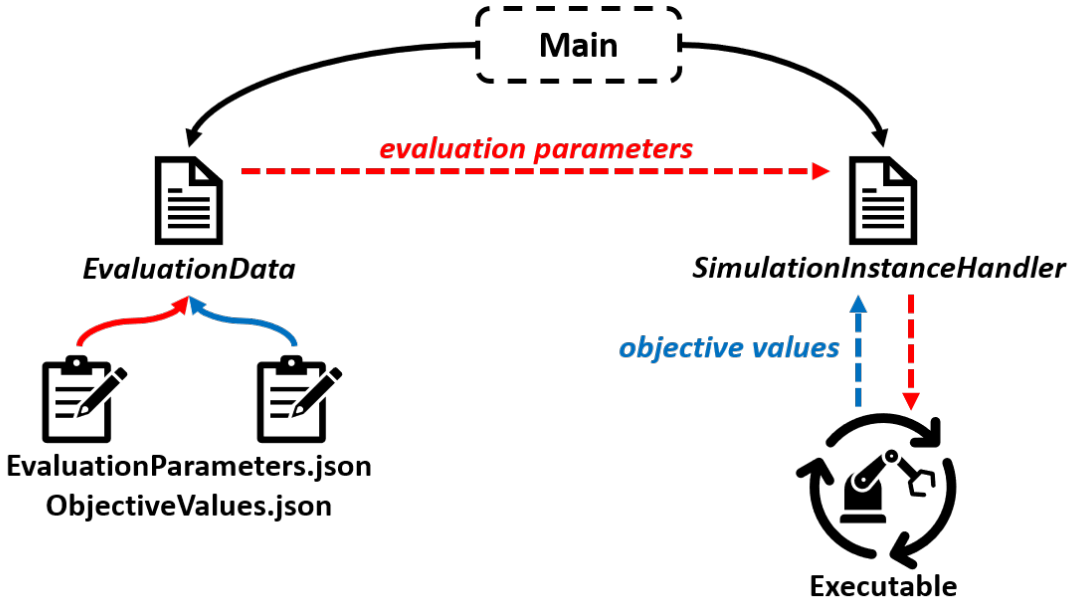

Fig. 6–4: Concept of executing and evaluating simulation executables. *Red arrows: evaluation parameters*, *blue arrows: objective values*.

reading the *.json* files, spanning a multi-dimensional grid over the *evaluation space* and carrying out a simulation for each combination of *evaluation parameters*. It does so by scheming the implementation of said functionality in *Microsoft*'s *.NET* platform.

From the main entry-point of the *.NET* implementation, the definition of the *evaluation space* in form of the two *.json* files is used. Therefore, a static *C#* class entitled *EvaluationData* was implemented, reading both files by falling back on the *Newtonsoft Json.NET* package. The class reads these files and spans the previously addressed multi-dimensional grid over the whole *evaluation space*. The resulting *evaluation space* is stored in a multi-dimensional *C#*-list. As depicted in Figure 6–4, the *path* of the *evaluation parameters* in this framework component is represented by red arrows, while the *objective values* are visualized in blue. Another implemented class of the *.NET* component entitled *SimulationInstanceHandler* retrieves each combination of *evaluation parameters* and is from then on responsible for executing simulations and fetching *objective values*. In short, one instance of the *SimulationInstanceHandler* class is created per *evaluation parameter* combination, and each one of them calls the executable, communicates its combination of *evaluation parameters* to the executable and receives results in the form of *objective values*. Both upcoming Sections 6.3.1 and 6.3.2 elucidate this process of communicating parameters between the *.NET* solution and the executable and the concept of parallelization implemented in this context.

6.3.1 Interprocess Communication

Simply calling and running the *Unity* executable is not enough for evaluating a *digital twin* of a mechatronic system. The course of the simulation, as well as the values of the *objective values* likely depend on the values of the *evaluation parameters*. Hence, as already mentioned, the standalone executable requires a combination of *evalua-*

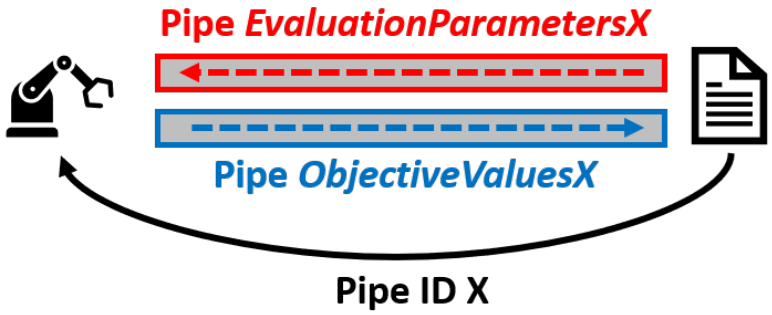

Fig. 6–5: Interprocess communication between the *.NET* solution and *Unity* executable.

tion parameters. Likewise, the *Execution and Evaluation* component of the evaluation framework has to fetch all *objective values* at the end of each simulation from each executable. Since the executable itself and the *Execution and Evaluation* component run on separate processes, a direct communication between both processes is referred to as interprocess communication. Figure 6–5 visualizes how this interprocess communication is realized, following the same scheme of colourization as used in Figure 6–4. Furthermore, the symbol representing the executable file of the simulation in Figure 6–4 is re-used likewise. For each combination of *evaluation parameters*, the corresponding *SimulationInstanceHandler* class calls the executable, sends its set of *evaluation parameters* and, when the simulation is completed, it receives and stores the resulting *objective values*. In Figure 6–5, this process is depicted for one *SimulationInstanceHandler* instance communicating with one executable.

The implementation falls back on a *C#* concept of interprocess communication called *pipes*. A *pipe* can be basically thought of as a connection between two processes. One of the processes simply writes data, or in other words messages, into the *pipe*, while the other process reads these messages from the *pipe*. [75] It is for this reason that *pipes* pose a very handy way of communicating *evaluation parameters* and *objective values* between the participating processes. While *C#* supports both anonymous and named *pipes*, in the context of the procedure of repeatedly and concurrently communicating with multiple executables at a time, using named *pipes* provides a more reliable and plausible means of message exchanging.
It can be seen in Figure 6–5 that two one-way *pipes* are used: one for sending a combination of *evaluation parameters* to the executable (red) and a second one for receiving *objective values* at the end of the simulation (blue). In order to achieve this behaviour, the two components of the *GameObject* added to the *Unity* scene while defining the *evaluation space* (see Section 6.2) open their end of the two *pipes* in *Unity*'s *Awake* and *OnApplicationQuit* functions for receiving *evaluation parameters* in the beginning and sending *objective values* in the end of the simulation, respectively. As mentioned earlier, these two *pipes* must have unique names assigned. To this end, each instance of the *SimulationInstanceHandler* class holds an integer representing its *pipe* ID. When launching an executable, the *pipe* ID is handed over to the executable as a command line argument, represented by the black arrow in Figure 6–5. Thereby, both processes communicating via the two *pipes* share a unique ID used to name both *pipes*

and can now directly exchange messages.
It is important to note that in *C#*, writing data into a *pipe* is done byte-wise without stating the type of data currently being transmitted. When reading the data, however, one obviously needs to know the data type. This problem is solved by allocating *evaluation parameters* and *objective values* in a pre-defined order. To keep it simple, this order matches exactly the order in which *evaluation parameters* and *objective values* were selected in *Unity*, and hence matches the order in which these parameters are written into and read from the *.json* files. That is why during the process of defining the *evaluation space* in *Unity*, arrays of all *evaluation parameters* and *objective values*, including the *GameObject*, *Component* and variable of each parameter, were stored in the simulation scene for reuse during simulation.

Alternatives for Interprocess Communication When implementing the interprocess communication, three additional options for designing the communication interface were investigated: command line parsing, shared memory and reading and writing files. The first method refers to the idea of specifying the combination of *evaluation parameters* as command line arguments to the *Unity* executable. Since command line arguments can easily be parsed form inside the executable, this method provides an easy to implement and efficient way to send data to the executable. However, command line arguments cannot be utilized to receive *objective values* from the executable. As a result, the *objective values* would have to be written into a temporary file and in turn read by the *SimulationInstanceHandler* object. This would result in a very in-efficient and cumbersome way of exchanging messages, especially when thousands or tens of thousands of simulations have to be executed and evaluated.
The second interprocess communication concept uses memory allocated specifically for the purpose of being shared between processes. All participating processes would have the possibility to access the same set of *evaluation parameters* from memory and store *objective values* at the end of simulation. Having said that, using shared memory can be very cumbersome when dealing with concurrent memory access, since memory locks have to be considered.
Finally, reading and writing files is another option realizing communication between the *Unity* executable and the *.NET* solution, that is, communication between two processes. In comparison to *pipe* communication, this method, however, is cumbersome in implementation and consumes more memory, and, hence is not as efficient as *pipe* communication.
As already mentioned, *evaluation parameters* and *objective values* are allocated and sent through the two *pipes* in a pre-defined order. In doing so, the need for cumbersome, additional data communication between the executable and *.NET* is prevented. However, in order to gain more flexibility and use the *Execution and Evaluation* component with simulation environments other than *Unity* (where the order of parameters might not be known to the simulation model), the order of *evaluation parameters* send to and *objective values* received from the model could initially be communicated via command line arguments, or written once into a file shared between both processes.
In the underlying context, only one-to-one communication is necessary, making it intuitively plausible to use *pipes* as a means of communication.

Conclusively, transmitting *evaluation parameters* and *objective values* via *pipes* seemed to provide the most efficient and straightforward instrument for interprocess communication in this application.

6.3.2 Asynchronous Operations and Parallel Execution

One main focus when implementing the evaluation framework's *Execution and Evaluation* component was laid on parallelizing the repeated execution of the executable. Spanning the aforementioned multi-dimensional grid in the *evaluation space* can lead to tens of thousands of combinations of *evaluation parameters*. Synchronously simulating one executable for each parameter combination might take up many hours or even days. *Microsoft .NET*'s *TPL* as introduced in Section 5.2.1 is a library for high-level implementation of parallel code. Figure 6–6 depicts in what way *TPL* is used in the *Execution and Evaluation* component to efficiently parallelize the loop in which the *Unity* executable is called with each combination of *evaluation parameters*. The figure highlights two main concepts in the parallel implementation of this code. Firstly, for each combination of *evaluation parameters* a new object of the *SimulationInstanceHandler* class is created. This can be completed before entering the main loop of executing and evaluating each executable. Secondly, the most efficient parallelization in this context is achieved by running one executable per CPU core. This ensures full saturation of CPU resources and achieves a CPU occupancy rate of 100 % during the whole process.

Architecture of Parallel Code The reasoning behind this choice of parallelization architecture is supported by various arguments. Ensuring that each combination of *evaluation parameters* is handled by a distinct object of the *SimulationInstanceHandler* class circumvents the expense of dealing with memory locks and faulty interaction between executables in a cumbersome way. Each *SimulationInstanceHandler* object has its unique *pipe* ID and holds one combination of *evaluation parameters*. The *pipe* IDs are defined when spanning the multi-dimensional grid of *evaluation parameters*: every parameter combination gets a unique *pipe* ID assigned, corresponding to the index of the parameter combination inside the grid. Thus, each object can individually launch the executable, pass the ID as a command line argument and independently communicate via two peculiarly named *pipes* with the executable. Furthermore, the *SimulationInstanceHandler* object retrieves the *objective values* of its assigned executable (blue arrows in Figure 6–6) and stores them individually until all executables have been called and evaluated. As soon as this main loop of simulating all combinations of *evaluation parameters* is complete, the stored data of all *SimulationInstanceHandler* objects is gathered and written into the resulting *.txt* file at once (violet arrows in Figure 6–6). As a result, all combinations of *evaluation parameters* are evaluated independently from each other and are perfectly suitable for being parallelized.
As already mentioned, *TPL* provides high-level tools for parallelizing execution in *.NET* solutions. Things like memory allocation for threads, properly opening and closing them or handling efficient thread distribution across resources are managed by *TPL* internally. Doing very CPU-heavy work, such as running multiple *Unity* executables, on one CPU can in some cases even slow down the execution. Hence, the processes running the simulation executables are automatically distributed across the available CPU resources,

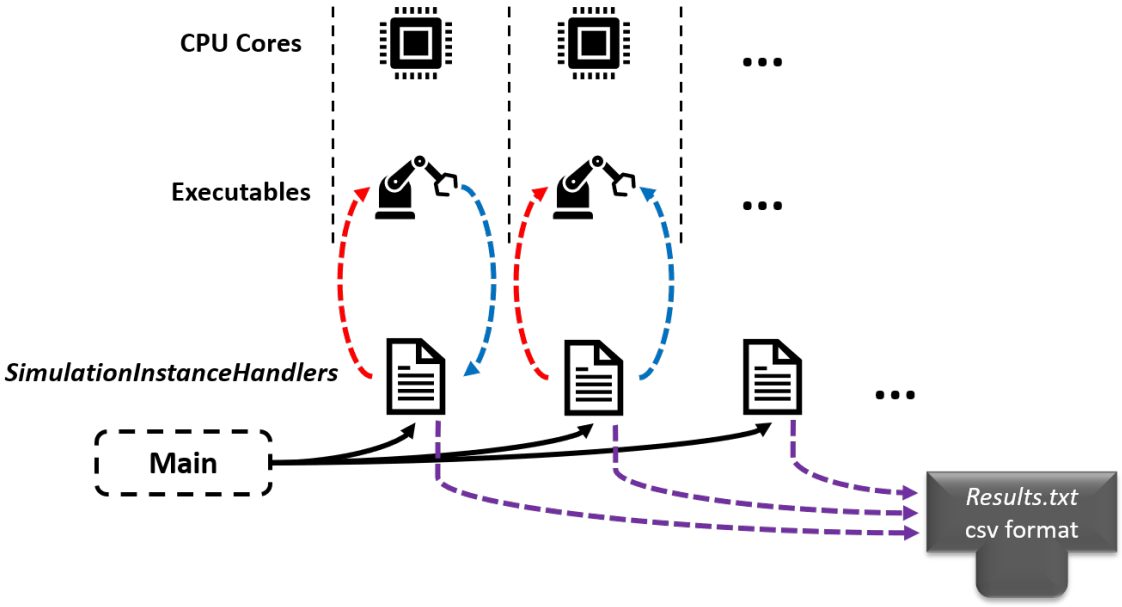

Fig. 6–6: Parallelization of communication with executables on CPU cores.

without overloading the hardware.

Asynchrony and TPL Utilization Going into more detail, each process of calling the executable from a *SimulationInstanceHandler* object, communicating *evaluation parameters* and *objective values* via two *pipes* and, when finished, disposing the executable process is packed into an *enclosed TPL task*. As soon as a CPU resource is free, that *task* is executed on a dedicated CPU core, thereby performing the following synchronous and asynchronous operations:

1. Create a *TPL task* to open a *pipe* to send *evaluation parameters* to the executable. Inside this *task*, asynchronously wait for a connection on the *pipe*,
   (a) When a connection is established, write each *evaluation parameter* into the *pipe* and dispose the *task*.
2. Create a *TPL task* to open a *pipe* to receive *objective values* from the executable. Inside this *task*, asynchronously wait for a connection on the *pipe*,
   (a) When a connection is established, read all *objective values* from the *pipe* and dispose the *task*.
3. Synchronously call the executable (while both *pipes* are asynchronously waiting for a connection) and block enclosing *task* execution until the executable process is disposed again,
4. Synchronously wait for the two *pipe tasks* to be disposed correctly,
5. Store results read from the *objective value pipe* and dispose enclosing *task*.

This combination of synchronous and asynchronous operations ensures that both communication *pipes* already wait for a connection when the executable itself is launched

and all data is transferred correctly. This design furthermore guarantees that each *task* is only disposed after the executable process and all sub-*tasks* were closed and the *objective values* were set aside. Lastly, representing each *pipe* by a separate *task* makes it easier to interrupt the *pipe*'s asynchronous waiting operations and successfully free all allocated memory in case the executable process is stuck and the execution has to be cancelled after a specified time limit is reached.

6.4 Point Cloud Visualization

Meaningful evaluation of a simulation model in the context of this evaluation framework strongly depends on an intuitively usable and highly informative visualization of the point clouds resulting form calculating *objective values* for all combinations of *evaluation parameters*. The evaluation framework's third component aims at achieving exactly this. Figure 6–7 shows a screenshot of the GUI developed in this thesis, which reads the point cloud results stored into a *.txt* file and visualizes them, adding some UI elements to allow the user to investigate the *evaluation space* thoroughly. The complete GUI was implemented in *Python tkinter*, parsing the data from the *.txt* file and preparing it for visualization is done with the help of *Python*'s *Pandas* library [76]. Generally, the GUI in Figure 6–7 can be split into two main parts, i.e. into two main *tkinter Frames* as introduced in Section 5.3. The upper *Frame* highlighted in red shows a 2D or 3D plot of the *evaluation space*, and the lower *Frame* depicts *evaluation parameters* and *objective values* alongside their range of values. Each point in the plot represents one point in the *evaluation space*. The plot shows axis labels for the currently selected parameters on the x- and y-axis. Furthermore, a third dimension of visualization is added by introducing a color bar, which colors the values inside the plot according to the values of a selected parameter. Additionally, a toolbar in the lower left corner of the upper *Frame* allows the user to zoom into the plot, drag the axes and save the plot as an image to the disk.
Each *widget* inside the two main *Frames* has a specific purpose in evaluating the simulation model and provides the user means to investigate the results in more detail. The *widgets* highlighted in Figure 6–7 and their functionality are summarized below:

- **Choose axis of plot** *(blue)*: Each of the plot's three dimensions mentioned above can represent any *evaluation parameter* or *objective value* in the *evaluation space*. Clicking on any of these elements opens a drop-down menu in which the user can select the desired parameter.

- **Limit parameter range** *(yellow)*: In the lower of the two main *Frames*, each *evaluation parameter* and *objective value* is listed. Besides its name, a slider depicts the range of values existing of the respective parameter in *evaluation space*. Both ends of the slider can be dragged left or right, thus limiting the range of the parameters that are displayed. The plot in the upper *Frame* only displays those points in color which lay between the left and right end of the sliders in the lower *Frame*. The residual points are greyed out.

- **Select a specific solution** *(violet)*: By clicking a point in the plot, the user can select it, thereby setting it as the currently selected solution. The point is then circumvented by a black circle (see violet arrow). Furthermore, the values of all

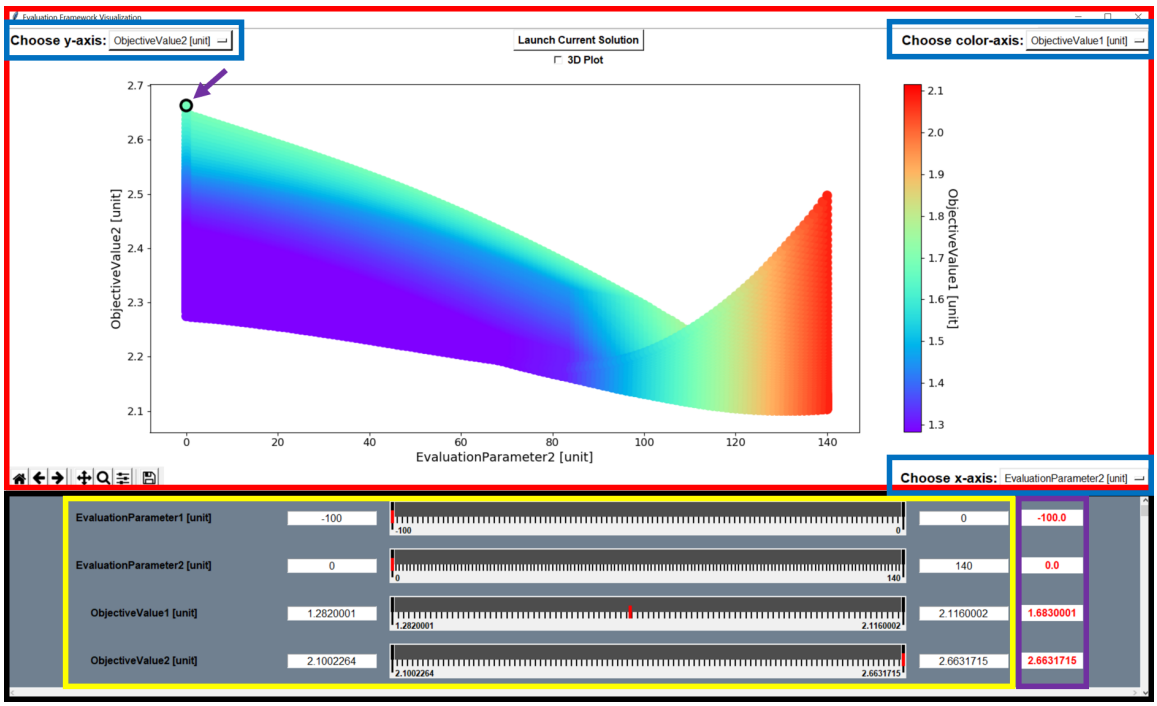

Fig. 6–7: Point cloud visualization with interactive UI elements.

evaluation parameters and *objective values* of the currently selected solution are displayed in red in the lower *Frame*. By clicking the ”Launch Current Solution” button in the top *Frame*, the executable is called once with the combination of *evaluation parameters* and executed visibly for the user.

All these UI elements were added to the overall GUI to improve its strength of visualizing and intuitively investigating the results of all evaluations. By choosing the axes of the plot, the user is able to identify connections and interdependencies between *objective values* and *evaluation parameters*. Furthermore, multiple *objective values* can be visualized against each other, thus revealing conflicts between *objective values* only solvable by choosing an appropriate trade off. Extending the plot by a color axis and adding the option to switch from a 2D- to a 3D-plot enables the user to compare multiple dimensions of the *evaluation space* at once. The slider *widgets* in the lower *Frame* in Figure 6–7 make it possible for the user to inspect, compare and evaluate only parts of the *evaluation space*, by limiting either *evaluation parameters* or *objective values*. This can be particularly useful when a specific range of *evaluation parameters*, or a specific range of resulting *objective values* is of interest for the user. Finally, selecting a solution in the plot, seeing its combination of *evaluation parameters* and its *objective values* and running a simulation with this set of parameters allows the user to inspect each simulation in detail. It is for these reasons that the design and structure of the GUI explained above were chosen, in order to visualize point clouds in the *evaluation space*.

7 Exemplary Applications and Evaluation Setup

This chapter covers two use cases demonstrating the application of the evaluation framework developed during this work. The focus lies on the creation of the simulation model of the *Panda* robot of *Franka Emika GmbH* being evaluated, the main ideas behind the two applications, their industrial context, outlines the setup of *Unity*, in which the *Panda* robot is simulated, and shows the *evaluation parameters* and *objective values* used for evaluation.
This chapter begins with Section 7.1 introducing a robot arm with seven DOF, namely the *Panda* robot of *Franka Emika GmbH*. The section furthermore presents how the *Panda* robot can be transformed into a digital twin in *Unity*, reflecting the real robot as accurately as possible. Afterwards, Section 7.2 explains the first of the two covered applications, which showcases how the framework can be used to evaluate trajectories of the *Panda* robot from different perspectives. Section 7.3 then presents the application of robotic bin picking with the same robot arm and how *ROS* can be consulted to extend the evaluation framework's capabilities in the context of this application.

