University of Bremen

Institute for Artificial Intelligence

Neurosymbolic Robot Programming

A Framework for AI-Enabled Programming of Robot Manipulation Tasks

Benjamin Alt

Vollständiger Abdruck der vom Fachbereich 3 (Mathematik und Informatik) der Universität Bremen zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Nico Hochgeschwender *Universität Bremen*
Erster Prüfer:	Prof. Michael Beetz, PhD *Universität Bremen*
Zweite Prüferin:	Prof. Aude Billard, PhD *École Polytechnique Fédérale de Lausanne*
Beisitzerin:	Prof. Dr. Tanja Schultz *Universität Bremen*

Die Dissertation wurde am 25.10.2024 bei der Universität Bremen eingereicht.
Das Kolloquium fand am 20.02.2025 statt.

Abstract

The vision of robots as intelligent assistants, capable of solving manipulation tasks in domains ranging from household assistance to industrial manufacturing, requires methods for humans to endow them with the cognitive and physical abilities to understand our intents and competently act in accordance with them. The need for capable robot behavior is accompanied by an equal need for control: Pervasive use of robots carries significant safety implications, implying a need for humans to understand robot behavior. This work introduces a neurosymbolic framework for robot programming that combines neural, subsymbolic representations that afford learning and first-order optimization with symbolic representations that afford human interaction and understanding. It introduces Neurosymbolic Robot Programs (NRPs), a dual robot program representation that associates a skill-based, symbolic robot program with a differentiable, predictive model of robot behavior. NRPs bridge the representational divide between symbolic and subsymbolic program representations and serve as a data structure for program synthesis and optimization algorithms that offer powerful artificial intelligence (AI) assistance to human programmers, while ultimately leaving the human in control of robot behavior. This work introduces a family of first-order program optimization algorithms that optimize robot program parameters and low-level motion trajectories with respect to near-arbitrary task objectives and constraints. It also introduces a family of program synthesis systems that generate executable robot programs by leveraging structured representations of task and domain knowledge. Taken together, they form a neurosymbolic programming framework capable of addressing major challenges in programming robots to solve complex, real-world manipulation tasks. The framework and its components are evaluated on tasks ranging from retail and household fetch-and-place to industrial surface treatment and electronics assembly.

Zusammenfassung

Die Vision von Robotern als intelligente Assistenten, die in der Lage sind, Manipulationsaufgaben in allen Bereichen des Lebens und der Arbeit zu lösen, hat eine lange Tradition. Ihre Umsetzung erfordert Methoden, Robotern ermöglichen, die Absichten menschlicher Programmierer zu verstehen und diese kompetent in Handlungen in der echten Welt zu übersetzen. Die Notwendigkeit fähigen Roboterverhaltens geht einher mit einer ebenso großen Notwendigkeit von Kontrolle: Der flächendeckende Einsatz von Robotern birgt erhebliche Risiken im Hinblick auf physische Sicherheit, und erfordert, dass Menschen das Verhalten von Robotern verstehen und gezielt beeinflussen können. Diese Arbeit stellt einen neurosymbolisches Framework für die Roboterprogrammierung vor, der neuronale, subsymbolische Programmrepräsentationen mit symbolischen Darstellungen kombiniert – mit dem Ziel, sowohl datengetriebenes Lernen und Programmoptimierung, als auch Kontrolle durch den Menschen zu ermöglichen. Es werden Neurosymbolische Roboterprogramme (NRPs) vorgestellt, eine duale Programmdarstellung, die ein fähigkeitsbasiertes, symbolisches Roboterprogramm mit einem differenzierbaren, prädiktiven Modell desselben Programms verbindet. NRPs überbrücken die Kluft zwischen symbolischen und subsymbolischen Programmrepräsentationen und dienen als Datenstruktur für Programmsynthese- und Optimierungsalgorithmen, die dem menschlichen Programmierer eine leistungsstarke Unterstützung bieten, während der Mensch letztendlich die Kontrolle über das Roboterverhalten behält. In dieser Arbeit wird eine Familie von Programmoptimierungsalgorithmen erster Ordnung vorgestellt, die in der Lage sind, Roboterprogrammparameter und Bewegungstrajektorien im Hinblick auf nahezu willkürliche Ziele hin optimieren, und gleichzeitig vorgegebene Einschränkungen einzuhalten. Zudem werden Systeme für die Programmsynthese vorgestellt, die unter Nutzung strukturierter Aufgaben- und Domänenrepräsentationen ausführbare Roboterprogramme erzeugen. Zusammen bilden sie ein neurosymbolisches Rahmenwerk, das die Herausforderungen bei der Programmierung von Robotern zur Lösung komplexer, realer Manipulationsaufgaben