7.1 Simulation Model of the Panda Arm

The robot simulated in the two applications of the evaluation framework is the *Panda* robot by *Franka Emika*. The robot is depicted in Figure 7–1. The *Panda* robot comprises seven revolute joints. It is capable of carrying a payload of three kilograms, which is as much as a quarter of the total moving mass of the robot. Sensing both forces and torques is supported at very high resolution, with over 100 sensors integrated into the *Panda* robot and, amongst others, accurate torque sensors in all seven axes. *ROS* supports loading, visualizing and controlling the robot through an open-source *ROS* package for the *Franka Emika Panda*, which includes *Panda*'s *URDF* file. The robot's specifications, such as *Denavit-Hartenberg* parameters, joint limits and maximum joint motor torques can be found in the documentation in the *Franka Emika* Github repository [77]. Due to its wide range of application in various industries, its open-source availability, and given the fact that *Franka Emika* is partner of a project at the *Chair of Astronautics* at *TUM*, *Panda* was chosen as a suitable robot arm for the work done in this thesis and is used as exemplary mechatronic system to demonstrate the evaluation framework. [78]

In order to model the *Panda* robot in *Unity* and simulating it in a physically correct simulation environment, besides the robot's kinematic parameters—i.e. its *Denavit-Hartenberg* parameters, lengths of links, etc.—dynamic parameters of the robot, such as masses of links and inertial properties have to be known. In the official *URDF* of the *Panda* robot, *Franka Emika* solely specifies kinematic parameters and provides no information on masses of links, positions of each link's center of mass (COM), and the robot's inertia tensor elements. In a paper by Gaz et al. [79] the authors developed an optimization algorithm to estimate the dynamic parameters of the *Franka Emika Panda* robot. As unofficially stated by *Franka Emika* on personal request, the results in

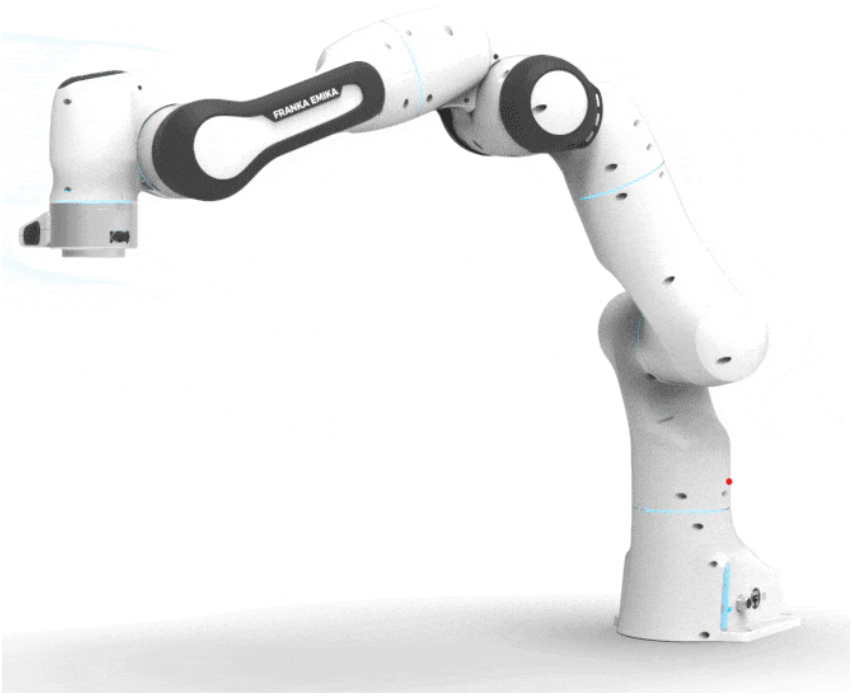

Fig. 7–1: *Franka Emika Panda* robot. (Source: [78])

obtained by Gaz et. al are close to the true dynamics of the robot arm, and hence are used in the remainder of this thesis to simulate the robot. Consequently, the *URDF* of the *Panda* robot was extended—as described in Section 5.4—by adding mass, inertia tensor and position of the COM as given by Gaz et al. in [79] to each of the robot's links. With the dynamic parameters identified, the robot can be evaluated in a physically realistic *Unity* simulation.

Importing the robot into a *Unity* scene is very easily achieved through *ROS#*'s *UrdfImporter* mentioned earlier in Section 5.5. Figure 7–2 depicts the *Panda* robot imported into a *Unity* scene, alongside its tree-like structure in *Unity*'s hierarchy window. The *UrdfImporter* takes care of the complete process of importing a robot from a *URDF* file and representing it properly in terms of *Unity*'s *GameObjects* and *Components*. As can be seen in this figure, the concept of parenting is extensively used to represent the real robot. The robot consists of seven links, represented by one *GameObject*, respectively. The following list illustrates how the *Unity Components* introduced in Section 5.1.1 are utilized in each link *GameObject* in this simulation model.

- **Rigidbody:** Each link has a *Rigidbody Component*, which holds the link's mass and inertia tensor as specified in the *URDF*.

- **MeshCollider:** In order to be able to collide with other objects, each link has *MeshCollider Component* (*BakedColliders* in Figure 7–2), which adds a complex Collider to each link, whose shape accurately matches the shape of the link.

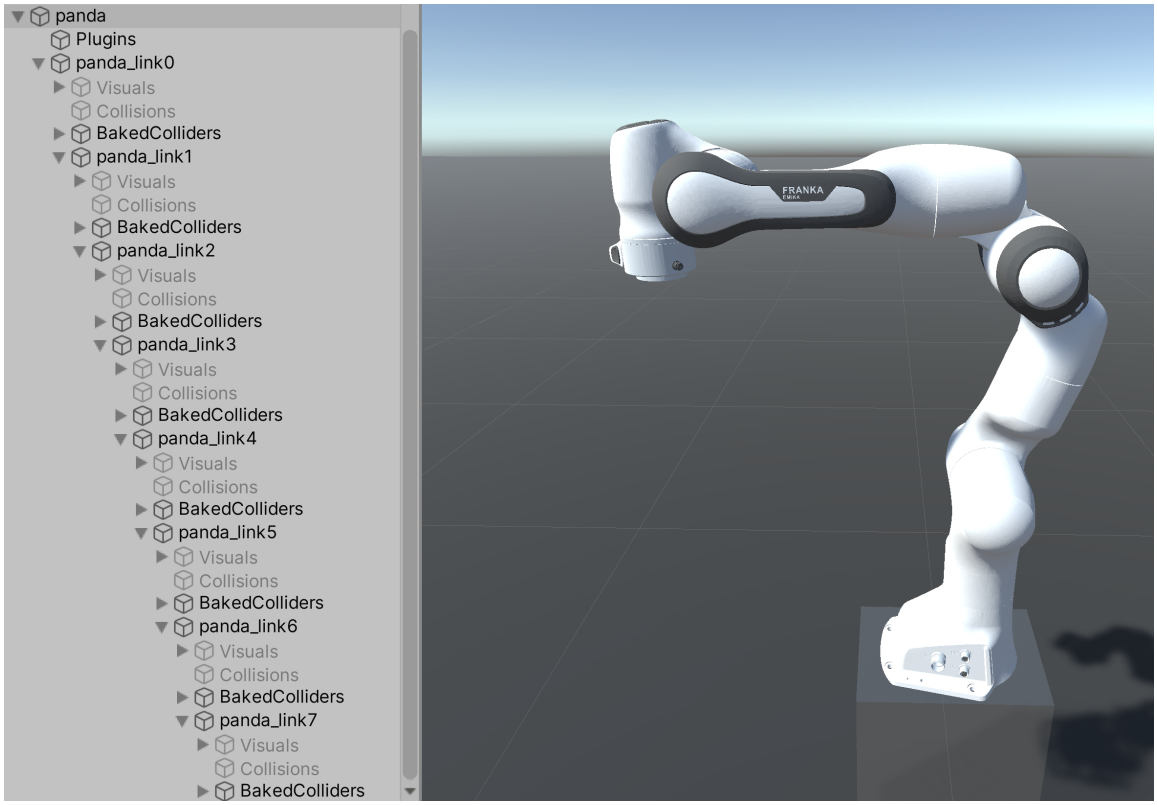

Fig. 7–2: Simulation model of the *Panda* robot in *Unity*.

- **HingeJoint:** Every link's *Rigidbody* is connected to the *Rigidbody* of its child *GameObject* through *Unity*'s *HingeJoint*. The joint's axis of rotation, the connected *Rigidbody* and joint position limits are specified as properties of the *HingeJoint*.
- **HingeJointMotor:** The motor of *Unity*'s *HingeJoint* is used to control the movement of each joint. It allows the specification of a maximum torque for each motor, which is directly extracted from the *URDF*. By setting a target velocity for each HingeJointMotor in the robot model, the *Panda* robot can be moved and controlled in a physically correct manner.

This robot model is used for both applications discussed in this chapter, and is the basis for the evaluation setup presented in the upcoming sections. Modelling the physics of the *Panda* robot arm was not initially considered a major issue in this thesis. However, while working on the two exemplary applications, creating and improving the simulation model of the robot turned out to be an important factor.

7.2 Evaluation of Trajectories with the Panda Arm

Today, robot arms are widely used in many different industries for e.g. assembling, welding, spraying, milling, polishing and packaging [80]. In all of these tasks, the robot arm has to move along different trajectories of different lengths and complexities, while following them as accurately as possible. To this end, simulating robot arms in physically realistic environments like *Unity* and evaluating them from different perspectives and

with different objectives can be of great value and provide important insights. The first application presented in the following evaluates the seven-DOF *Panda* robot arm in the form of comparing different trajectories from the perspective of, amongst others, execution time, energy efficiency and total Cartesian and joint distance. This section summarizes the process of generating robot trajectories from discrete waypoints, as well as how physical properties of the robot arm, such as inertia and mechanical energy, can be estimated from the system, in order to provide a physically-realistic simulation. This section closes by explaining the *evaluation space* being used for this application.

7.2.1 Parabolic Blending

Trajectories consists of waypoints parametrized by time. In order to ensure that the robot arm follows the trajectory, a method denoted as parabolic blending is used cf. Kunz and Stilman [81]. The paper shows how robot configurations on a path can be turned into a time-parametrized trajectory by using parabolic blending. Simply defining waypoints in the robot's configuration space and connecting them with straight lines results in discontinuities at the waypoints. The robot has non-zero mass and only finite torques can be applied to its motors. Hence, the robot is not able to instantaneously change its direction of motion to move across all waypoints on the path. The only way of following the path would be to add a complete stop of the robot at each waypoint, ultimately leading to slower, inconsistent motion along the path. [81]
As a solution, adding parabolic blends at the waypoints is one standard method commonly applied in robotics. Figure 7–3 illustrates the idea of approximating the straight line connecting two waypoints by parabolic functions around the waypoints. The algorithm was firstly introduced in [82] and [83] and builds the foundation for the work of Kunz and Stilman in [81]. The basic concept of parabolic blending will be briefly summarized below. The information provided about this concept is taken from [81], rather than directly from [82] and [83], hence represents second-hand information.

Hereinafter, the aforementioned waypoints in the configuration space $\mathcal{C}$ of the robot are denoted $q_i,\ i \in \{1...n\}$, with n being the total number of specified configurations on the path. In the case of the *Panda* robot, a configuration q_i is represented by a vector with seven entries, one for each joint position. Intuitively, connecting these waypoints to a path can be achieved by drawing straight line segments between them, as depicted in Figure 7–3. Now, a trajectory is a function q mapping any point in time between 0 and the total duration of the trajectory t_f to a robot configuration: $q\ :\ [0, t_f] \rightarrow \mathcal{C}$. [81] Two conditions further constrain the trajectory: (1) at the first and last waypoint the robot has zero velocity and the trajectory starts and ends, respectively, at exactly these waypoints; (2) during the trajectory, both velocity and acceleration constraints of the robot have to be satisfied: $\forall t : |\dot{q}(t)| \leq v_{max} \wedge |\ddot{q}(t)| \leq a_{max}$. [81] It is important to note that the algorithm does not guarantee that each waypoint on the path is exactly reached by the robot. That being said, the trajectory between two waypoints consists of two phases: a linear phase during which the robot follows the straight line segment between the two configuration (see Figure 7–3) and a parabolic blend phase. During the linear phase, the robot moves with zero acceleration, hence at constant velocity and linearly changing its position in time. When entering the parabolic blending phase, the robot switches to

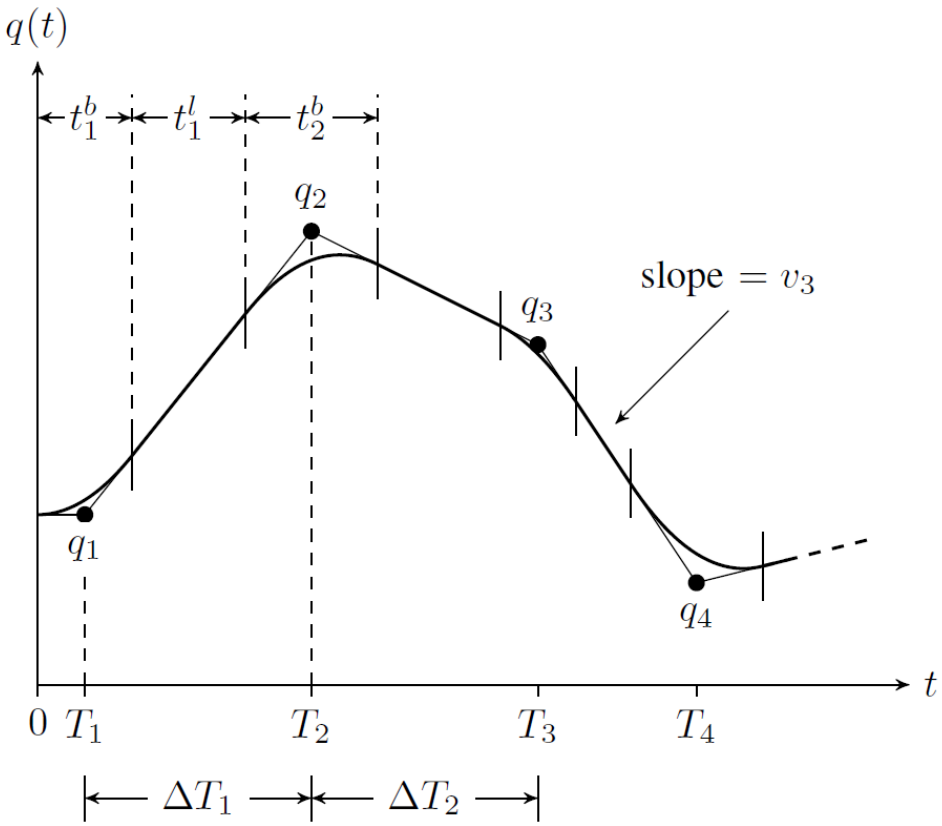

Fig. 7–3: One-dimensional example of parabolic blending at waypoints on a path. (Source: [81])

constant acceleration, and hence its configuration changes quadratically in time, as can be seen in Figure 7–3. The algorithm as presented in [82] and [83] assumes that the timing of all waypoints is known. Thus, the time ΔT_i the robot needs to move between two waypoints $q_i,\ q_{i+1}$, as well as the duration t_i^b of the blending phase at waypoint i are given. Knowing these timings, the constant velocity of the robot during linear phase between q_i and q_{i+1} can be calculated as follows:

$$v_i = \frac{q_{i+1} - q_i}{\Delta T_i}. \tag{7–1}$$

Since moving with this velocity along a straight line during the whole time interval would lead to discontinuous velocities at each waypoint, parts of two neighbouring linear segments are replaced by a parabola. This parabola is followed by the robot with constant acceleration

$$a_i = \frac{v_i - v_{i-1}}{t_i^b}. \tag{7–2}$$

An important condition is given by Equation 7–3, which ensures that two blend phases of neighbouring waypoints do not overlap each other:

$$t_i^b + t_{i+1}^b \leq 2\Delta T_i. \tag{7–3}$$

After introducing the blend phases, the robot starts following the straight line segment between waypoints q_i and q_{i+1} at $T_i + \frac{t_i^b}{2}$ and enters the subsequent blend phase around q_{i+1} at $T_{i+1} - \frac{t_{i+1}^b}{2}$.
As can be seen in Figure 7–3, the trajectory does not start at $t = T_0$ and eventually does not end at $t = T_n$. The first and last waypoints can be treated exactly the same way each waypoint is considered, by pretending that the first waypoint is preceeded by a linear line segment with velocity zero, and accordingly such a line segment follows the last waypoint. As a result, both the first and last waypoint now have two neighbouring linear phases. To this end, the trajectory has to start $\frac{t_0^b}{2}$ before the first waypoint and $\frac{t_n^b}{2}$ after the last waypoint, in order to fully include the first and last blend phase in the trajectory. Consequently, Equation 7–4 states the total duration of the trajectory from waypoint 0 to waypoint n.

$$t_f = \frac{t_0^b}{2} + \sum_{i=1}^{n} \Delta T_i + \frac{t_n^b}{2} \tag{7–4}$$

Following this process, the final trajectory $q : [0, t_f] \to \mathcal{C}$ is given by

$$q(t) = \begin{cases} q_i + v_{i-1}(t - T_i) + \frac{1}{2}a_i(t - T_i + \frac{t_i^b}{2})^2, \\ \qquad \text{if } T_i - \frac{t_i^b}{2} \leq t \leq T_i + \frac{t_i^b}{2}, \quad i \in \{1...n\}, \\ q_i + v_i(t - T_i), \\ \qquad \text{if } T_i + \frac{t_i^b}{2} \leq t \leq T_{i+1} - \frac{t_{i+1}^b}{2}, \quad i \in \{1...n-1\}, \end{cases} \tag{7–5}$$

with $v_0 = v_{n+1} = 0$, thus satisfying the previously mentioned condition of starting and finishing the trajectory at rest. Now, linear phases are given by the second part of Equation 7–5, whereas robot configurations according to the first part of Equation 7–5 correspond to blend phases. By explicitly calculating the time derivatives of the trajectory given by Equation 7–5, velocities and accelerations of the robot on the trajectory are obtained. [81]
As explained earlier, the trajectory above can only be generated when the timings of the waypoints and parabolic blending phases are known. Generally, however, in robotics applications these timings are not known to the developer and hence have to be determined on-the-fly. Kunz and Stilman [81] propose a method of obtaining these values automatically, while still satisfying the aforementioned velocity and acceleration constraints of the robot. In their paper [81], the authors present an iterative method of choosing timings of the waypoints, as well as durations of each parabolic blend phase. Firstly, the timings of the waypoints are chosen in such way that the joint velocities are maximized:

$$\Delta T_i = \max_j \frac{|q_{i+1}[j] - q_i[j]|}{v_{max}}. \tag{7–6}$$

In Equation 7–6, $[j]$ refers to the j-th component of the joint configuration q. Furthermore, maximization of joint velocities in this context refers to calculating the timings of the waypoints such that at least one joint moves at its maximum velocity. [81] Secondly, the durations of the blend phases t_i^b are chosen likewise, such that at least one joint moves

with maximum acceleration during each blend phase:

$$t_i^b = \max_j \frac{|v_i[j] - v_{i-1}[j]|}{a_{max}} \tag{7–7}$$

Velocity and acceleration limits of all joints of the *Franka Emika Panda* robot can be found in [77].

While this idea seems straightforward, Kunz and Stilman point out that it is not guaranteed that the condition given by Equation 7–3 is satisfied when choosing ΔT_i and t_i^b in that way. Hence, they propose a method to iteratively adapt the length of blend phases and waypoint timings until no neighbouring blend phases overlap. [81] Due to the simplicity of trajectories generated in the application in this thesis, however, the first iteration given by Equations 7–6 and 7–7 already fulfils this condition. To this end, the aforementioned iterative approach of Kunz and Stilman will not be further discussed here. The interested reader is referred to Chapter 4 in their publication [81].

7.2.2 Estimation of the Electric Energy Consumption

Being able to meaningfully compare and evaluate the robot's energy consumption on different trajectories requires to know the electric power fed into every joint motor. This section presents how this motor power consumption is calculated for the simulation model of the *Panda* robot. Since the technical details beyond datasheet information are not known for the joints of the *Panda* robot, several model assumptions on the robot's motors and their power consumption are made and highlighted accordingly. However, the objective of this application is to qualitatively evaluate the electric energy consumption of the robot on different trajectories. In order to compare them and identify the best solutions, the actual values required to drive each motor are of minor interest, which is why we presume that, even if the the given model assumptions might deviate from the actual energy consumption values in reality, the identified *Pareto*-optimal solutions are sufficiently accurate.
As it is common in industrial robots, it is assumed that the *Panda* robot uses brushless direct current (BLDC) motors in all its joints. Motor specific constants required to calculate the electric power are taken representatively from motor specifications of a robot arm designed by the *Siemens AG* in Munich.

Equations of Motion The equations of motion of a robot are generally represented by Equation 7–8 and can be derived from the *Newton-Euler* or *Lagrangian* formulation of robot dynamics: [84]

$$\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q). \tag{7–8}$$

In Equation 7–8, τ refers to the joint-side torques of the robot, $q,\ \dot{q},\ \ddot{q}$ represent the angular position, velocity and acceleration, respectively. $M(q)$ is the so-called mass matrix in $\mathcal{C}$-space. It is important to note, that the mass matrix $M(q)$ is positive definite and symmetric. Off-diagonal terms in this matrix reflect the inertia coupling between robot joints, i.e. acceleration of one joint influences other joint-side torques through these terms. $C(q, \dot{q})$ represents centrifugal and *Coriolis* terms acting on the robot, and

$g(q)$ describes the gravitational forces. [84] In the case of the *Panda* robot, it holds

$$\tau,\ q,\ \dot{q},\ \ddot{q} \in \mathbb{R}^{7x1}. \tag{7–9}$$

Based on the joint torques, the energy consumption of a joint motor of the *Panda* robot can be calculated. With the positions, velocities and accelerations of each joint known at any point in time in the simulation, the corresponding joint torques τ have to be determined. This problem, i.e. determining the joint torques required to achieve a desired motion, is referred to as inverse dynamics [85].
The Recursive Newton-Euler Algorithm *(*RNEA*)* [85] provides a computationally efficient means for solving the inverse dynamics problem, specifically. For multi-body systems with n DOF, the *RNEA*'s time complexity is linear in n, i.e. $O(n)$ [85]. The *RNEA* comprises two steps: forward and backward recursion. In the forward recursion, kinematic variables and momentum changes of the robot are calculated for all joints, originating from the robot's root and recursively going outwards towards the end effector. Subsequently, the backward recursion step begins at the end effector and recursively calculates each joint's forces and torques, based on the values obtained during *forward recursion*. [85] Since *RNEA* is state of the art for solving inverse dynamics problems, the underlying formulas will not be provided here. For more information, the interested reader is referred to e.g. [85].
The *RNEA* was implemented in *Unity* for the *Panda* robot, thereby falling back to the dynamic parameters of each link taken from Gaz et al. [79]. At each point in time, i.e. after each physics engine update step, the joint-side torques τ are obtained by means of the *RNEA* and used for calculating each joint motor's electric energy consumption. In the following, the superscript j refers to the joint being considered, i.e. refers to the subsystem of links rotating around joint j. All following motor calculations are taken from [86].

Electric Power of a BLDC Motor The electric power $P^j_{el}(t)$ required to drive the motor of joint j is determined by Equation 7–10.

$$P^j_{el}(t) = P^j_{loss}(t) + P^j_{motor}(t). \tag{7–10}$$

$P^j_{loss}(t)$ is the power lost due to the motor current $I^j_{motor}(t)$ flowing through the terminal resistance R_T:

$$P^j_{loss}(t) = R_T * I^j_{motor}(t)^2. \tag{7–11}$$

The terminal resistance usually lies below one ohm, and in this model is assumed to be $R_T = 0.22\ \Omega$. Energy losses through the electric inverter and power supply itself are neglected in this model.
$P^j_{motor}(t)$ is the motor power transformed into mechanical power by the joint motor, in order to move the robot. The mechanical power can be calculated as product of the joint-side torque as given by Equation 7–8 and the joint's angular velocity:

$$P^j_{mech}(t) = \tau^j(t) * \omega^j(t). \tag{7–12}$$

The corresponding motor power required to generate said mechanical power requires to know the motor speed and gear ratio, and additionally considers gear and motor

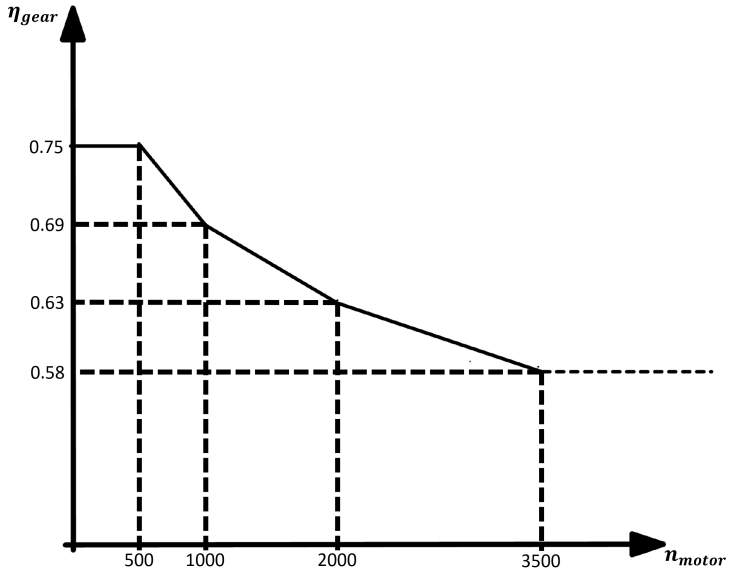

Fig. 7–4: Relation between gear efficiency and motor speed.

efficiencies. Typically, robot arms such as the *Panda* robot use harmonic gearing. Following this, the joint rotates at a given joint velocity $\omega^j(t)$, which is related to the motor speed through the gear ratio $R = 100$:

$$n^j_{motor}(t) = \omega^j(t) * \frac{60}{2\pi} * R. \tag{7–13}$$

The gear efficiency itself is a function of the motor's speed. Generally it holds, the higher the revolutions per minute (rpm), the lower the gear efficiency:

$$\begin{aligned} &\eta_{gear}(n_{motor} \leq 500) = 0.75 \\ &\eta_{gear}(n_{motor} = 1000) = 0.69 \\ &\eta_{gear}(n_{motor} = 2000) = 0.63, \\ &\eta_{gear}(n_{motor} = 3000) = 0.58. \end{aligned} \tag{7–14}$$

Between these corner points specified in Equation 7–14, the gear efficiency is assumed to decrease linearly. Figure 7–4 depicts this relation between gear efficiency and motor speed graphically. The values in Equation 7–14 are taken from the same robot arm of *Siemens* mentioned above.
Now, for the calculation of the motor-side torque required to generate a specific joint-side torque, two cases have to be distinguished:

$$\tau^j_{motor}(t) = \begin{cases} \frac{1}{\eta_{gear} * \eta_{motor}} * \frac{\tau^j(t)}{R}, & \text{if the motor is accelerating,} \\ \eta_{gear} * \eta_{motor} * \frac{\tau^j(t)}{R}, & \text{if the motor is recuperating,} \end{cases} \tag{7–15}$$

with the constant motor efficiency $\eta_{motor} = 0.92$. Thus, the total motor power can be calculated as follows:

$$P^j_{motor}(t) = \tau^j_{motor}(t) * 2\pi * n^j_{motor}(t). \tag{7–16}$$

In order to calculate the power loss, the motor current $I^j_{motor}(t)$ is required, which is proportional to the motor-side torque:

$$\tau^j_{motor}(t) \sim I^j_{motor}(t). \tag{7–17}$$

Using this relation, motor torque and current can be linked by a proportionality constant referred to as *torque constant* k_M, which is, like the other motor specific variables, taken from the *Siemens* robot and assumed to be constant at $k_M = 0.134\ \frac{Nm}{A}$. Generally, this torque constant k_M is dependent on the ambient temperature. This dependency, however, is neglected in these calculations, assuming that the accuracy of the resulting energy suffices in the context of this application. It follows, that

$$\tau^j_{motor}(t) = k_M * I^j_{motor}(t) \rightarrow I^j_{motor}(t) = \frac{\tau^j_{motor}(t)}{k_M}. \tag{7–18}$$

Thus, the total electric power supplied to the joint motor results to

$$P^j_{el}(t) = P^j_{loss}(t) + P^j_{motor}(t) = R_T * I^{j^2}_{motor}(t) + P^j_{motor}(t). \tag{7–19}$$

Recuperation in the joint motor reflects in the second summand of Equation 7–19, that is, the motor power. The mechanical power of the robot, and hence the motor power, becomes negative, if the joint-side torque $\tau^j(t)$ and the joint's angular velocity $\omega^j(t)$ point in opposite directions. In this case, the motor is braking and the negative mechanical power is transformed into electric power and fed back into the motor. To this end, negative mechanical power leads to negative electric power, and hence, corresponds to recuperation.

Electric Energy Finally, the change in electric energy consumed by the motor of joint j at time t over the time interval Δt is

$$\Delta E^j_{el}(t) = P^j_{el}(t) * \Delta t. \tag{7–20}$$

Summing this change in electric energy for all time steps results in the total energy consumption of each motor. Additionally, the sum over all joint motors results in the total energy consumption of the *Panda* robot on a trajectory.

7.2.3 Evaluation Setup

The main idea of this application is to evaluate properties of different robot trajectories. As a consequence, these different trajectories have to be generated in some way, and then simulated in the form of the *Panda* robot moving along each trajectory.
Generating trajectories in *Unity* is done according to the concept of parabolic blending discussed in Section 7.2.1. Thus, multiple robot configurations q_i, representing a path in the *Panda* robot's configuration space $\mathcal{C}$ were defined and turned into a trajectory following Equation 7–5. In total, all trajectories evaluated during the simulations comprise three robot configurations $q_1,\ q_2,\ q_3$, representing the start, intermediate and goal configuration of the robot on the trajectory, respectively. Furthermore, the start and goal configurations q_1 and q_3 are kept constant for all evaluated trajectories, while solely the

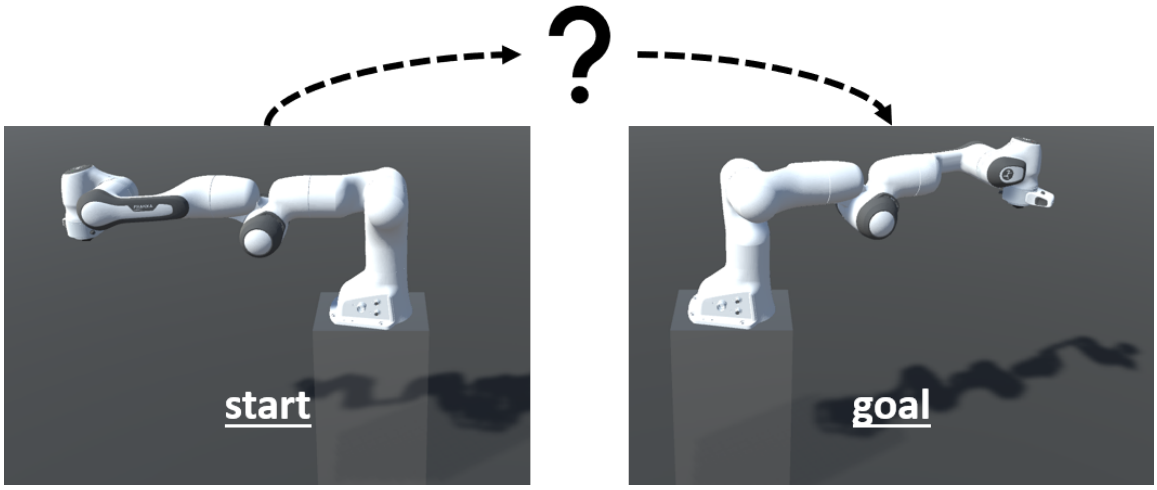

Fig. 7–5: Start, intermediate and goal configurations on the evaluated robot trajectories.

intermediate joint position q_2 is modified.

Figure 7–5 visualizes exactly this approach to generating different trajectories. Both q_1 and q_3 represent a configuration where the robot arm is fully extended horizontally. Their only difference lies in the angle of the second joint, which simply has to rotate $180\,^\circ$ for the robot to move from q_1 to q_3. Therefore, the start and goal configurations are selected as follows:

$$q_n = (q_n[1] \quad 90\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 90\,^\circ \quad 0\,^\circ),\ n \in \{1, 3\}, \tag{7–21}$$

with $q_1[1] = 0\,^\circ$ and $q_3[1] = -180\,^\circ$. Joint six was additionally rotated by $90\,^\circ$, in order to align the end effector of the robot along the vertical, just as if the robot was carrying an object. It is important to note that in *Unity*, the initial configuration corresponding to q_1 in Equation 7–21 is automatically set as a configuration with all $0\,^\circ$ angles. Thus, Equation 7–22 will be used in the remainder of this thesis as the start and goal configurations of the *Panda* robot.

$$q_n = (q_n[1] \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ),\ n \in \{1, 3\}, \tag{7–22}$$

with $q_1[1] = 0\,^\circ$ and $q_3[1] = -180\,^\circ$. This kind of robot motion is inspired by the idea of a human picking up a cup of coffee on a table to his or her left, and putting the cup down on a table to the right, only using one arm. Both tables, however, are in such distance to the human, that the cup can only be reached and released with a fully extended arm. Imagining this scenario, the human now has multiple options to move the arm between the two moments of picking up and putting down the cup: keeping the arm stretched out and simply rotating from the shoulder, moving the cup close to the body by additionally bending the elbow, and many others. Likewise, the *Panda* robot can move along various trajectories, in order to rotate from q_1 to q_3. These different possibilities are represented by the intermediate configuration q_2.
Following this approach, in this application the *evaluation parameters* represent the different intermediate configurations q_2. In order to keep the evaluation as simple as

possible, but still generate an interesting variety of trajectories for the *Panda* robot, only two joint angles were changed from q_1 and q_3 to q_2. Furthermore, the second joint angle of q_2 was chosen such that the configuration actually represents the intermediate joint position on the trajectory. In other words, the second joint angle of q_2 lies exactly in the middle of $q_1[1] = 0\,°$ and $q_3[1] = -180\,°$, where $q_i[j]$ represents the j-th component of the configuration vector. Equation 7–23 depicts this intermediate configuration q_2.

$$q_2 = (-90\,° \quad q_2[2] \quad 0\,° \quad q_2[4] \quad 0\,° \quad 0\,° \quad 0\,°) \tag{7–23}$$

The two joint angles differing from both q_1 and q_3 (neglecting the constant angle of the first joint $q_2[1] = -90\,°$) are the second and fourth joint angle. Falling back to the comparison of this application to a human lifting and moving a cup of coffee, the second joint angle can be thought of as the shoulder of the robot, while the fourth joint angle corresponds to its elbow. The specification of $q_2[2]$ and $q_2[4]$ as *evaluation parameters* are given below.

$$q_2[2] \in [-100\,°,\ 20\,°], \quad q_2[4] \in [0\,°,\ 140\,°], \quad s_{2/4} = 1\,° \tag{7–24}$$

In Equation 7–24, $s_{2/4}$ represents the step size for both *evaluation parameters*, which determines how each parameter is increased each *evaluation step*. The border cases of these two *evaluation parameters* can be thought of as follows:

- $q_2[2] = -100\,°,\ q_2[4] = 0\,°$: Fully extended, vertically aligned robot arm.
- $q_2[2] = 0\,°,\ q_2[4] = 140\,°$: The shoulder is kept steady, while the elbow bends downwards $140\,°$.
- $q_2[2] = -0\circ,\ q_2[4] = 0\,°$: The robot arm remains horizontally and fully extended and simply rotates around the second joint.
- $q_2[2] = -100\,°,\ q_2[4] = 140\,°$: Both joints move. The second joint lifts the robot arm up, while the fourth joint bends the elbow downwards.

Combining both these parameters in Equation 7–24, the—in this case 2D—grid spanned over the *evaluation space* contains $121 * 141 = 17061$ combinations of *evaluation parameters* in total.

What is missing in order to actually evaluate these trajectories is the specification of *objective values*. As a reminder, *objective values* represent properties of the mechatronic system or the application itself, whose values and their relation to different *evaluation parameters* the user is interested in. In this application, the main objective is to investigate the energy efficiency of the motion of the *Panda* robot on all trajectories. This includes, amongst others, the energy consumption of each joint motor in total. Besides that, the execution time of each trajectory, as well as the distance the robot covers in both Cartesian and joint space will be examined. While many other *objective values* can be of interest in this kind of evaluation, such as maximum joint velocities and accelerations on the trajectories, or peak torques in the joints, the *objective values* mentioned above comprise the main evaluation criteria of this application.
The results of this application are presented in Section 8.1.

7.3 Automated Robotic Bin Picking with the Panda Arm

The second application presented in this thesis deals with robotic bin picking in an industrial plant. The idea of this application is part of the *SAINT* project of *Franka Emika GmbH* and *FIEGE Logistik Stiftung & Co. KG*, in cooperation with the *Chair of Astronautics* and *Chair of Applied Mechanics* at *TUM*. With bin picking being a popular field of use of robot arms in industrial applications, the motivation of *SAINT* is to realize bin picking in unknown environments, while satisfying high reliability requirements of robotic grasping. [87] The two cooperating chairs a *TUM* aim at improving existing path planning and computer vision algorithms, in order to equal the efficiency of humans in bin picking and achieve a low error rate at the same time. Furthermore, the concept of supervised autonomy in the *SAINT* project is realized through a fault recovery and teleoperation module. With the former, the robot is able to autonomously avoid faulty configurations and recover from them, if necessary. When the robot is stuck in a bad condition in which the fault recovery module cannot find a way to recover a valid state of the robot, the teleoperation module allows a human operator to manually move the robot into a valid configuration. [87]

The industrial plant, in which the robot operates, is depicted in Figure 7–6. As can be seen in this figure, *Franka Emika*'s *Panda* robot is used as robot manipulator in the *SAINT* project. Operating in this industrial environment, the objective of the robot is to detect objects in boxes—in the case of *SAINT* these objects are pieces of clothing packed in plastic bags—by applying machine vision algorithms, autonomously grasp these objects and place them into a specific area on a conveyor. At the same time, it should be possible for a human to safely work in the same environment alongside the robot. The teleoperation module allows another human to recover valid states of the robot by controlling it over a screen via a joystick.
In the context of the *SAINT* project, the application presented in this thesis models the industrial plant and the *Panda* robot in *Unity* and simulates the process of picking up an object, moving the robot arm above the conveyor and dropping the object at the correct position. The goal of this application is to evaluate the bin picking process for different positions of the *Panda* robot inside the industrial plant. For simplicity, automatically detecting objects through computer vision is neglected in this application. Furthermore, the fault recovery and teleoperation modules are not integrated into the simulation. However, the evaluation of this application with the evaluation framework was designed in such way that these three components of *SAINT* can be easily integrated into the simulation and evaluation in the future. The clothing object picked up by the robot is simulated in *Unity* by a pendulum-like *GameObject* hanging from the *Panda* gripper. Most of the weight of the simulated object is positioned at the end of the pendulum ($250\ g$), while a small weight represents the stick of the pendulum ($20\ g$). Thus, the total weight of the clothing object is assumed to be $270\ g$ and can swing freely around its anchor point at the *Panda* gripper.

The upcoming sections discuss how trajectories for bin picking are generated in this application, and how, accordingly, the evaluation framework was extended with a *ROS* interface, in order to generate those trajectories and incorporate components and

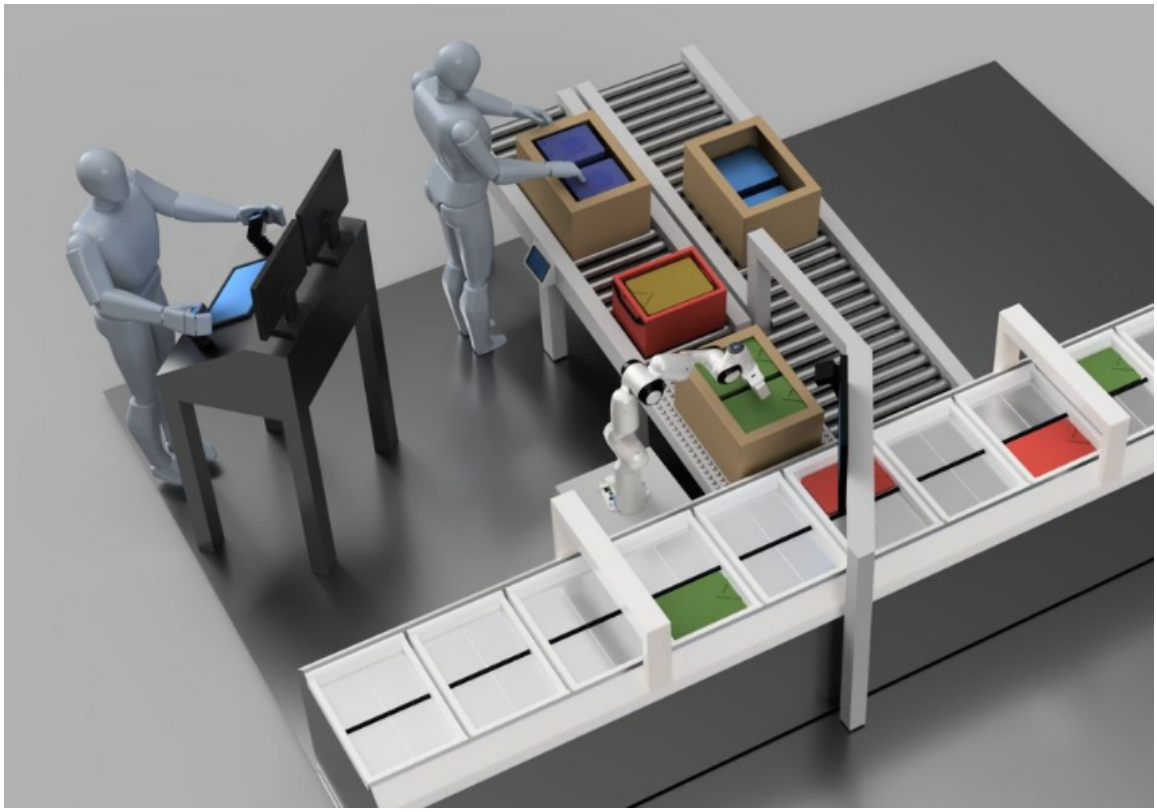

Fig. 7–6: Industrial plant of the *SAINT* project. (Source: [87])

concepts of *SAINT* into the evaluation of the *Panda* robot in the industrial plant.

7.3.1 Generation of Bin Picking Trajectories with MoveIt!

As an entry point for evaluating a simulation model of the *Panda* robot in the *SAINT* environment, this application aims at simulating the bin picking process of the robot. Since such a bin picking operation requires the robot to move along complex trajectories, the idea of generating trajectories from distinct robot configurations as described in Section 7.2.1 is likely not sufficient to realize bin picking in the industrial plant. To this end, another solution of generating trajectories for the *Panda* robot was exploited. In order to provide a basis for further extensive evaluation of concepts of *SAINT*, the application was designed in such way that the results obtained from evaluating the system provide meaningful insights, while at the same time the application can be easily extended to integrate more components of *SAINT* in the future.
In the *SAINT* project, the path planning algorithms developed at *TUM*'s *Chair of Applied Mechanics* are based on *MoveIt!*, the *ROS* Motion Planning framework (see Section 5.4.1). As a result, it was decided to generate the trajectories for robotic bin picking in the industrial plant for this application by motion planning algorithms of *MoveIt!* directly. This enables the user to evaluate the robotic bin picking process based on trajectories generated with *MoveIt!*'s in a first step, and later on exchange the integration of *MoveIt!* by the motion planning algorithms developed at the chair.

MoveIt! supports generating trajectories from a series of waypoints in Cartesian space.

In this application, the robotic bin picking was represented by seven distinct Cartesian positions $p_i,\ i \in \{1...7\}$ as listed below.

1. p_1: Position above box containing the object to pick up
2. p_2: Position in box when picking up the object, vertically below p_1
3. p_3: Position above box with object in gripper ($p_3 = p_1$)
4. p_4: Position above conveyor where the object should be placed
5. p_5: Position close to the conveyor to release the object, vertically below p_4
6. p_6: Position above conveyor after releasing the object ($p_6 = p_4$)
7. p_7: Position above box to pick up the next object ($p_7 = p_3 = p_1$)

Providing these waypoints to the *MoveIt!* planning library via the *ROS* service (see Section 5.4) *GetCartesianPathRequest* results in a *GetCartesianPathResponse*, which contains the trajectory along these waypoints in the robot's joint space. The result of the Cartesian path planning is a list of joint positions, velocities and accelerations on the trajectory, as well as the timings of each of these configurations. Besides that, the *GetCartesianPathResponse* contains a number representing the percentage of the trajectory *MoveIt!* was actually able to plan for the given waypoints, without colliding with obstacles or leaving the robot's workspace.
For the simulation in *Unity*, a detailed model of the whole industrial plant in form of a CAD file was already available. However, generating the trajectories for bin picking in *MoveIt!* required to model those parts of the plant that are potential obstacles during bin picking. By adding these obstacles to the *MoveIt!* planning scene, the motion planning algorithm factors these obstacles into the planning process and tries to return a trajectory that does not collide with any of them.

Now, to simulate the bin picking in *Unity*, the generated trajectory has to be communicated to the *Unity* executable. The next section explains how the evaluation framework as discussed in Chapter 6 is extended, in order to be able to communicate with *ROS*.

7.3.2 Integration of ROS Interface

Varying the robot's positioning inside the industrial plant requires to re-plan the bin picking trajectory every time, since the seven Cartesian waypoints and all obstacles on the trajectory depend on the robot's position. Hence, for each socket position of the *Panda* robot, *MoveIt!* must plan a new trajectory. In Section 5.5, *ROS#* was introduced as a library, providing tools to communicate between *ROS* and *.NET* applications, specifically *Unity* simulations. *ROS#* was primarily developed to communicate with *ROS* programs directly from *Unity*, but also contains *C#* libraries for integration of *ROS#* into independent *.NET* solutions. Thus, two possibilities for designing the integration of *ROS* and *MoveIt!* into the evaluation framework by means of *ROS#* were examined:

- Establishing a connection to the *ROS* system originating directly from each *Unity* executable called during the evaluation process of a simulation model. Using this

method, *ROS#* sets up a connection between the *Unity* executable and *ROS* once for each robot's socket position. Each connection is then used to plan a trajectory in *MoveIt!* corresponding to the current socket position, and to communicate this trajectory directly back to the *Unity* executable.

- Connecting the *.NET* solution of the evaluation framework's execution and evaluation component to the *ROS* system **once** before entering the main loop of evaluating a simulation model. With this concept, *ROS#* only establishes a connection to the *ROS* program once. Thus, *MoveIt!* successively plans all trajectories corresponding to each position of the robot that the user wants to simulate. Afterwards, the execution and evaluation component of the framework is responsible for communicating the respective trajectory to each executable.

One major drawback of the first method is its slow performance and communication overhead. Individually connecting each executable to *ROS* means that for each simulation, *ROS#* has to establish a connection to the *ROS* system via *rosbridge_suite* (see Section 5.5) and send the current socket position of the robot. *MoveIt!* subsequently plans a trajectory for said socket position and returns it to the executable. Finally, *ROS#* must perform operations to close the connection again. While the process of connecting to *ROS* via *rosbridge_suite* and afterwards closing the connection is rather quick, experience has shown that repeating this process during the evaluation of thousands of simulations does increase the duration of the total evaluation drastically. Besides that, connecting to *MoveIt!* directly from inside each *Unity* simulation results in multiple calls at the same time, since one executable is run on each CPU core. While multiple clients can easily connect to a *ROS* system via *ROS#*, it would be cumbersome to deal with multiple, independent planning instances of *MoveIt!*. As a result, the second approach was chosen.

The following steps describe how the evaluation framework's execution and evaluation component is modified and extended to realize these requirements. In order to keep the *ROS* integration as generic as possible, the extension was structured in such way, that the combination of the evaluation framework and *ROS* can easily be deactivated or extended even further.

1. **Read data:** As described in Sections 6.2 and 6.3, the component initially reads the *evaluation space* from *.json* files. Defining the *evaluation space* in *Unity* is extended to include the definition of an additional set of parameters, denoted ROS *parameters*. Selecting these parameters is implemented and handled in exactly the same way as *evaluation parameters* and *objective values* are. These ROS *parameters* represent the parameters used inside the *Unity* simulation, originating from a *ROS* system, in this case originating from *MoveIt!*. Consequently, the *EvaluationData* class of the execution and evaluation component also reads these parameters from the corresponding *.json* file.

2. **Establish connection to *ROS*:** Only once, after parsing the *.json* files, the framework component connects to the *ROS* system running on a virtual machine with the Ubuntu OS. On the virtual machine, one main *ROS* node runs the *MoveIt!*

planning library.

3. **Generate trajectories:** Subsequently, the execution and evaluation component sends all combinations of *evaluation parameters* (only those parameters that affect the trajectory planning in *MoveIt!*) to *ROS*. Thereafter, *MoveIt!* successively generates all trajectories based on the received *evaluation parameters*, and returns them to the *.NET* solution as the aforementioned ROS *parameters*. Afterwards, the connection between *ROS* and the *.NET* solution is immediately closed.
4. **Evaluate simulations and write results:** From then on, the process of evaluating all simulations is equivalent to the process discussed in Section 6.3, with only the difference that now an additional pipe for the ROS *parameters* is opened for the interprocess communication between *.NET* and executables, thus resulting in three pipes in total.

Thus, only small adjustments to the evaluation framework had to be made, while only very few new code had to be implemented, since most code required for the integration of *ROS* could be re-used from the respective components of the evaluation framework.

7.3.3 Evaluation Setup

Robotic bin picking with *Franka Emika*'s *Panda* manipulator is the core idea of this application. Correspondingly, this section presents the definition of the *evaluation space* in which the *Panda* robot and the process of bin picking are evaluated. The idea of this evaluation is to examine how different positions of the *Panda* robot between the box and the conveyor (see Figure 7–6) influence the robot's ability to carry out the complete bin picking process. Mounting the robot too close to or too far away from either the box or the conveyor might result in the robot not being able to reach the object in the box or to drop the object onto the conveyor. Besides that, different socket positions of the robot could increase the chance of collisions along the trajectory. Additionally, things like the relation between energy efficiency of bin picking and the robot's positioning, or examining connections between different box geometries and the robot's ability to reach the object inside the box can be evaluated.

As already mentioned, based on a CAD file accurately modelling the industrial plant, the proper environment for the *Panda* robot to perform bin picking could easily be imported into a *Unity* simulation. The robot's origin, i.e. zero position, corresponds to the socket position of the robot as depicted in Figure 7–6. Based on that origin, the seven Cartesian positions $p_i,\ i \in 1...7$ as presented in Section 7.3.1, as well as the positions of box, conveyor and other obstacles relative to the socket position of the robot were determined. With these values specified, the initial planning scene for the *MoveIt!* node running on the *ROS* system was set up. Now, changing the robot's socket position in *Unity* is equal to shifting all waypoints p_i on the trajectory and all obstacles in *MoveIt!* relative to the robot in the opposite direction of the position change. This enables the usage of one *MoveIt!* planning environment, while still generating valid trajectories for all socket positions of the robot.

According to the idea of this application, the *evaluation parameters* of this simula-

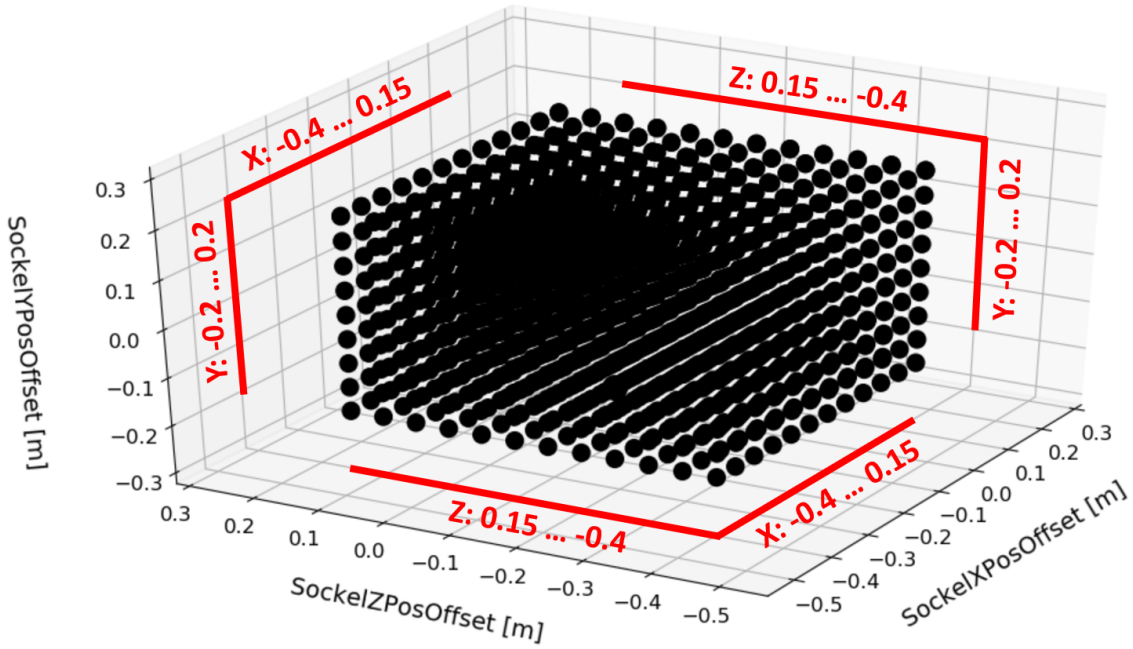

Fig. 7–7: The *Panda* robot's x-, y- and z-positions inside the industrial plant as *evaluation parameters* for the bin picking process.

tion comprise the x-, y- and z-position offsets ($x_{pos},\ y_{pos},\ z_{pos}$) of the robot relative to the zero position the bin picking should be evaluated for. Figure 7–7 visualizes the *evaluation parameters* and all combinations of them chosen in this application, forming a cuboid in *evaluation space*. The dimensions shown in Figure 7–7 correspond to the directions of axes as specified in *Unity* and are here related to the industrial plant depicted in Figure 7–6: the positive x- direction points towards the conveyor of the industrial plant, the positive y-axis is directed upwards and the positive z-direction points towards the box containing the clothing-object. Equation 7–25 shows the minimum and maximum values chosen for each position offset in meters. The ranges of these values were chosen such that the robot is still in a valid position inside the plant without instantly colliding with any obstacle. Other than that, the ranges of position offsets were limited by socket positions that obviously would lead to the *Panda* robot not being able to reach the box or the conveyor at all.

$$x_{pos} \in [-0.4\ m,\ 0.15\ m], \quad y_{pos} \in [-0.2\ m,\ 0.2\ m], \quad z_{pos} \in [-0.4\ m,\ 0.15\ m] \tag{7–25}$$

For each of these *evaluation parameters*, the step size is set to $s = 0.05\ m$.
One additional *evaluation parameter* was added to the *evaluation space*. Represented by a boolean, this parameter specifies if the box containing the clothing object has a special geometry, which is often seen in the real industrial environment of the industry partner of *SAINT*. The two different boxes are depicted in Figure 7–8. What makes the box geometry special, is that it is not a simple cuboid with an open top, but has additional overhanging pieces intruding into the open top of the box. Thereby, the opening of the box becomes a little smaller, making it harder for the robot to actually grasp the object without colliding with the box. In the remainder of this thesis, the two types of boxes will

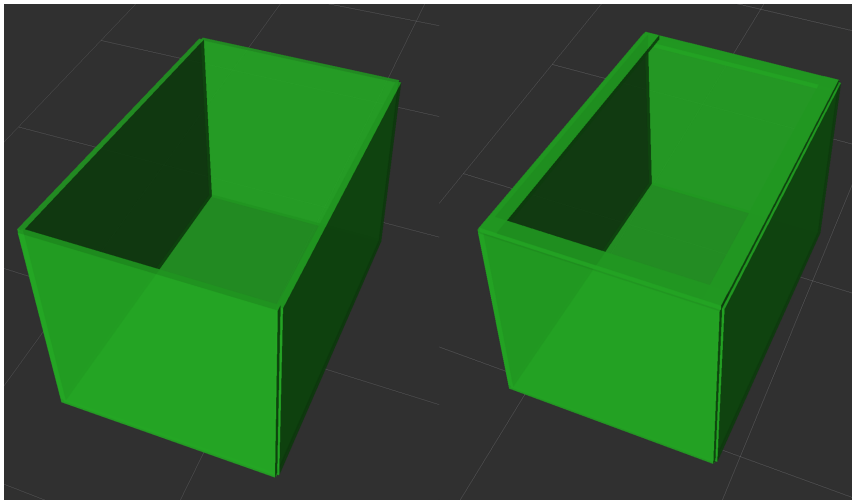

Fig. 7–8: Simple and complex box in the industrial plant of *SAINT*. Left: simple box, right: complex box.

be referred to as *simple* and *complex* boxes. Consequently, this evaluation comprises $12 * 9 * 12 * 2 = 2592$ combinations of *evaluation parameters*.

As explained in Section 7.3.1, the bin picking trajectories the *Panda* robot follows in *Unity* are generated by *MoveIt!*. To this end, the evaluation framework was extended, in order to be able to specify additional ROS *parameters*, that is, parameters used inside *Unity* but not being part of the *evaluation parameters* themselves. In this context, three components of the generated *MoveIt!* trajectory were specified as ROS *parameters*. Firstly, the timings of all configurations on the trajectory. Secondly, the corresponding joint positions on the trajectory. And lastly, the fraction of the trajectory *MoveIt!* was able to plan for the robot without colliding or leaving the robot's workspace. While the former two parameters are required to actually allow the robot to move along the trajectory in *Unity*, the latter *ROS* parameter is used as an *objective value* itself, since it provides interesting insights into how much of a trajectory could be executed based on the robot's socket position.

The following additional *objective values* were defined for this application. A boolean determining whether the robot was able to completely perform the act of bin picking, i.e. if the robot was positioned in such way that it could pick up the object, move above the conveyor, release the object and return to its initial position, was selected as another *objective value*. Furthermore, the energy consumption of the robot during its motion, calculated according to Section 7.2.2, was evaluated.
Section 8.2 presents the results corresponding to this evaluation setup.

8 Evaluation Results

In this chapter, the evaluation results of the two exemplary applications discussed in Chapter 7 are presented. All results are obtained from the *Point Cloud Visualization* component of the evaluation framework. Section 8.1 works through the result obtained from the robot trajectory evaluation described in Section 7.2. Subsequently, Section 8.2 presents the results obtained from evaluating the robot bin picking process in the context of the *SAINT* project introduced in Section 7.3.

8.1 Robot Trajectory Evaluation

The main objective of the evaluations carried out in the context of the application presented in Section 7.2 is to analyse the electric energy consumption of the *Panda* robot on different trajectories. As specified in Section 7.2.3, the two *evaluation parameters* represent the second and fourth joint angle $q_2[2]$ and $q_2[4]$, respectively, of the intermediate configuration on the robot trajectories. The total electric energy is obtained by summing up the individual energy consumption of each joint motor, which in turn is calculated according to the approach presented in Section 7.2.2. Generally, every solution in the *evaluation space*, that is, every pair of *evaluation parameters*, is represented by a circle in the plot. Table 8–1 summarizes all *evaluation parameters* and *objective values* of this evaluation.

Figure 8–1 depicts the total electric energy consumption of the *Panda* robot for all combinations of *evaluation parameters*, that is, for all trajectories simulated in *Unity*. In

Tab. 8–1: *Evaluation parameters* and *objective values* of the robot trajectory evaluation.

Name	Symbol	Unit	Min	Max
SecondJointAngle	$q_2[2]$	$^\circ$	-100	20
FourthJointAngle	$q_2[4]$	$^\circ$	0	140
TotalElectricEnergyConsumption	E_{el}	Ws	19.46	110.02
JointwiseElectricEnergyConsumption1	$E_{el}[1]$	Ws	5.08	40.27
JointwiseElectricEnergyConsumption2	$E_{el}[2]$	Ws	4.78	63.88
JointwiseElectricEnergyConsumption4	$E_{el}[4]$	Ws	1.31	35.64
TrajectoryExecutionTime	t	s	1.28	2.12
TotalJointDistance	s_{joint}	$^\circ$	177.5	590.2
TotalCartesianDistance	s_{Cart}	m	2.17	2.90
AverageJointVelocities1	$\dot{q}_{avg}[1]$	$^\circ/\mathrm{s}$	78.5	129.5
AverageJointVelocities2	$\dot{q}_{avg}[2]$	$^\circ/\mathrm{s}$	0.5	88.0
AverageJointVelocities4	$\dot{q}_{avg}[4]$	$^\circ/\mathrm{s}$	0.8	120.3

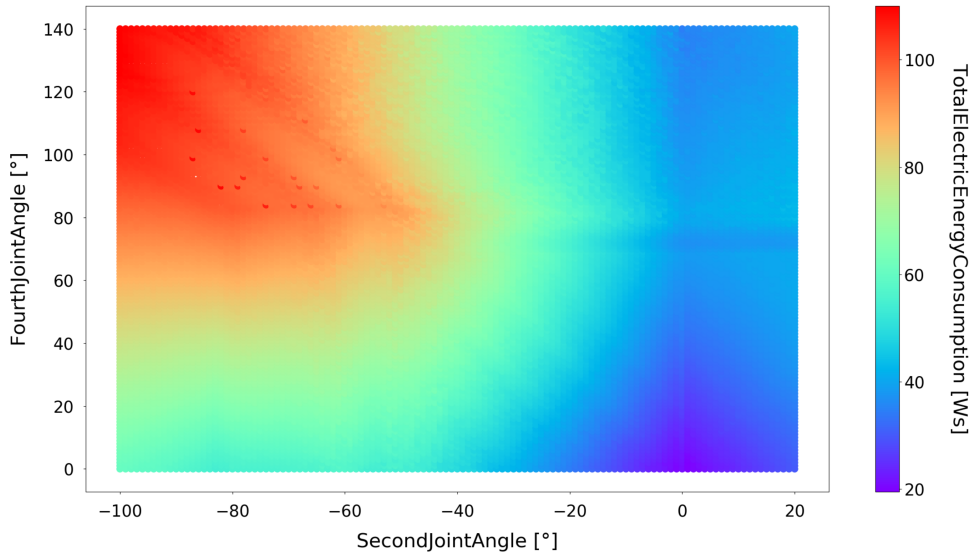

Fig. 8–1: Total electric energy consumption for all trajectories.

Figure 8–1, the x-axis represents $q_2[2]$, ranging from $-100\ °$ to $20\ °$, with a resolution, i.e. step size, of $1\ °$. With the same resolution, the y-axis ranges from $0\ °$ to $140\ °$, representing $q_2[4]$. Hence, the plot in Figure 8–1 comprises the full 2D grid in *evaluation space* spanned by the two *evaluation parameters*. The third dimension in the plot, that is, the color bar, shows the spectrum of electric energy consumption E_{el} in watt-seconds. The resulting energy consumption for all trajectories ranges from $E_{el,min} = 19.46\ \mathrm{Ws}$ in violet, corresponding to $[q_2[2] = 0\ °,\ q_2[4] = 0\ °]$, to $E_{el,max} = 110.02\ \mathrm{Ws}$ in red, corresponding to $[q_2[2] = -100\ °,\ q_2[4] = 140\ °]$.
The differences in electric energy consumption between neighbouring solutions are relatively homogeneous and smooth. Neighbouring solutions here refers to the 2D grid of *evaluation parameters*, and thus refers to trajectories being similar and only differing slightly in the intermediate configuration q_2. An accumulation of trajectories with very low electric energy consumption can be identified for values of $q_2[2]$ and $q_2[4]$ close to zero degree. A clear trend towards higher energy consumption can be seen when going towards trajectories, in which both the second and fourth joint move significantly.

Since on all trajectories, only the first, second and fourth joint of the *Panda* robot move actively, while the remaining joints hold their positions, the individual electric energy consumption of these three joints are depicted in Figures 8–2, 8–3 and 8–4. All these figures show the two *evaluation parameters* $q_2[2]$ and $q_2[4]$ on the x- and y-axis, respectively. The color bar represents the electric energy consumption of the first joint in Figure 8–2, the second joint in Figure 8–3 and the fourth joint in Figure 8–4.

Looking at *Panda*'s first joint motor and its electric energy consumption on all trajectories visualized in Figure 8–2, an accumulation of high energy consumption values can be identified for all trajectories with $-85\ ° \leq q_2[2] \leq -40\ °$ and $q_2[4] \approx 85\ °$, with the maximum $E_{el,max}[1] = 40.27\ \mathrm{Ws}$ for $q_2[2] = -74\ °$ and $q_2[4] = 85\ °$. Besides that, clearly the lowest energy consumption values result from trajectories with $q_2[4] \approx 0\ °$ and $q_2[2] = -100\ °$. Indeed, the minimum electric energy consumption of the first joint motor

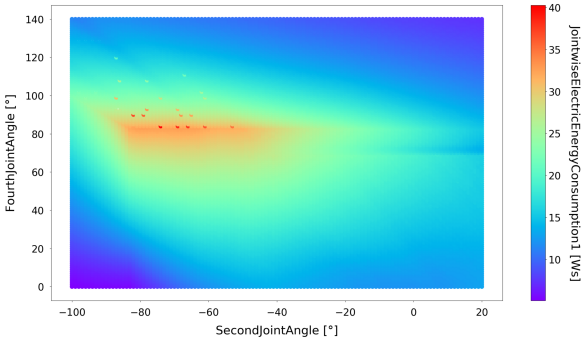

Fig. 8–2: Electric energy consumption of the first joint for all trajectories.

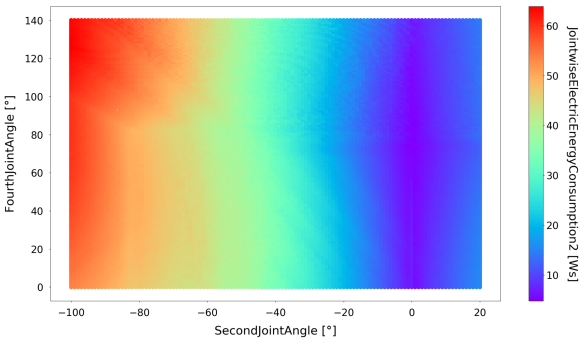

Fig. 8–3: Electric energy consumption of the second joint for all trajectories.

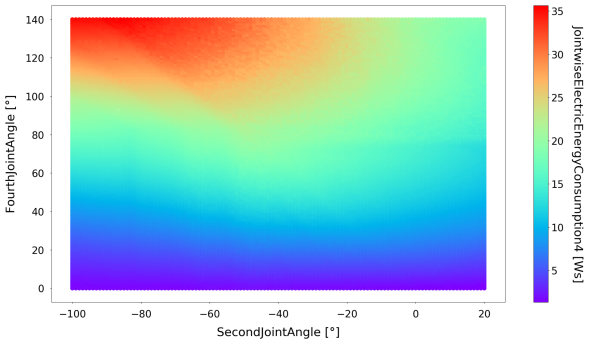

Fig. 8–4: Electric energy consumption of the fourth joint for all trajectories.

$E_{el,min}[1] = 5.08$ Ws is obtained on the trajectory $[q_2[2] = -100\ °,\ q_2[4] = 0\ °]$.
Going into more detail, Figure 8–3 depicts the energy consumption of the *Panda* robot's second joint motor. The lowest electric energy consumption of this motor $E_{el,min}[2] = 4.78$ Ws is obtained on the trajectory $[q_2[2] = 1\ °,\ q_2[4] = 76\ °]$. In general, the lowest electric energy consumption of the second joint motor is obtained on trajectories, where $q_2[2]$ remains close to zero, independent from the value of $q_2[4]$. The solution leading to the maximum of all energy consumptions of the second joint motor corresponds to the trajectory $[q_2[2] = -100\ °,\ q_2[4] = 126\ °]$ and results in $E_{el,max}[2] = 63.88$ Ws. Furthermore, a trend can be seen, showing a higher electric energy consumption for smaller values of $q_2[2]$, while the fourth joint angle $q_2[4]$ shows almost no influence on $E_{el}[2]$.
Lastly, Figure 8–4 shows the electric energy consumption of the fourth joint motor for all trajectories. The minimum for the fourth joint lies at $E_{el,min}[4] = 1.31$ Ws on the trajectory corresponding to $[q_2[2] = 20\ °$ and $q_2[4] = 0\ °$, while the maximum lies at $E_{el,max}[4] = 35.64$ Ws for $[q_2[2] = -86\ °,\ q_2[4] = 140\ °]$. This plot clearly shows, the smaller the fourth joint angle at q_2, the lower the fourth joint motor's electric energy consumption. On trajectories with lots of movement in both the second and fourth joints, the fourth joint motor consumes the most electric energy. Thus, as opposed to the electric energy consumption of the second joint motor being almost independent from $q_2[4]$, $E_{el}[4]$ is clearly influenced by the movement of the second joint.

In order to gain a deeper understanding of the behaviour of the electric energy con-

sumption of the robot on different trajectories, Figures 8–5 and 8–6 visualize the relation between the total electric energy consumption and the overall distance covered by the robot.

For all simulated trajectories, Figure 8–5 shows the correlation between the robot's total electric energy consumption E_{el} on the y-axis and the overall joint distance s_{joint}, specified in degree. This joint distance is simply obtained by summing up the absolute values of the changes in joint angles for all time steps and all joints. On all trajectories, the shortest joint distance covered equals $s_{joint,min} = 177.5\ °$ for $[q_2[2] = 0\ °,\ q_2[4] = 0\ °]$, while the longest joint distance lies at $s_{joint,max} = 590.2\ °$ on the trajectory with $[q_2[2] = -100\ °,\ q_2[4] = 140\ °]$. Figure 8–5 shows, that the trajectory with the highest total joint distance $s_{joint,max}$ simultaneously represents one of the highest electric energy consumptions $E_{el} = 109.95\ \mathrm{Ws}$. Furthermore, the trajectory with the lowest joint distance $s_{joint,min}$ corresponds to $E_{el,min} = 19.46\ \mathrm{Ws}$. In general, the figure shows that on trajectories with a total joint distance of approximately $s_{joint} = 400\ °$, the range of energy consumption values is the highest. As an example, the trajectory $[q_2[2] = 0\ °,\ q_2[4] = 140\ °]$ leads to $s_{joint} = 428.8\ °$ and an energy consumption of $E_{el} = 34.42\ \mathrm{Ws}$, while the trajectory $[q_2[2] = -91\ °,\ q_2[4] = 88\ °]$ results in almost the same joint distance $s_{joint} = 429.0\ °$ but consumes $E_{el} = 100.21\ \mathrm{Ws}$ of electric energy. Overall, a slight tendency towards high electric energy consumption for trajectories with high joint distance and vice-versa can be identified.
Similarly, Figure 8–6 visualizes the relation between total electric energy consumption and distance covered in Cartesian space, that is, the total distance in meters the robot's end effector moves on the trajectories. The figure shows, that on the trajectory $[q_2[2] = -100\ °,\ q_2[4] = 140\ °]$, the robot has to cover the shortest Cartesian distance, $s_{Cart,min} = 2.17\ m$, of all trajectories. This trajectory results in the highest electric energy consumption $E_{el,max} = 110.02\ \mathrm{Ws}$. Furthermore, the trajectory with the longest Cartesian distance of $s_{Cart,max} = 2.90\ m$ results in a relatively low electric energy consumption of $E_{el} = 39.86\ \mathrm{Ws}$ and corresponds to the trajectory $[q_2[2] = 20\ °,\ q_2[4] = 140\ °]$. Additionally, Figure 8–6 shows an accumulation of trajectories with high electric energy consumption for relatively small Cartesian distances. Trajectories on which the robot's end effector covers long Cartesian distances tend to result in relatively low electric energy consumption. However, in contrast to these observed tendencies, the minimum electric energy consumption $E_{el,min}$ results from a trajectory with a rather short Cartesian distance of $s_{Cart} = 2.27\ m$.

Finally, Figure 8–7 shows a 3D plot, visualizing the relation between total Cartesian distance s_{Cart}, total joint distance s_{joint}, trajectory execution time t in seconds and total electric energy consumption E_{el}. In general, no clear relation between the four visualized *objective values* can be identified. However, this figure highlights an accumulation of high electric energy consumption values on trajectories with short Cartesian distance, long execution time and long joint distance. Additionally, a trend towards lower electric energy consumption E_{el} for longer Cartesian distances s_{Cart}, similar to the observation in Figure 8–6, can be observed. The general influences of s_{joint} on the electric energy consumption as highlighted in Figure 8–5 can also be examined here. The trajectory execution time t seems to have the least influence on the robot's total electric energy

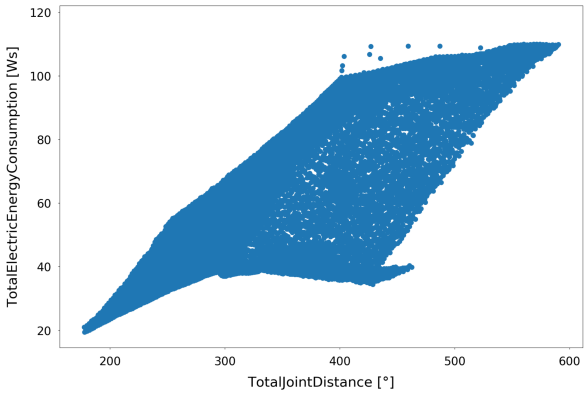

Fig. 8–5: Electric energy consumption over joint distance for all trajectories.

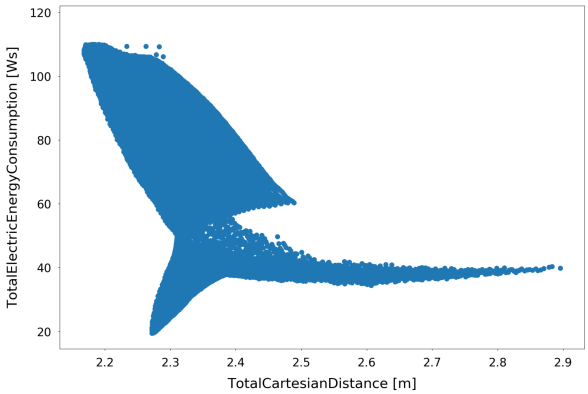

Fig. 8–6: Electric energy consumption over Cartesian distance for all trajectories.

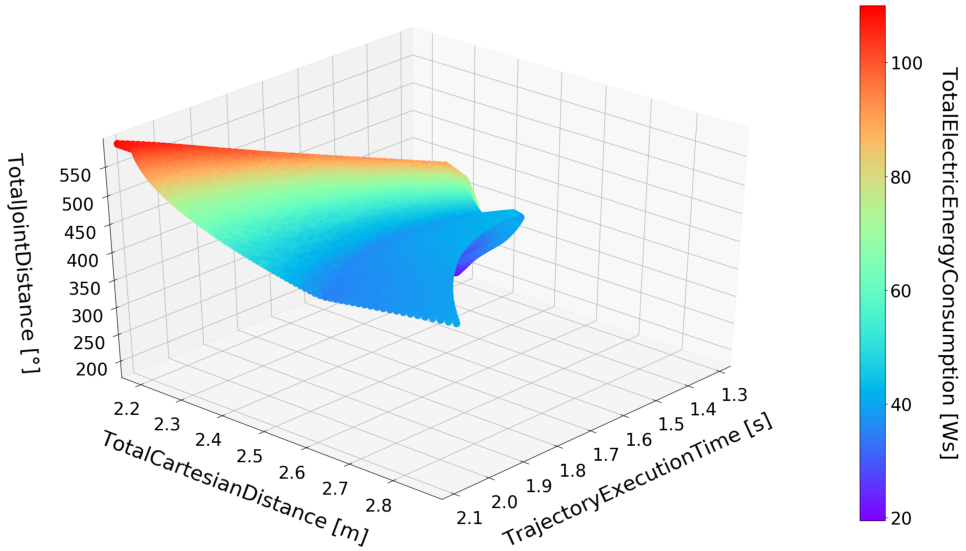

Fig. 8–7: 3D representation of total Cartesian and joint distance, execution time and electric energy consumption for all simulated trajectories.

consumption.

In Appendix B, additional results listed in Table 8–1 obtained from the evaluation performed here are visualized. However, they are not part of the main evaluation and discussion of the energy-optimality of robot trajectories.

8.2 Robotic Bin Picking Evaluation

The results presented in this section originate from the exemplary application of robotic bin picking in the context of the *SAINT* project, as explained in Section 7.3. In this application, the evaluation framework is used to evaluate the positioning of the *Panda* robot inside an industrial plant. This is done by identifying robot positions that allow the robot to successfully perform the whole process of bin picking, and subsequently analyse these positions for execution time and distance covered by the robot. As a reference, the 3D grid spanned by the three main *evaluation parameters* of the *evaluation space*

is depicted in Figure 7–7. In the following, successfully performing bin picking refers to the robot being able to follow all seven waypoints $p_i,\ i \in \{1...7\}$ (see Section 7.3.1) without collision. The corresponding robot position is denoted a valid position. Table 8–2 summarizes all *evaluation parameters* and *objective values* of this evaluation.

For all robot positions simulated in this evaluation, Figure 8–8 visualizes their ability to perform the complete process of bin picking. The x-, y- and z-position offsets are denoted *SockelXPosOffset, SockelYPosOffset, SockelZPosOffset*, respectively. Red circles correspond to robot positions that lead to a successful bin picking, while the violet circles represent all solutions for which the robotic bin picking process was terminated early. As mentioned in Section 7.3.3, besides the three position parameters, a fourth, boolean, *evaluation parameter* is specified. This parameter determines, if a special geometry for the box holding the clothing object is simulated, which models the true box geometry of the industrial plant more closely and leads to a more narrow aperture of the box (see Figure 7–8). To this end, the robotic bin picking process was simulated twice for each robot position. Correspondingly, Figure 8–8 shows the results with the simple box geometry, and Figure 8–9 equally illustrates the results of all evaluations in which the complex box geometry was simulated.

In total, 1296 robot positions were tested for both types of boxes, respectively. With the simple box geometry, the results in Figure 8–8 show that overall 139 valid positions, that is, $10.73\ \%$ of all simulated positions, exist. In contrast, only 46 positions, i.e. $3.55\ \%$, allowed the robot to completely move along all waypoints when falling back to the complex box geometry. All valid positions for bin picking with the complex box geometry are also valid positions when simulating the simple box geometry. Going into more

Tab. 8–2: *Evaluation parameters* and *objective values* of the robotic bin picking evaluation.

Name	**Symbol**	**Unit**	**Min**	**Max**
SockelXPosOffset	x_{pos}	m	-0.4	0.15
SockelYPosOffset	y_{pos}	m	-0.2	0.2
SockelZPosOffset	z_{pos}	m	-0.4	0.15
UseSimpleBoxGeometry				
FullTrajectoryExecuted	$-$	$-$	0	1
TrajectoryExecutionTime	t	s	11.94	19.39
TotalJointDistance	s_{joint}	°	1643.9	3590.3
TotalCartesianDistance	s_{Cart}	m	3.56	9.14
UseComplexBoxGeometry				
FullTrajectoryExecuted	$-$	$-$	0	1
TrajectoryExecutionTime	t'	s	12.90	17.46
TotalJointDistance	s'_{joint}	°	1836.3	3031.0
TotalCartesianDistance	s'_{Cart}	m	3.64	4.84

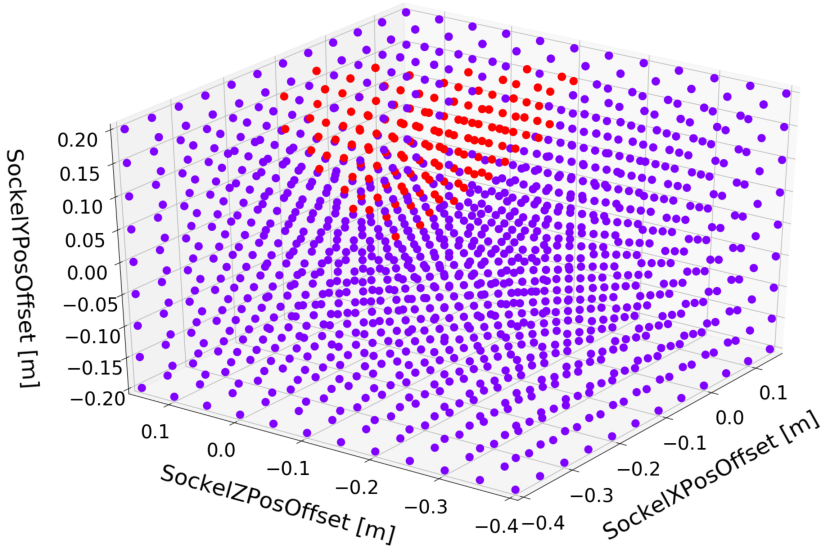

Fig. 8–8: 3D representation of all evaluated robot positions and their ability to perform bin picking with the simple box geometry. *Red*: bin picking successfully performed, *violet*: bin picking terminated early.

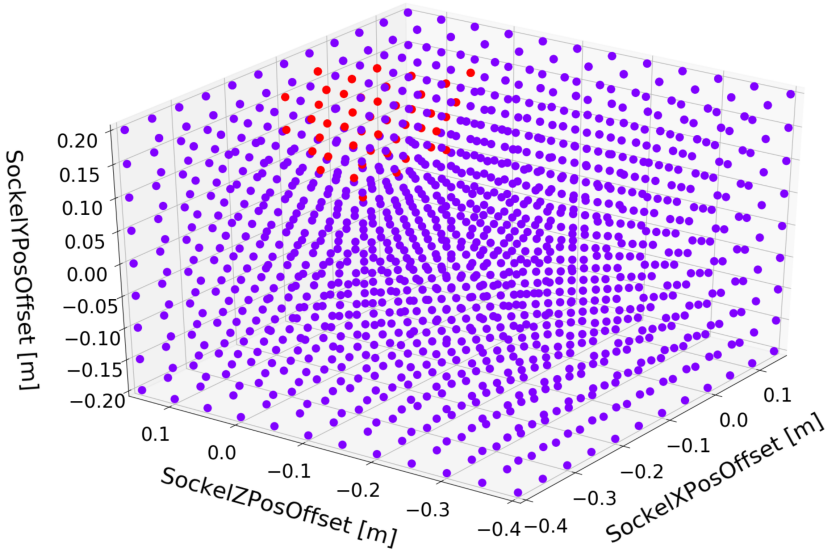

Fig. 8–9: 3D representation of all evaluated robot positions and their ability to perform bin picking with the complex box geometry. *Red*: bin picking successfully performed, *violet*: bin picking terminated early.

detail, using the simple box geometry, the positions in Figure 8–8 leading to a successful bin picking have the following characteristics. It is important to note, that the position values in *evaluation space* represent the offset of the robot position from its origin, which is set as the position of the robot depicted in Figure 7–6.

- X**-positions:** Of the total range of x-position offset values, only those in the interval $[-0.2\ m,\ 0.15\ m]$ lead to at least one complete bin picking process.

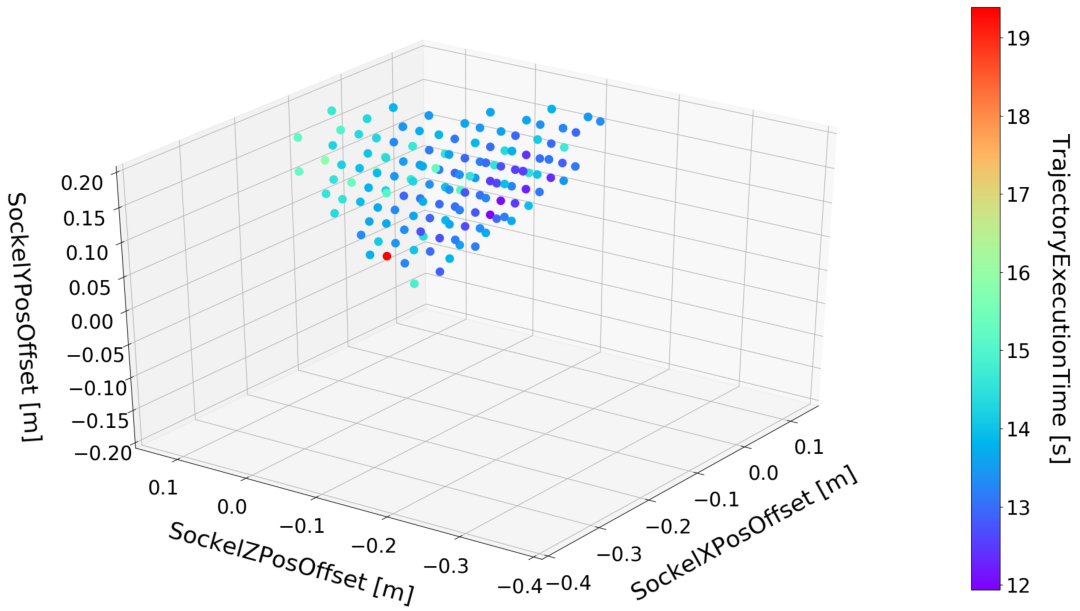

Fig. 8–10: Execution time of each bin picking process for all valid positions with the simple box geometry.

- Y**-positions:** For the y-position offset, only one solution with a negative offset is valid, more precisely, only one solution for $y_{pos} = -0.05\ m$ lead to a successful bin picking. Furthermore, all simulated positive y-position offset values lead to at least one valid solution.
- Z**-positions:** Similar to the x-position offsets, at least one valid solution exists for all values in the interval $[-0.2\ m,\ 0.15\ m]$ simulated for the z-position offset.

Correspondingly, the following lists the characteristics of valid positions depicted in Figure 8–9 corresponding to valid solutions:

- X**-positions:** Of the total range of x-position offset values, only those in the interval $[-0.2\ m,\ 0\ m]$ lead to at least one complete bin picking process.
- Y**-positions:** Only y-position offsets in the interval $[0.05\ m,\ 0.2\ m]$, i.e. only positive position offsets, lead to at least one successful bin picking.
- Z**-positions:** For all values in the interval $[-0.1\ m,\ 0.15\ m]$ simulated for the z-position offset at least one valid solution exists.

Besides these results, the *objective values* of the *evaluation space* defined for this application also comprise the total execution time in seconds required for the bin picking process, as well as the corresponding Cartesian and joint distance covered by the robot during bin picking, in meters and degress, respectively. The results for the execution time are depicted in Figure 8–10 for all valid positions with the simple box geometry. All non-valid solutions are removed from the plots and not considered in the scale of the color bar. The results corresponding to the simulations carried out with the complex box geometry can be found in Appendix C.

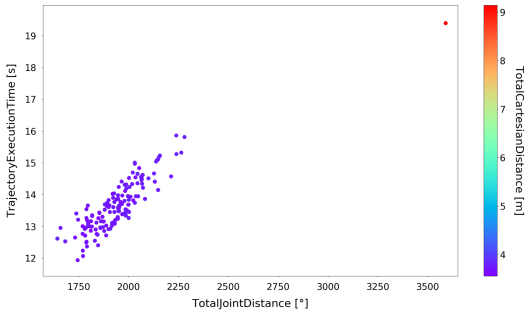

Fig. 8–11: Relation between total Cartesian distance, total joint distance and execution time with the simple box geometry.

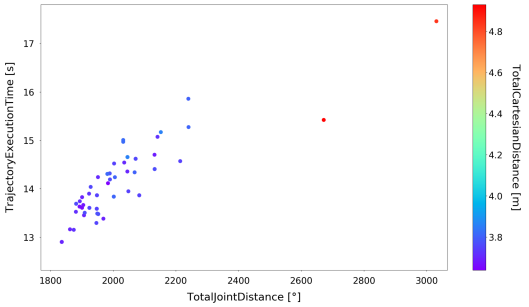

Fig. 8–12: Relation between total Cartesian distance, total joint distance and execution time with the complex box geometry.

The results of the execution time obtained from all valid positions for the simple box geometry, depicted in Figure 8–10, show a wide range of values. The minimum execution time was $t_{min} = 11.94\ s$, corresponding to the robot position with $x_{pos} = 0\ m,\ y_{pos} = 0.05\ m,\ z_{pos} = -0.05\ m$. The longest execution time was obtained from the robot position corresponding to $x_{pos} = 0\ m,\ y_{pos} = -0.05\ m,\ z_{pos} = 0.1\ m$, with a total time of $t_{max} = 19.39\ s$. For reference, the minimum execution time for bin picking with the complex box geometry was $t'_{min} = 12.90\ s$ for the robot position $x_{pos} = -0.1\ m,\ y_{pos} = 0.1\ m,\ z_{pos} = 0\ m$, while position offset $x_{pos} = 0\ m,\ y_{pos} = 0.1\ m,\ z_{pos} = 0\ m$ resulted in the longest time, $t'_{max} = 17.46\ s$. Furthermore, Figure 8–10 shows a trend towards higher execution time for valid positions with positive offset in the z-direction. A similar trend is visible for the solutions with the complex box geometry. The longest execution time seems to be an outlier when compared to all other valid solutions in Figure 8–10.

Finally, Figures 8–11 and 8–12 depict the relation between three different *objective values*: (1) execution time in seconds on the vertical axis; (2) total joint distance in degree on the horizontal axis; (3) total Cartesian distance in meters on the color bar. Here, Figure 8–11 shows the results obtained for all valid positions with the simple box geometry, Figure 8–12 displays the same results with the complex box geometry.

Figure 8–11 clearly highlights, that the aforementioned outlier concerning the execution time for position $x_{pos} = 0\ m,\ y_{pos} = -0.05\ m,\ z_{pos} = 0.1\ m$ also reflects in total joint and Cartesian distance covered by the robot during bin picking, simultaneously representing the position with the highest joint distance, Cartesian distance and execution time. Looking at the results obtained from simulating the complex box geometry, two similar outliers can be identified. Furthermore, all remaining solutions have very similar values for the total Cartesian distance ($s_{Cart} \approx 4\ m$). Besides that, an almost linear relation between trajectory execution time and total joint distance can be identified in Figure 8–11. The minimum joint distance $s_{joint,min} = 1643.9\ °$ is obtained when positioning the robot with an offset of $x_{pos} = -0.2\ m,\ y_{pos} = 0.2\ m,\ z_{pos} = -0.2\ m$. Furthermore, the position offset $x_{pos} = -0.1\ m,\ y_{pos} = 0.1\ m,\ z_{pos} = 0\ m$ results in $s'_{joint,min} = 1836.3\ °$, while at the same time corresponding to the position offset leading to $t'_{min} = 12.90\ s$.

Appendix C shows additional results obtained from the evaluation of robotic bin picking in the *SAINT* project. However, these results will not be discussed any further.

9 Discussion

This chapter discusses and critically assesses the results of the two exemplary applications presented in Chapter 8. The purpose of this chapter is to examine the results in detail, identify the application- and objective-specific optimal solutions, observe relations between *objective values* and *evaluation parameters*, and understand and explain the interdependencies of the various presented evaluation results. Section 9.1 discusses the results obtained from the robot trajectory evaluation presented in Section 8.1, Section 9.2 assesses the results of the robotic bin picking evaluation showcased in Section 8.2.

9.1 Energy-Optimal Robot Trajectories

As explained in Section 7.2.2, the joint-side torques of a robot follow the equations of motion, and thus entail complex interrelations between the robot links, such as mass and inertia coupling, as well as the influence of gravitational forces. Thus, it is difficult to explain the reasons behind the obtained energy consumption values, since many factors have to be considered, that might not be intuitively graspable. This section aims at discussing the results of the robot trajectory evaluation and finding plausible explanations for the observed behaviour.

Looking at Figure 8–1, the energy-optimal intermediate configuration of the *Panda* robot on the trajectories simulated in this evaluation corresponds to $[q_2[2] = 0\,^\circ,\ q_2[4] = 0\,^\circ]$. Below, five configurations on such a trajectory are listed, adding the configurations q_{1-2} between q_1 and q_2 and q_{2-3} between q_2 and q_3.

$$
\begin{aligned}
& q_1 = (0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_{1-2} = (-45\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_2 = (-90\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_{2-3} = (-135\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_3 = (-180\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ).
\end{aligned}
$$

Figure 9–1 shows the *Panda* robot in *Unity* for these five configurations. Furthermore, the trajectory leading to the highest electric energy consumption corresponds to $[q_2[2] = -100\,^\circ,\ q_2[4] = 140\,^\circ]$, again with five configurations on this trajectory specified below.

$$
\begin{aligned}
& q_1 = (0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_{1-2} = (-45\,^\circ \quad -50\,^\circ \quad 0\,^\circ \quad 70\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_2 = (-90\,^\circ \quad -100\,^\circ \quad 0\,^\circ \quad 140\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_{2-3} = (-135\,^\circ \quad -50\,^\circ \quad 0\,^\circ \quad 70\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ) \\
\rightarrow \quad & q_3 = (-180\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ \quad 0\,^\circ).
\end{aligned}
$$

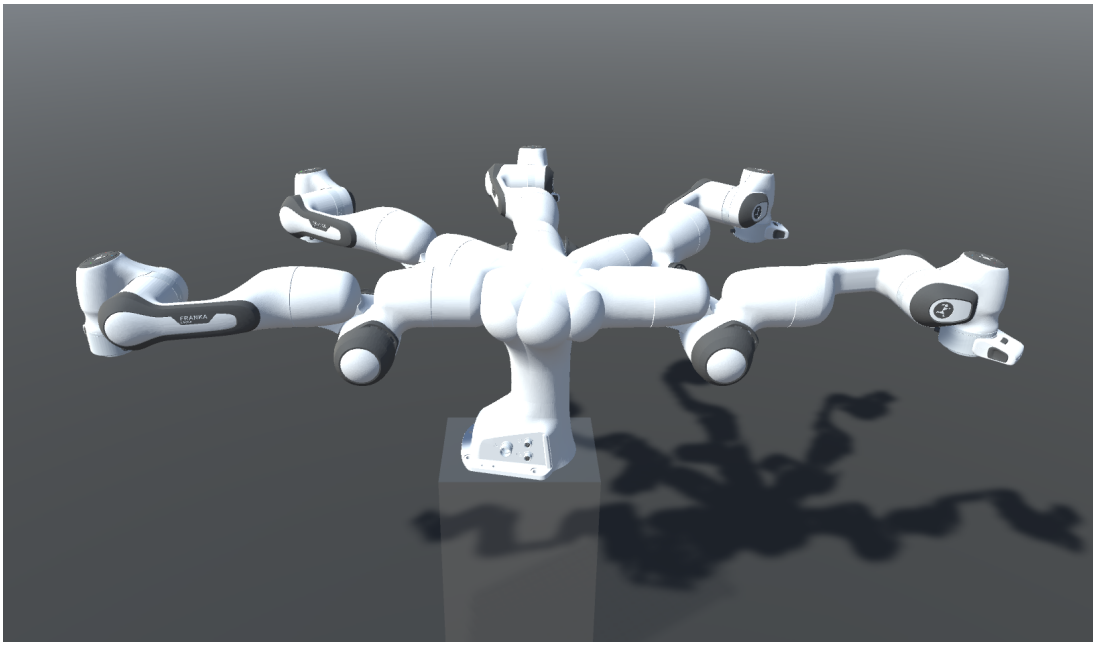

Fig. 9–1: Five configurations on the energy-optimal trajectory. The motion is progressing in clock-wise direction.

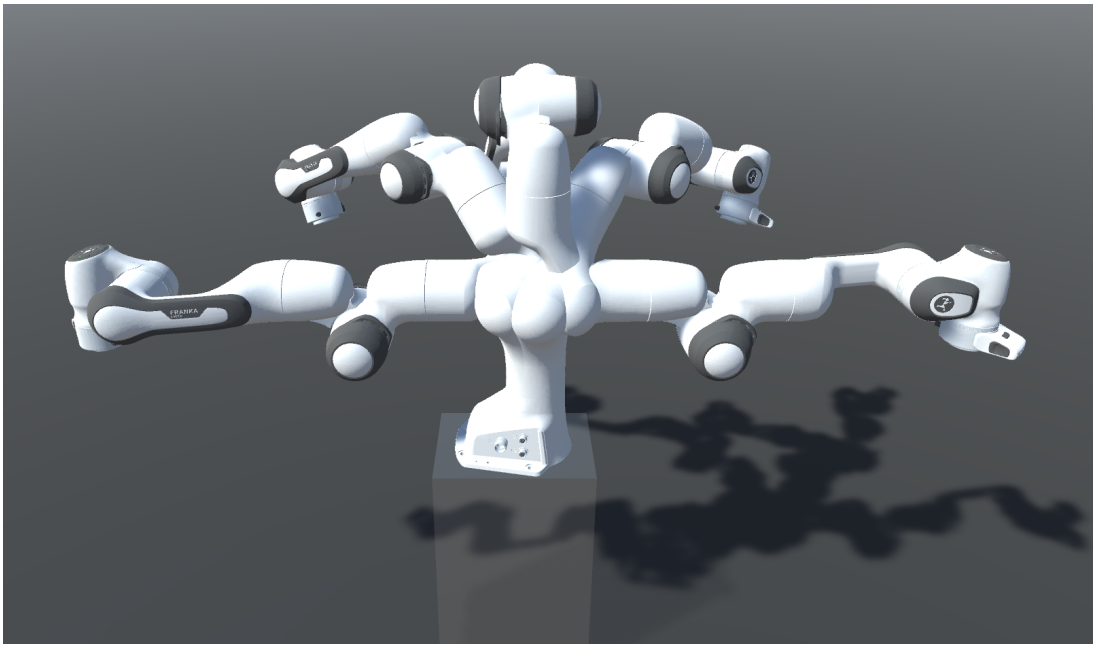

Fig. 9–2: Five configurations on the trajectory with the highest electric energy consumption. The motion is progressing in clock-wise direction.

In Figure 9–2 these five configurations are depicted.

Comparing the scenarios of the robot moving on the different trajectories to a human, picking up a cup from a table on his/her left, and releasing it on another table to the right, the observed behaviour of the robot's electric energy consumption is unexpected. Intuitively, a human would likely pick up the cup, bend the elbow and lower the arm from

the shoulder, thus forming a v-shape with the lower and upper arm, in order to bring the cup close to the body. Going back to the context of robot motion, this idea would correspond to the solutions where the intermediate configurations on the trajectory bring the end effector close to the axis of rotation. In the *evaluation space* of this application, such a trajectory corresponds to a lift of the robot coming from the second joint, i.e. $q_2[2] \ll 0\,^\circ$, and a simultaneous downwards bend of the robot's elbow coming from the fourth joint, i.e. $q_2[4] \gg 0\,^\circ$, thus forming a reverse v-shape. On the contrary, as shown in Figure 8–1, these trajectories seem to result in the highest electric energy consumptions in the whole *evaluation space*. However, there are multiple reasons why the human analogy does not necessarily apply to this application. Firstly, the underlying criteria that lead to the human moving the arm in a specific way are not fully understood [88]. Besides energy efficiency, there might be other, conflicting objectives for the selected motion, leading to a trade off that does not comply with the energy-optimal solution. Secondly, many of the evaluated trajectories do not comply with the physiological range of motion of a human elbow and shoulder, such as bending the lower arm downwards while keeping the upper arm straight, or forming a reverse v-shape with the lower and upper arm.

The results show a strong bias towards higher electric energy consumption when increasing the total joint motion on the trajectory. Clearly, the highest energy consumption values accumulate at trajectories with $q_2[2] < -60\,^\circ$ and $q_2[4] > 80\,^\circ$. Reducing the amount of joint motion comes along with a smooth decrease in total electric energy consumption. These observations are directly reflected in, both Figures 8–5 and 8–6, which relate the electric energy consumption to the joint and Cartesian distance covered by the robot, respectively. A long distance in joint space corresponds to a short distance in Cartesian space, since controlling both joints in the way specified by the *evaluation parameters* moves the end effector close to the robot socket, and hence reduces the overall Cartesian distance. While there is no linear relation between these distance values and the electric energy consumption, the evaluations highlight, a shorter total joint distance, that is, a longer Cartesian distance, results in lower electric energy consumption.

Generally, the torque exerted on the second joint through gravity is the highest when keeping the robot arm straight, i.e. $q_2[2] = 0\,^\circ$ and $q_2[4] = 0\,^\circ$, since the COM rotating around the second joint is relatively far away from its pivot point, while at the same time the axis of rotation and direction of gravity are fully aligned. Both, lifting the robot arm from the second joint or bending the elbow downwards, i.e. $q_2[4] > 0\,^\circ$, reduce the impact of this force onto the joint. However, the power required by each motor to realise the additional joint motion clearly outweighs the energy savings resulting from the reduced impact of gravitational forces onto the joints. Additionally, the coupling between joint torques and joint accelerations through the off-diagonal entries of the mass matrix $M(q)$ adds up. This means, due to the impact of inertia, acceleration in one joint applies additional stress onto another joint, thus increasing its required motor power to realise a specific motion.

Looking closer at the individual energy consumption values for joints one, two and four in Figures 8–2, 8–3 and 8–4, allows to break down the overall results into its roots.

The motor of *Panda*'s first joint consumes the least electric energy when the second joint fully lifts the robot upwards and the fourth joint remains at $0\ °$. This most likely arises from the fact that lifting the robot arm into a vertical position drastically reduces the inertia counteracting the rotational motion of the first joint. In contrast, an angle at the intermediate configuration of $q_2[4] \approx 85\ °$ results in the highest electric energy consumption for a large range of values of $q_2[2]$. Again, complex coupling between joints through their inertial properties and angular accelerations could be the reason behind these observations.
As shown in Figure 8–3, the second joint requires the highest electric energy input of all joints, on average. This lies in the fact that a large fraction of the robot's mass is connected to this joint, while its axis of rotation is, for a horizontally stretched robot arm, perpendicular to the effective direction of gravity. Consequently, lifting the robot arm by decreasing $q_2[2]$ consumes lots of electric energy. Even though the influence of gravity onto the second joint is the largest for $q_2[2] = 0\ °$, keeping the joint still requires by far the least electric energy.
Additionally, the electric energy consumption of the fourth joint motor depicted in Figure 8–4 shows similar behaviour, indicating a significant biases towards high energy consumption for trajectories with much joint motion.

Ultimately, the discussed results show that the total amount of distance covered by all joints of the robot constitutes the major fraction of electric energy consumption on a trajectory. These results do not comply with the initially expected evaluation outcome, which was deduced from the aforementioned human analogy.

9.2 Optimal Robot Positions for Bin Picking

Identifying valid positions for robotic bin picking in the industrial plant of the *SAINT* project by looking at the plots in Figures 8–8 and 8–9 matches the expected results. Due to the more narrow opening of the complex box compared to the simple one, the scope of valid positions for the *Panda* robot is reduced by about 100 possibilities. Moreover, none of the positions with $y_{pos} \leq 0\ m$, i.e. with a y-position of the robot lower than the origin, lead to a successful bin picking when the robot is confronted with the complex box geometry. This comes due to the fact that the narrow box opening leads to parts of the fourth and fifth link colliding with the box. With a wider opening, the robot has more space to move into the box without the risk of touching it. To this end, there exist also valid solutions for $y_{pos} \leq 0$ when using the simple box shape. For the exact same reason, the reduced scope of valid positions reflects in the possible position offsets in x- and z- direction when comparing the two types of boxes. Generally, the maximum position offsets $x_{pos} = 0.15\ m$ and $z_{pos} = 0.15\ m$ tested in this evaluation represent valid positions when picking objects out of the simple box. These offsets correspond to robot positions very close to both the conveyor and the box, which ensures that the box itself, as well as the position of the conveyor to place the object upon are within range of motion of the robot. As a result, moving the robot further away, at some point it becomes impossible for the robot to grab the object or reach the conveyor. This directly reflects in the fact that no valid solution exists for $x_{pos} < -0.2\ m$ and $z_{pos} < -0.2\ m$. Figure 8–9 furthermore demonstrates, that the robot being too close to the conveyor

prevents him from performing the complete bin picking process for the complex box shape, since no position with $x_{pos} > 0\ m$ is valid in this case. This, again, is founded on the reduced space available to the *Panda* robot, resulting from the narrow box opening, when grasping the object inside the box.

Looking into the valid positions in more detail, the time-optimal positioning of the robot when using the simple box corresponds to $x_{pos} = 0\ m,\ y_{pos} = 0.05\ m,\ z_{pos} = -0.05\ m$. However, multiple solutions exist with very similar execution times, only deviating by a few tenths of a second. Moreover, the time-optimal robot position when confronted with the complex box geometry corresponds to $x_{pos} = -0.1\ m,\ y_{pos} = 0.1\ m,\ z_{pos} = 0\ m$, while simultaneously resulting in the lowest distance covered in joint space. Generally, the results in Figure 8–10 show that an increase in the z-position offset simultaneously increases the time required for the complete bin picking process. Furthermore, a similar relation for the x- and y-position offsets cannot be identified. Using the feature of the evaluation framework's point cloud visualization component, that allows the user to re-run a simulation corresponding to a specific solution and looking at the simulation progress, made it possible to recognise the reason behind this relation between execution time and the robot's z_{pos} offset. The closer the *Panda* is positioned to the box, the more difficult it is for the robot to move into the box and lift the object, without self-colliding. If the robot stands very close to the box, a simple bend downwards from the second joint to reach the object is not possible, since this would lead to the robot colliding with the box's back wall. Hence, the robot has to perform a complex combination of joint movements, while avoiding collision between links. As a result, the joint trajectory in *MoveIt!* is planned at a higher resolution, that is, with more discrete joint configurations in critical positions, and lower joint velocities and accelerations to ensure a successful and safe execution. As a result, the overall bin picking execution time increases.

The outlier in Figure 8–10 with by far the longest execution time of $t = 19.39\ s$ simultaneously also represents the position that led to the highest overall joint and Cartesian distances covered by the robot, which are obtained for $x_{pos} = 0\ m,\ y_{pos} = -0.05\ m,\ z_{pos} = 0.1\ m$. By looking at this specific simulation, it could be observed that the *Panda* robot has to perform a complex combination of joint movements to actually reach inside the box and lift the object. Additionally, this results in a joint configuration at the third waypoint p_3—the position above the box with the clothing object in the gripper—which does not allow the robot to simply rotate clockwise around the first joint to reach the fourth waypoint p_4 above the conveyor, since it would exceed some of the robot's joint limits and likely result in a self-collision at the same time. Instead, the trajectory is planned such that the robot rotates counter-clockwise, thereby covering a long joint and Cartesian distance around joint one, while simultaneously ensuring that the gripper remains oriented vertically by additionally moving multiple robot joints in a complex manner. Thereby, the time necessary to execute the full trajectory, as well as the total joint and Cartesian distance covered during bin picking increase drastically.

Figure 9–3 depicts the robot position in the *Unity* simulation model leading to the shortest (left) and longest (right) execution time. Clearly, the robot position on the right is very close to the box and lower than the time-optimal position on the left, and

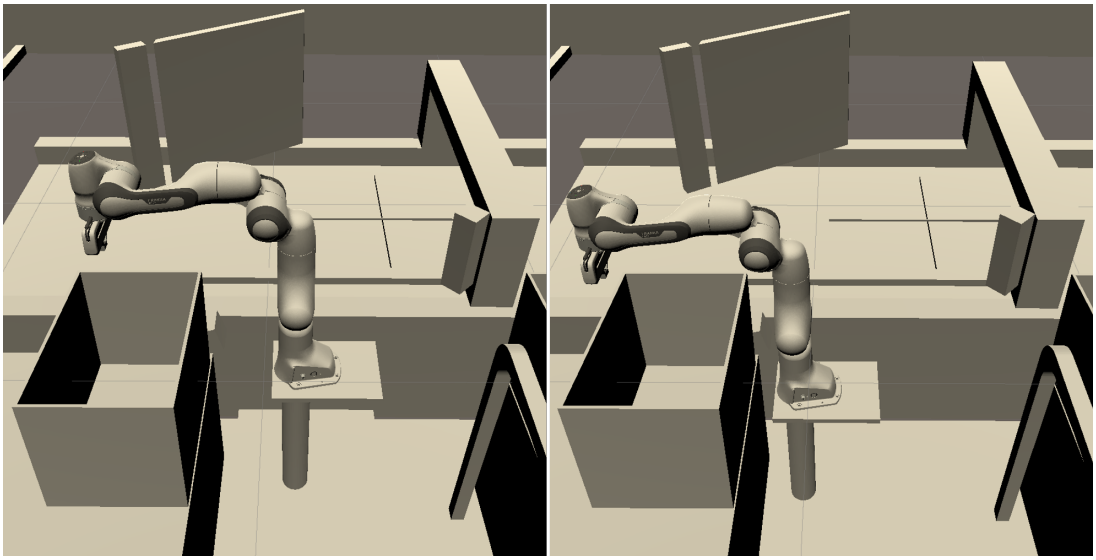

Fig. 9–3: Robot positions leading to shortest (left) and longest (right) execution time for bin picking.

thus confirms the aforementioned reasoning behind the long execution time. Further comparing these results to the execution time of all valid positions with the complex box geometry, neglecting the outlier discussed above, the average trajectory execution time increases by about one second. This, again, is based on the fact that with the reduced space available to the robot when grasping the object, especially the fifth robot link comes very close to the box, which is why the discretization of the trajectory in these cases includes more joint configurations with less distance in between and leads to reduced joint velocities during bin picking.

In Figures 9–4 and 9–5, three robot configurations on the bin picking trajectory are displayed, respectively. Figure 9–4 corresponds to the robot position resulting in the lowest execution time and joint distance when simulating the complex box geometry, $x_{pos} = -0.1\ m,\ y_{pos} = 0.1\ m,\ z_{pos} = 0\ m$. In contrast, Figure 9–5 displays the joint configurations on a trajectory obtained with the robot position $x_{pos} = 0.05\ m,\ y_{pos} = 0.05\ m,\ z_{pos} = 0.15\ m$, i.e. a position very close to the box, resulting in a longer execution time $t = 15.81\ s$ and joint distance $s_{joint} = 2281.0\ °$. Both figures show three configurations: (1) the joint configuration at waypoint p_2 inside the box, (2) the joint configuration at waypoint p_5 at the conveyor, (3) the joint configuration between waypoints p_6 and p_7. The figures support the reasoning behind the difference in execution time and joint distance for robot positions very close to the box.

Finally, Figures 8–11 and 8–12 illustrate the relation between the total joint and Cartesian distance and the trajectory execution time. First of all, the Cartesian distance covered by the robot is almost identical for all robot positions except the aforementioned outlier. Hence, this dimension does not carry much information, which is also reflected in the two figures showing that the colors of all solutions are almost identical. This similarity in Cartesian distances arises from the fact that the same seven waypoints $p_i,\ i \in \{1...7\}$ were used to generate the bin picking trajectory for all robot positions,

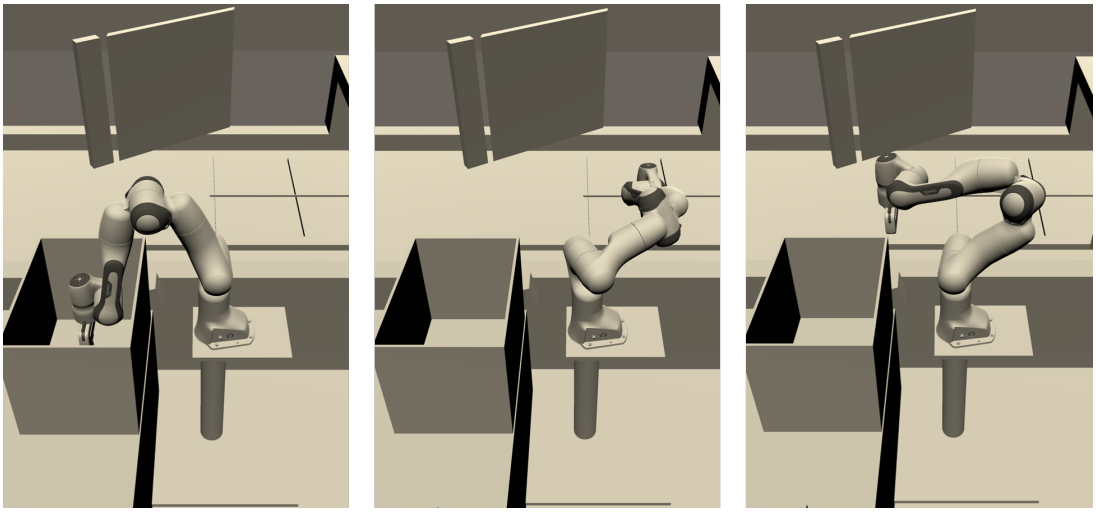

Fig. 9–4: Bin picking for the robot position with the lowest execution time and joint distance.

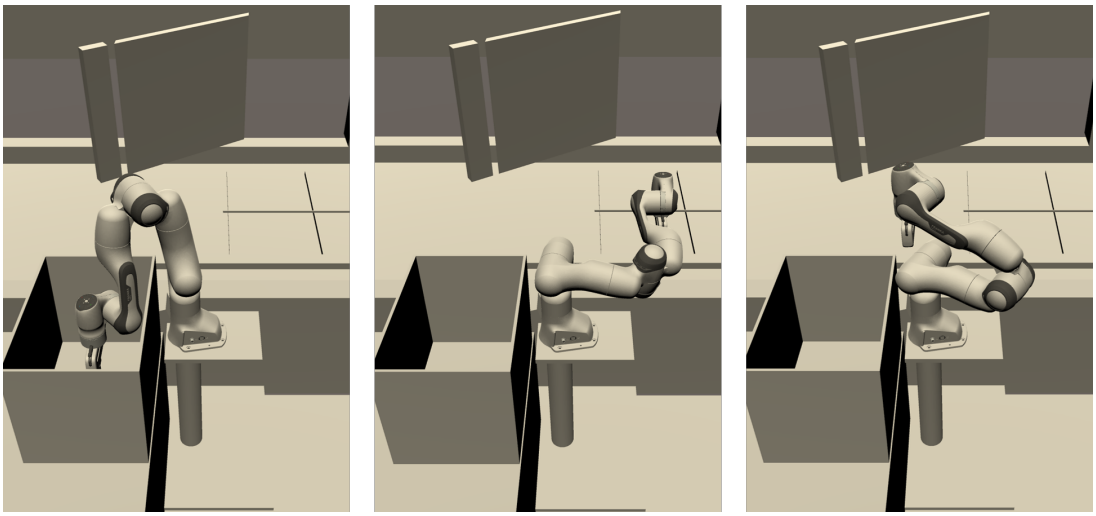

Fig. 9–5: Bin picking for the robot position with a high execution time and joint distance.

and thus fully matches the expectations. Besides that, these results show a clear tendency towards higher execution time for longer joint distances. This observation fits the explanation given above, that certain positions require the robot to perform very complex combinations of joint motions, in order to avoid self-collision or collision with the box, which increases the total execution time as explained above. Obviously, all motions simultaneously involving several joints increase the total joint distance.

10 Conclusion

The main objective of using the evaluation framework for simulating and evaluating different robot trajectories with the *Panda* robot was to identify the energy-optimal trajectory. In other words, the goal was to find the trajectory—parametrized by the second and fourth joint angle of the trajectory's intermediate configuration—leading to the least electric energy consumption of the *Panda* robot. The results show that for the trajectory with $[q_2[2] = 0\ °,\ q_2[4] = 0\ °]$ the electric energy consumption is minimized to $E_{el,min} = 19.46\ Ws$. In conclusion, the evaluation highlights, that moving the end effector close to the main axis of rotation by actively controlling both the second and fourth joint—similar to how a human would most likely move his/her arm to pick and place a cup—leads to the highest electric energy consumption of the *Panda* robot. This demonstrates, that the electric power required by the joint motors to actively move each joint significantly exceeds the benefit from reducing influence of gravity onto the joint motors by moving the robot's mass close to its origin. To this end, in the simulated pick and place application, the most energy efficient trajectory result from moving as few joints as possible, thus only rotating the first joint $180\ °$. However, it is important to note that these results have to be treated carefully, since many model assumptions were made for both controlling the *Panda* robot in *Unity* and modelling its joint motors.

In summary, the goal of evaluating different robot positions for bin picking in the context provided by the *SAINT* project was to identify the scope of valid robot positions available in the project. Again, a position is valid in the sense that the robot is able to perform the complete process of bin picking simulated in *Unity* (move across all seven Cartesian waypoints $p_i,\ i \in \{1...7\}$) without collision. Furthermore, two different box geometries containing the clothing object were considered. In total, $10.73\ \%$ of all simulated positions turned out to be valid when considering the simple box geometry, while only $3.55\ \%$ are valid when using the complex box geometry. This shows, that the scope of possible positions available in the project is rather small. Presumingly, the main reason for that is the relatively small *Panda* robot's size and range of motion, compared to its workspace in the industrial plant. Furthermore, the results show, there is a relation between bin picking trajectories with low trajectory execution time and short joint distance. Time-optimality, i.e. the shortest execution time, is obtained for the robot position corresponding to the offset $x_{pos} = 0\ m,\ y_{pos} = 0.05\ m,\ z_{pos} = -0.05\ m$ with the simple box geometry and $x_{pos} = -0.1\ m,\ y_{pos} = 0.1\ m,\ z_{pos} = 0\ m$ for the complex box geometry. In case of the simulation with the complex box, the same position offset results in the least distance covered by the robot in joint space. In conclusion, the robot position has to be chosen very carefully, in order to ensure a safe and collision-free bin picking, while considering influence of specific robot position offsets onto execution time and total joint distance.

The two evaluations carried out by means of the evaluation framework developed in this thesis demonstrate its wide range of applicability and its capabilities in extensively evaluating decision criteria and design alternatives of mechatronic systems in simulation. As shown in the two exemplary applications, based on an existing simulation model

in *Unity*, the evaluation framework can be used to efficiently evaluate the simulation objectives for all alternatives and visualize the resulting multi-criteria point clouds in a powerful GUI, that allows the user to gain an in-depth understanding of the impact of design modifications on the overall system performance from multiple perspectives. The utilization of the evaluation framework in the two presented applications and the presented results, furthermore demonstrate the framework's flexibility in examining the solution space and its independence from the actual context of the application and the underlying evaluation objectives. To this end, the evaluation framework serves as a great instrument for *virtual prototyping* of mechatronic systems.

Looking back at the objectives and requirements for this thesis, as presented in Chapter 3, ultimately the main goal is satisfied and all specified requirements are met. All components of the evaluation framework were implemented and all functionalities tested successfully. Furthermore, both initially specified scenarios used to demonstrate the evaluation framework's capabilities were implemented, evaluated with the framework's execution and evaluation component, and the application-specific optimal solutions identified by using the framework's *Point Cloud Visualization* component. Additionally, a strong focus and great amount of time was put into modelling the *Panda* robot in a physically realistic simulation environment by estimating the robot's inertial and energetic properties, as well as controlling the robot motion in *Unity* without the access to advanced and sophisticated control techniques for the *Panda* robot. Due to time constraints, the optional requirement of integrating machine learning algorithms into the framework was not dealt with. However, the general idea of integrating optimization algorithms into the evaluation framework is briefly discussed in the final Chapter 11.

Conclusively, the evaluation framework developed in this thesis proves to be a helpful tool for basic, general evaluation of simulation models, that is, for performing virtual prototyping on mechatronic systems. With the brute-force evaluation of all alternatives in the evaluation space, and the general, interactive visualization of the results, the evaluation framework poses a great instrument for initial, extensive evaluation of virtual prototypes, and can serve as the basis for more detailed model-based design and *virtual prototyping*.

11 Outlook

Due to the framework's generic nature, many possibilities for future applications, further improvements and framework extensions are provided, which will be briefly discussed in this chapter. Firstly, Section 11.1 presents two applications, in which the evaluation framework is intended to be used in the future. Subsequently, Section 11.2 discusses possible extensions of the evaluation framework, ranging from the utilization of various optimization techniques to the integration and combination of additional simulation software solutions.

11.1 Future Applications

Generally, the evaluation framework can be used to evaluate any simulation model of any system kind, independent from the field of application and evaluation objectives. Even the software used as simulation environment is not restricted to *Unity* only. In the following, two specific applications will be presented, in which the evaluation framework is intended to be used in the near future, alongside the objectives of deploying the framework in these applications.

11.1.1 Evaluation of Geometric and Dynamic Robot Parameters

Since no information on the *Panda* robot beyond the technical details specified in the publicly available datasheets is available, some assumptions on the simulation model of the Franka Emika *Panda* robot were taken over from specifications of a robot arm created at *Siemens AG* in Munich. Consequently, *Siemens* intends to deploy the evaluation framework in the near future for evaluating their own robot arm. With full technical specification of all robot components, including the exact dynamic parameters of the robot, detailed motor models with temperature and speed dependent efficiencies and information regarding workload-specific motor voltage and current, there is no need for all the assumptions made for the *Panda* robot as presented in Section 7.2. In contrast to the evaluation carried out in this thesis, using the extensive information available about the *Siemens* robot drastically increases the degree of detail, accuracy and validity obtained from evaluating the robot arm with the developed evaluation framework.
Similar to the objective presented in this thesis, one result of interest for *Siemens* is the total electric energy consumption of the robot when deployed in different environments. Beyond that, different design choices of the robot arm and their influence on the overall performance, as well as the impact of different dimensioning of joint motors and, e.g., the results of adding an additional joint at the end effector shall be investigated.

11.1.2 Testing and Evaluation of the Complete SAINT Project Implementation

The bin picking scenario implemented and evaluated in *Unity* in this thesis is part of the *SAINT* project at *TUM*. In order to keep the complexity of the implementation of this scenario within the scope of this thesis, the full process of bin picking with the robot was reduced to the problem of following a trajectory parametrized by multiple

Cartesian positions. While the results of the evaluation carried out in this context provide meaningful insights into the scope of valid robot positions available in the industrial plant, applying the evaluation framework to parts, or even the whole implemented *SAINT* system can benefit the current state of development. All components of the *SAINT* project implemented at *TUM* are part of a complex *ROS* program consisting of various nodes realising different functionalities. To this end, as already mentioned, the integration of *MoveIt!* into the evaluation framework's execution and evaluation component was implemented in a generic way, such that the simple *ROS* system only running *MoveIt!* can easily be replaced by the *SAINT* program in the future. Connecting the complete *ROS* program to the evaluation framework allows to evaluate the whole workflow of the bin picking process. This would allow the project partners of *SAINT* to efficiently evaluate, train and benchmark the implemented machine vision module responsible for recognizing the clothing objects, the algorithm identifying valid gripper positions and planning individual trajectories, as well as the fault recovery module for countless different scenarios in simulation.

11.2 Framework Extensions

In this section, several future extension possibilities for the implemented framework will be briefly discussed.

11.2.1 Mutli-Objective Design Optimization

The evaluation framework evaluates simulation models of mechatronic systems in a brute-force, extensive manner. Each *evaluation parameter* has to be defined for a specific range and step size, and the framework, subsequently, performs evaluations for all possible combinations of *evaluation parameters* and visualizes the complete planning scope, i.e. the scope of design alternatives, to the user. While this process might be very meaningful for initial, broad model evaluation, evaluating mechatronic system components in simulation at a later state of the development process might require more advanced search techniques, combined with optimization methods to identify objective-specific optimal solutions. As presented in Chapter 2, the resulting scope of design alternatives presented to the user after extensive evaluation represents a MCDM problem. In this context, finding the optimal solution among a collection of objective function,s that not only influence the overall system performance individually, but also impact each other, is referred to as multi-objective optimization, or multi-criteria design optimization. There exist countless approaches and architectures in the literature solving these kinds of multi-criteria optimization problems. In [89, 90] a summary and review of existing multi-criteria and multidisciplinary design optimization methods are presented. Genetic algorithms pose another advanced method for performing mutli-objective design optimization, see e.g. [91, 92, 93]. Furthermore, the framework could also be extended to perform optimization by means of evolutionary algorithms as described in, e.g. [94, 95, 96].
There exist many more approach to solve said type of problems. Extending the framework by one, or even multiple of these algorithms, would allow users to combine a first, broad and extensive evaluation to obtain an idea about the scope of design alternatives worth considering, and use optimization to identify specific optimal solutions.

11.2.2 External Software Interfaces

Another future extension possibility is to integrate other simulation software solutions into the evaluation framework. As shown in Section 7.3.2, the execution and evaluation component of this framework was easily extended by an interface to any *ROS* system running on a Linux machine. Similarly, additional interfaces can be added to the evaluation framework and integrated especially into the workflow of the execution and evaluation component. As it is often the case in *virtual prototyping* and *virtual commissioning*, experts from different domains need to test sub-component, domain specific features and design choices of the complete system, in order to see their impact on the overall system behaviour. Ideally, the process of testing these design alternatives does not proceed individually and detached from other domain-specific design modifications. By extending the execution and evaluation component to communicate and combine other simulation software, like e.g. *MATLAB/Simulink* and *ROS*, it would be possible to, e.g., simulate control-software of a robot arm in *MATLAB/Simulink*, generate robot trajectories in *MoveIt!* and combine both results in a physically realistic simulation model in *Unity*. Consequently, the extended framework would make it possible to evaluate simulation models of mechatronic systems by simulating domain-specific system components in their respective optimized simulation environment and combine the results into one *Unity* simulation, thereby closely and accurately mirroring the physical product in a realistic simulation environment.
Furthermore, a possible extension for the evaluation framework is the implementation of the parallel execution of simulations on GPUs, in order to increase the efficiency of the framework even more.

Bibliography

[1] R. Wagner, B. Schleich, B. Haefner, A. Kuhnle, S. Wartzack, and G. Lanza, “Challenges and potentials of digital twins and industry 4.0 in product design and production for high performance products,” *Procedia CIRP*, vol. 84, pp. 88–93, 2019.

[2] G. Ferretti, G. Magnani, and P. Rocco, “Virtual prototyping of mechatronic systems,” *Annual Reviews in Control*, vol. 28, no. 2, pp. 193–206, 2004.

[3] F. Zorriassatine, C. Wykes, R. Parkin, and N. Gindy, “A survey of virtual prototyping techniques for mechanical product development,” *Proceedings of the institution of mechanical engineers, Part B: Journal of engineering manufacture*, vol. 217, no. 4, pp. 513–530, 2003.

[4] “Integrated mechanical electronic systems,” in *Mechatronic Systems: Fundamentals*. London: Springer London, 2005, pp. 1–32. [Online]. Available: https://doi.org/10.1007/1-84628-259-4_1

[5] C. J. Paredis, A. Diaz-Calderon, R. Sinha, and P. K. Khosla, “Composable models for simulation-based design,” *Engineering with Computers*, vol. 17, no. 2, pp. 112–128, 2001.

[6] B. Schleich, N. Anwer, L. Mathieu, and S. Wartzack, “Shaping the digital twin for design and production engineering,” *CIRP Annals*, vol. 66, no. 1, pp. 141–144, 2017.

[7] S. Boschert and R. Rosen, “Digital twin—the simulation aspect,” in *Mechatronic futures*. Springer, 2016, pp. 59–74.

[8] P. Hoffmann, R. Schumann, T. M. Maksoud, and G. C. Premier, “Virtual commissioning of manufacturing systems a review and new approaches for simplification.” in *ECMS*. Kuala Lumpur, Malaysia, 2010, pp. 175–181.

[9] C. G. Lee and S. C. Park, “Survey on the virtual commissioning of manufacturing systems,” *Journal of Computational Design and Engineering*, vol. 1, no. 3, pp. 213–222, 2014.

[10] J. Blackborow, “Digital experiences in the physical world - are AEC and manufacturing companies ready for real-time 3D?” *A Forrester Consulting Thought Leadership Paper Commissioned By Unity*, Mar. 2020, Retrieved on: 2020-06-10. [Online]. Available: https://create.unity3d.com/unity-forrester-study

[11] Verein Deutscher Ingenieure (VDI)-Fachbereich Produktentwicklung und Mechatronik, “VDI 2206 - design methodology for mechatronic systems,” june 2004.

[12] I. Gräßler, J. Hentze, and X. Yang, “Eleven potentials for mechatronic v-model,” in *6th International Conference Production Engineering and Managment*, vol. 29, no. 30.09, 2016.

[13] J. Banks, J. S. CARSON II, L. Barry *et al.*, “Discrete-event system simulation,” 2005.

[14] “ns-3,” https://www.nsnam.org/, Accessed on: 2020-06-30.

[15] “OMNet++,” https://omnetpp.org/, Accessed on: 2020-06-30.

[16] “MATLAB/Simulink,” https://de.mathworks.com/products/simulink.html, Accessed on: 2020-06-30.

[17] “Unreal Engine,” https://www.unrealengine.com/en-US/, Accessed on: 2020-06-30.

[18] “Unity,” https://unity.com/, Accessed on: 2020-06-30.

[19] “Catia - Dassault Systèmes,” https://www.catia.com/, Accessed on: 2020-06-30.

[20] “Delmia - Dassault Systèmes,” https://www.delmia.com/, Accessed on: 2020-06-30.

[21] “Adams,” https://www.mscsoftware.com/product/adams/, Accessed on: 2020-06-30.

[22] “Altair HyperWorks,” https://altairhyperworks.com/, Accessed on: 2020-06-30.

[23] “Simscape Multibody,” https://de.mathworks.com/products/simmechanics.html, Accessed on: 2020-06-30.

[24] “Modelica language,” https://www.modelica.org/modelicalanguage, Accessed on: 2020-06-30.

[25] H.-J. Zimmermann and L. Gutsche, “Multi-criteria-entscheidungen,” in *Multi-Criteria Analyse: Einführung in die Theorie der Entscheidungen bei Mehrfachzielsetzungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 21–33. [Online]. Available: https://doi.org/10.1007/978-3-642-58198-4_3

[26] E. Triantaphyllou, “Multi-criteria decision making methods,” in *Multi-criteria Decision Making Methods: A Comparative Study*. Boston, MA: Springer US, 2000, pp. 5–21. [Online]. Available: https://doi.org/10.1007/978-1-4757-3157-6_2

[27] M. Velasquez and P. T. Hester, “An analysis of multi-criteria decision making methods,” *International journal of operations research*, vol. 10, no. 2, pp. 56–66, 2013.

[28] J. Mustajoki and M. Marttunen, “Comparison of multi-criteria decision analytical software for supporting environmental planning processes,” *Environmental Modelling & Software*, vol. 93, pp. 78–91, 2017.

[29] P. Korhonen and J. Wallenius, “Visualization in the multiple objective decision-making framework,” in *Multiobjective optimization*. Springer, 2008, pp. 195–212.

[30] J. Gettinger, E. Kiesling, C. Stummer, and R. Vetschera, “A comparison of representations for discrete multi-criteria decision problems,” *Decision support systems*, vol. 54, no. 2, pp. 976–985, 2013.

[31] H. Chernoff, “The use of faces to represent points in k-dimensional space graphically,” *Journal of the American statistical Association*, vol. 68, no. 342, pp. 361–368, 1973.

[32] C. Sinhaseni, “Unity Technologies announces Unity Simulation - a new cloud product to train, test, or validate products and services at scale,” *Unity News*, Accessed on: 2020-06-30. [Online]. Available: https://unity.com/our-company/newsroom/unity-technologies-announces-unity-simulation-new-cloud-product-train-test-or

[33] “Unity - our company,” https://unity.com/our-company, Accessed on: 2020-06-09.

[34] S. Axon, “Unity at 10: For better—or worse—game development has never been easier,” *Ars Technica*, Accessed on: 2020-06-09. [Online]. Available: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/

[35] D. Helgason, “Leading Unity into the future,” *Unity Blog*, Accessed on: 2020-06-09. [Online]. Available: https://blogs.unity3d.com/2014/10/22/leading-unity-into-the-future/

[36] J. Brodkin, “How Unity3D became a game-development beast,” *Dice*, Accessed on: 2020-06-09. [Online]. Available: https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/

[37] D. Adams, “No limits – Unity in cross industry development,” *Unity Blog*, Accessed on: 2020-06-10. [Online]. Available: https://blogs.unity3d.com/2014/06/05/no-limits-unity-in-cross-industry-development/

[38] “Unity industrial bundle,” https://unity.com/solutions/automotive-transportation-industrial-bundle, Accessed on: 2020-06-09.

[39] “Sales & marketing,” https://unity.com/solutions/brands-and-creative-agencies, Accessed on: 2020-06-09.

[40] “Film, animation and cinematics,” https://unity.com/solutions/film-animation-cinematics, Accessed on: 2020-06-09.

[41] “Automotive, transportation & manufacturing,” https://unity.com/solutions/automotive-transportation-manufacturing, Accessed on: 2020-06-09.

[42] “Architecture, engineering & construction,” https://unity.com/solutions/architecture-engineering-construction, Accessed on: 2020-06-09.

[43] N. Davis, “How real-time 3D is changing every industry,” *Unity Blog*, Accessed on: 2020-06-10. [Online]. Available: https://blogs.unity3d.com/2019/06/19/how-real-time-3d-is-changing-every-industry/

[44] “Unity’s interface,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/UsingTheEditor.html

[45] “Console window,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/Console.html

[46] “Game objects,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/GameObjects.html

[47] “Colliders,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/CollidersOverview.html

[48] “Rigid body,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/class-Rigidbody.html

[49] “Hinge joint,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/class-HingeJoint.html

[50] “Creating components with scripting,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/CreatingComponents.html

[51] “Scripting overview,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/ScriptingConcepts.html

[52] “Event functions,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/EventFunctions.html

[53] “Order of execution for event functions,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/ExecutionOrder.html

[54] “Extending the editor,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/ExtendingTheEditor.html

[55] “ScriptableObject,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-11, Documentation version: 2019.3. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/class-ScriptableObject.html

[56] “What is .NET?” *Channel 9, .NET Core 101 Series*, Accessed on: 2020-06-12. [Online]. Available: https://channel9.msdn.com/Series/NET-Core-101/What-is-NET

[57] “Tour of .NET,” *Microsoft Documentation .NET*, Accessed on: 2020-06-12. [Online]. Available: https://docs.microsoft.com/en-gb/dotnet/standard/tour

[58] “Async,” *Microsoft Documentation .NET*, Accessed on: 2020-06-15. [Online]. Available: https://docs.microsoft.com/en-gb/dotnet/standard/async

[59] “Parallel programming in .NET,” *Microsoft Documentation .NET*, Accessed on: 2020-06-15. [Online]. Available: https://docs.microsoft.com/en-gb/dotnet/standard/parallel-programming/

[60] “Task Parallel Library (TPL),” *Microsoft Documentation .NET*, Accessed on: 2020-06-16. [Online]. Available: https://docs.microsoft.com/en-gb/dotnet/standard/parallel-programming/task-parallel-library-tpl

[61] “Data structures for parallel programming,” *Microsoft Documentation .NET*, Accessed on: 2020-06-16. [Online]. Available: https://docs.microsoft.com/en-gb/dotnet/standard/parallel-programming/data-structures-for-parallel-programming

[62] J. Ousterhout, “History of Tcl,” *Tcl Developer Xchange*, Accessed on: 2020-06-12. [Online]. Available: http://www.tcl.tk/about/history.html

[63] “Tk backgrounder,” *Tk Docs*, Accessed on: 2020-06-12. [Online]. Available: https://tkdocs.com/resources/backgrounder.html

[64] “Tk concepts,” *Tk Docs*, Accessed on: 2020-06-12. [Online]. Available: https://tkdocs.com/tutorial/concepts.html

[65] “The grid geometry manager,” *Tk Docs*, Accessed on: 2020-06-12. [Online]. Available: https://tkdocs.com/tutorial/grid.html

[66] P. Yoonseok, C. Hancheol, J. Leon, and L. Darby, *ROS Robot Programming (English)*. ROBOTIS, 12 2017. [Online]. Available: http://community.robotsource.org/t/download-the-ros-robot-programming-book-for-free/51

[67] “Nodes,” *ROS Wiki*, Accessed on: 2020-06-15. [Online]. Available: http://wiki.ros.org/Nodes

[68] “Master,” *ROS Wiki*, Accessed on: 2020-06-15. [Online]. Available: http://wiki.ros.org/Master

[69] “URDF tutorials,” *ROS Wiki*, Accessed on: 2020-06-15. [Online]. Available: http://wiki.ros.org/urdf/Tutorials

[70] D. Coleman, I. Sucan, S. Chitta, and N. Correll, “Reducing the barrier to entry of complex robotic software: a MoveIt! case study,” pp. 1–14, 2014. [Online]. Available: http://arxiv.org/abs/1404.3785

[71] “ROS#,” *GitHub Siemens*, Accessed on: 2020-06-15. [Online]. Available: https://github.com/siemens/ros-sharp

[72] “Newtonsoft Json.NET,” https://www.newtonsoft.com/json, Accessed on: 2020-06-18.

[73] “System.Text.Json namespace,” *Microsoft Documentation .NET*, Accessed on: 2020-06-30. [Online]. Available: https://docs.microsoft.com/en-gb/dotnet/api/system.text.json?view=netcore-3.1

[74] “JSON utility,” *Unity Documentation, Unity Manual*, Accessed on: 2020-06-30. [Online]. Available: https://docs.unity3d.com/2019.3/Documentation/Manual/JSONSerialization.html

[75] “Pipe operations in .NET,” *Microsoft Documentation .NET*, Accessed on: 2020-06-22. [Online]. Available: https://docs.microsoft.com/en-gb/dotnet/standard/io/pipe-operations

[76] “pandas,” https://pandas.pydata.org/, Accessed on: 2020-06-18.

[77] “Robot and interface specifications,” *Franka Emika, Franka Control Interface*, Accessed on: 2020-06-25. [Online]. Available: https://frankaemika.github.io/docs/control_parameters.html

[78] “Panda technology,” *Franka Emika*, Accessed on: 2020-06-24. [Online]. Available: https://www.franka.de/technology

[79] C. Gaz, M. Cognetti, A. Oliva, P. Robuffo Giordano, and A. De Luca, “Dynamic identification of the Franka Emika Panda robot with retrieval of feasible parameters using penalty-based optimization,” *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4147–4154, 2019.

[80] “Collaborative robot applications,” *Universal Robots*, Accessed on: 2020-06-24. [Online]. Available: https://www.universal-robots.com/applications/

[81] T. Kunz and M. Stilman, “Turning paths into trajectories using parabolic blends,” *Georgia Institute of Technology*, 2011.

[82] J. J. Craig, *Introduction to Robotics: Mechanics and Control (3rd Edition)*, 3rd ed. Prentice Hall, August 2004.

[83] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*. London: Springer, 2009.

[84] M. W. Spong and M. Vidyasagar, *Robot Dynamics and Control*, 1st ed. USA: John Wiley & Sons, Inc., 1989.

[85] R. Featherstone, “Inverse dynamics — the recursive Newton-Euler method,” in *Robot Dynamics Algorithms*. Boston, MA: Springer US, 1987, pp. 65–77. [Online]. Available: https://doi.org/10.1007/978-0-387-74315-8_4

[86] “Gleichstrommotor,” *studyflix*, Accessed on: 2020-06-30. [Online]. Available: https://studyflix.de/elektrotechnik/gleichstrommotor-1368

[87] J. Kiesbye, “SAINT,” *Chair of Astronautics, TUM*, Accessed on: 2020-06-26. [Online]. Available: https://www.lrg.tum.de/en/lrt/research-at-the-chair-of-astronautics/robotic-operations/saint/

[88] R. Marshall, G. Wood, and L. Jennings, “Performance objectives in human movement: A review and application to the stance phase of normal walking,” *Human Movement Science*, vol. 8, no. 6, pp. 571–594, 1989.

[89] G. Odu and O. Charles-Owaba, “Review of multi-criteria optimization methods–theory and applications,” *IOSR Journal of Engineering (IOSRJEN)*, vol. 3, no. 10, pp. 1–14, 2013.

[90] J. R. Martins and A. B. Lambe, “Multidisciplinary design optimization: a survey of architectures,” *AIAA journal*, vol. 51, no. 9, pp. 2049–2075, 2013.

[91] L. G. Caldas and L. K. Norford, “A design optimization tool based on a genetic algorithm,” *Automation in construction*, vol. 11, no. 2, pp. 173–184, 2002.

[92] B. S. D’Souza and T. W. Simpson, “A genetic algorithm based method for product family design optimization,” in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 36223, 2002, pp. 681–690.

[93] M. Bischoff and K. Daechert, “Allocation search methods for a generalized class of location–allocation problems,” *European Journal of Operational Research*, vol. 192, no. 3, pp. 793–807, 2009.

[94] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[95] R. Saravanan, S. Ramabalan, N. G. R. Ebenezer, and C. Dharmaraja, “Evolutionary multi criteria design optimization of robot grippers,” *Applied Soft Computing*, vol. 9, no. 1, pp. 159–172, 2009.

[96] A. Ghosh and S. Dehuri, “Evolutionary algorithms for multi-criteria optimization: A survey,” 2005.

A Software and Development Environment Versions

The following list summarizes the versions of all software used by and required for the evaluation framework developed in this thesis. Accordingly, the evaluation framework was only tested to run on these versions.

- **Unity:** 2019.3.3f1
- **ROS#:** Version 1.6, Release of 12-20-2019
- **.NET:** .NET Core 3.1.300
- **Newtonsoft Json.NET:** 12.0.3
- **Python:** 3.7.7
- **tkinter:** 8.6
- **Pandas:** 1.0.3
- **NumPy:** 1.18.1
- **Matplotlib:** 3.1.3
- **Virtual Machine:** Ubuntu 16.04.1
- **Ubuntu Kernel:** 4.15.0-106-generic
- **ROS:** ROS Kinetic
- **MoveIt!:** Latest version from pre-built binaries, obtained by running *sudo apt install ros-kinetic-moveit*

The following list summarizes all development environments used to implement all code during the course of this thesis:

- **Development OS:** Windows 10
- **C# and .NET IDE:** Microsoft Visual Studio Community 2019, 16.6.0
- **Python Version Handling:** Anaconda 4.8.3
- **Python IDE:** Spyder 4.1.3

B Additional Results of the Robot Trajectory Evaluation

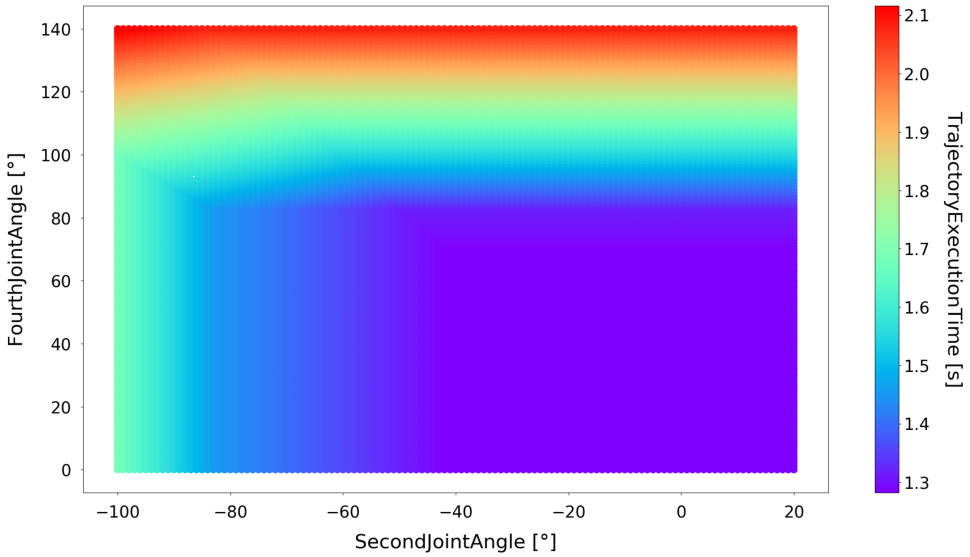

Fig. B–1: Trajectory execution time of all trajectories.

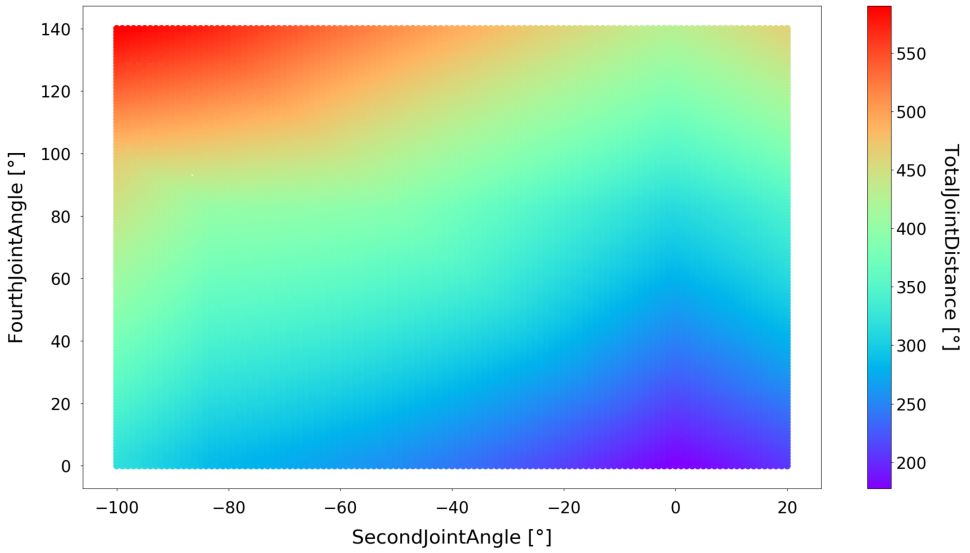

Fig. B–2: Total joint distance covered by the robot for all trajectories.

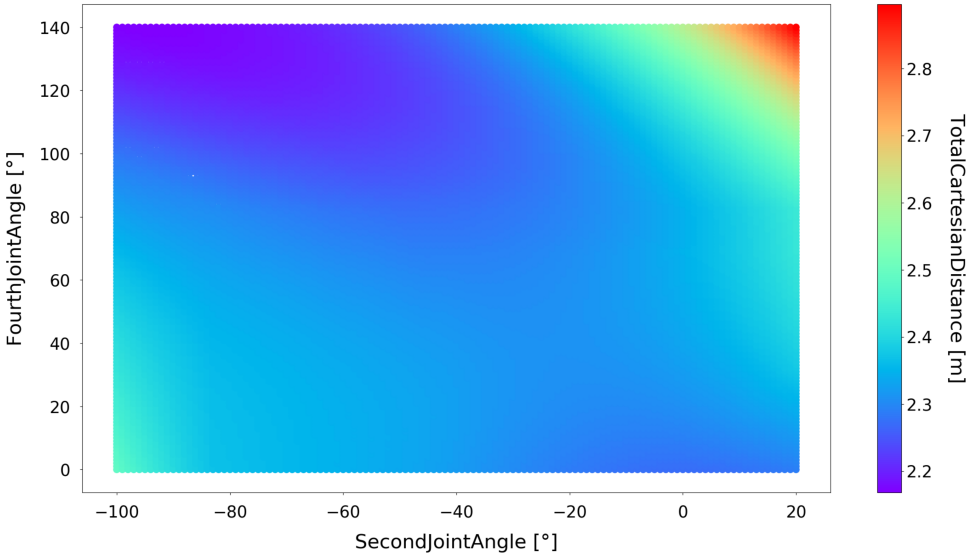

Fig. B–3: Total Cartesian distance covered by the robot for all trajectories.

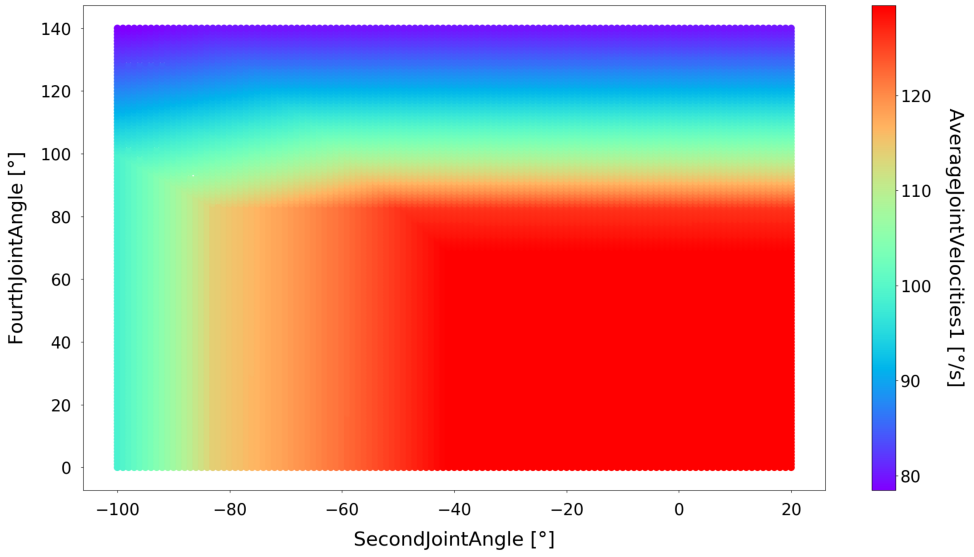

Fig. B–4: Average joint velocity of the first joint for all trajectories.

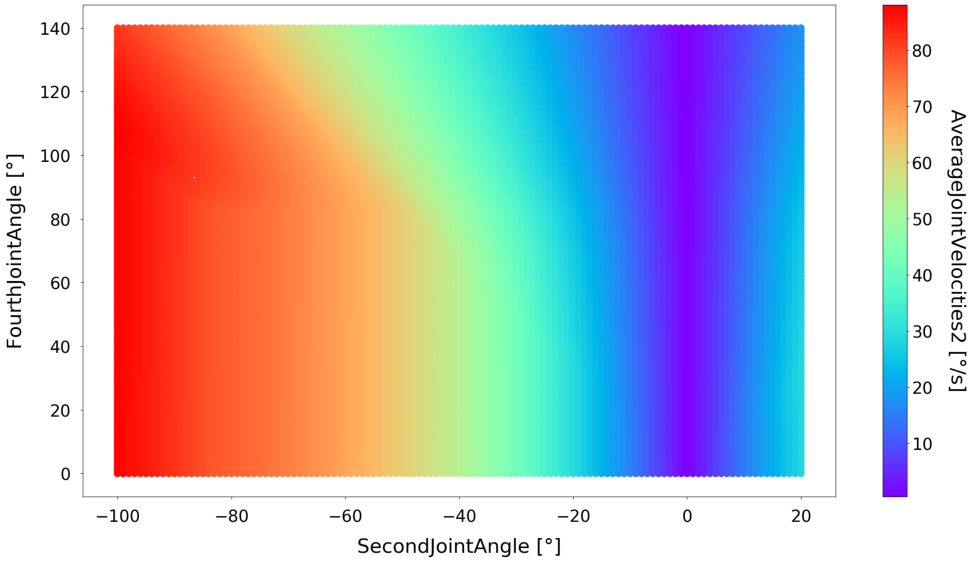

Fig. B–5: Average joint velocity of the second joint for all trajectories.

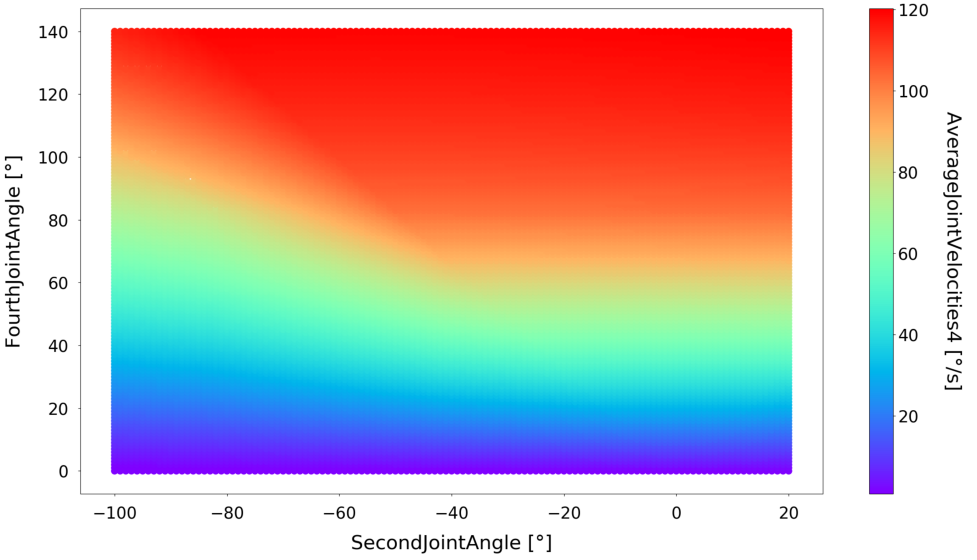

Fig. B–6: Average joint velocity of the fourth joint for all trajectories.

C Additional Results of the Robotic Bin Picking Evaluation

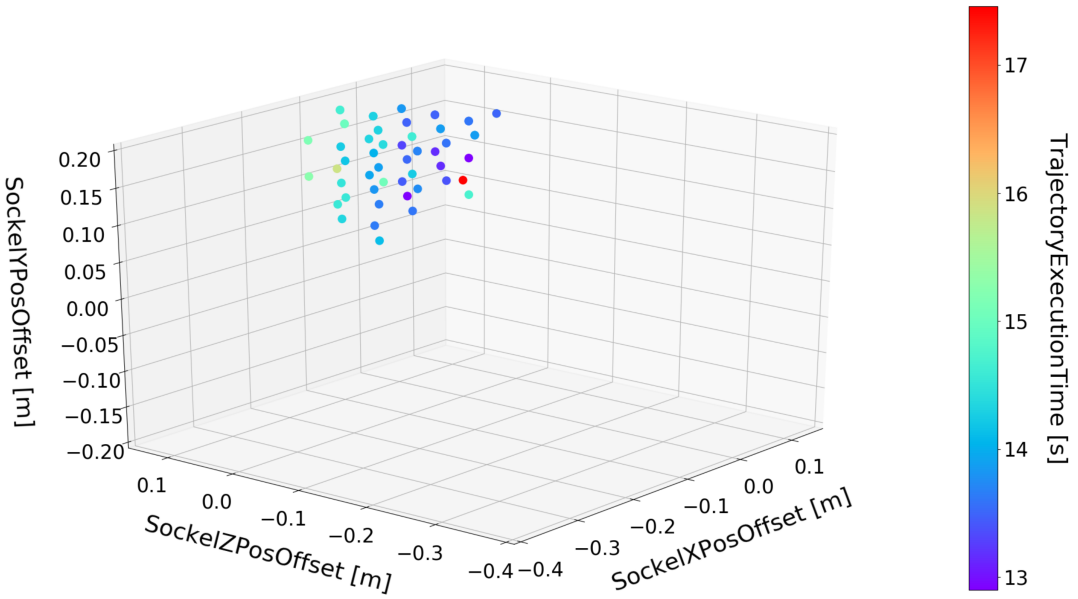

Fig. C–1: Execution time of each bin picking process for all valid positions with the complex box geometry.

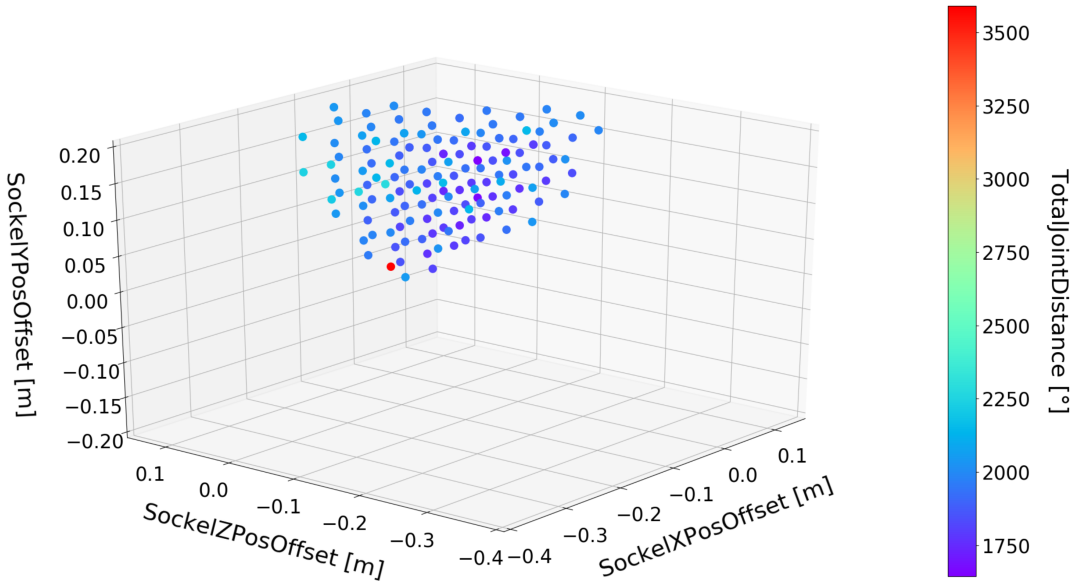

Fig. C–2: Covered joint distance of each bin picking process for all valid positions with the simple box geometry.

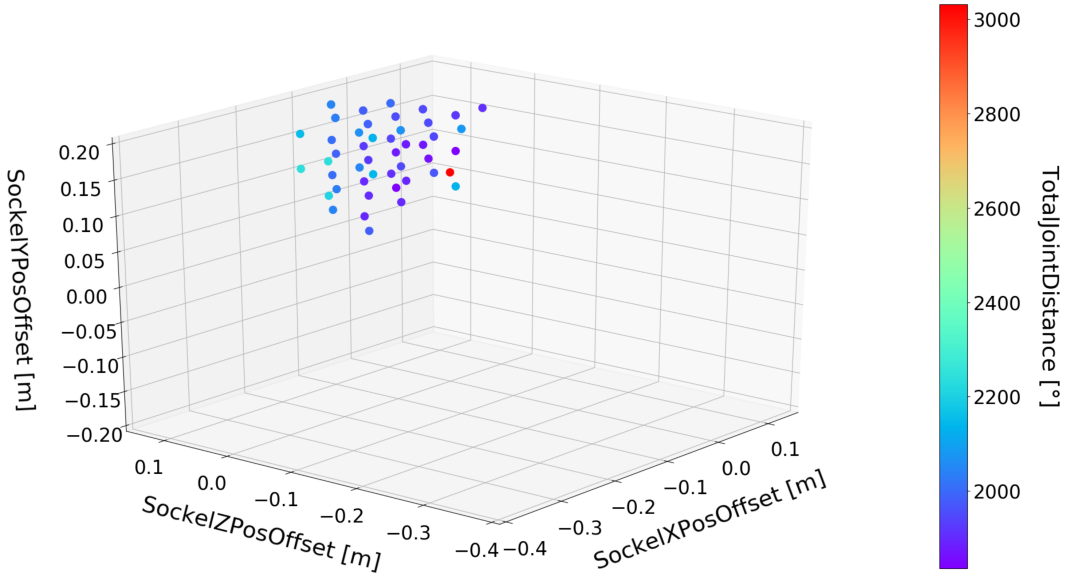

Fig. C–3: Covered joint distance of each bin picking process for all valid positions with the complex box geometry.

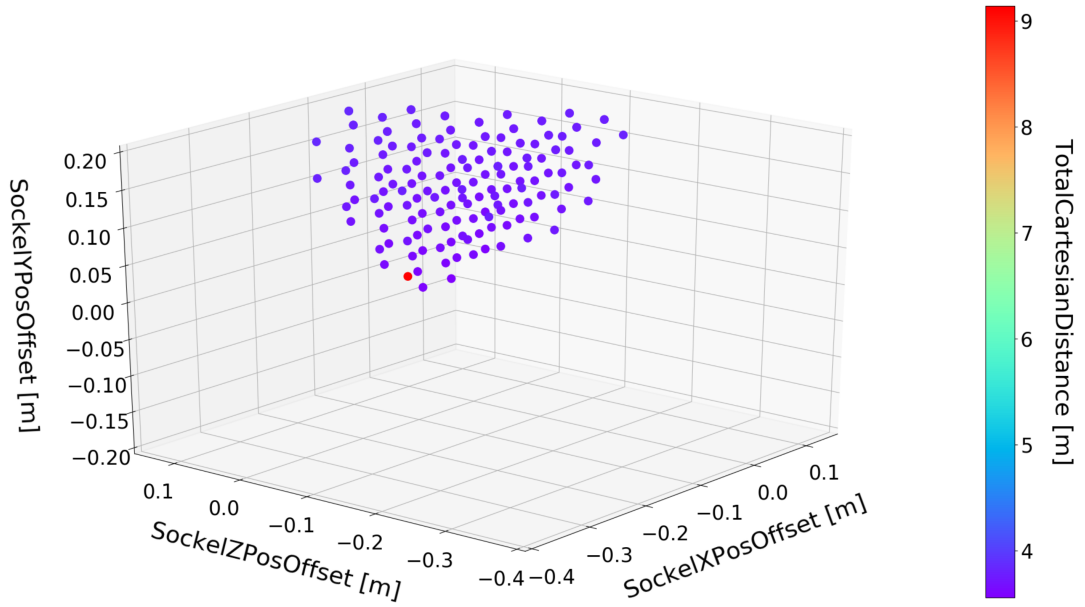

Fig. C–4: Covered Cartesian distance of each bin picking process for all valid positions with the simple box geometry.

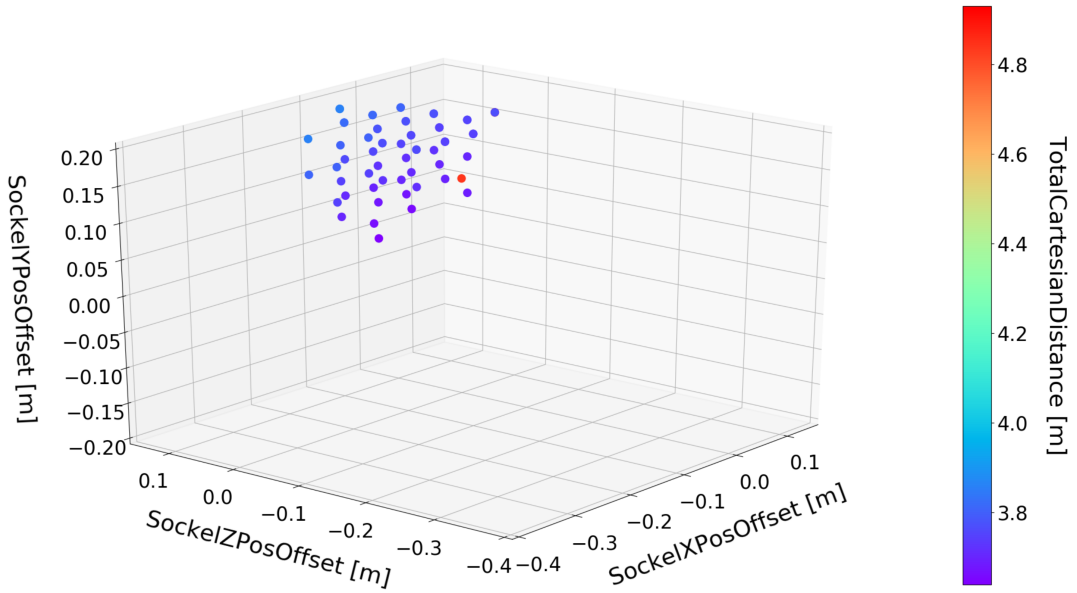

Fig. C–5: Covered Cartesian distance of each bin picking process for all valid positions with the complex box geometry.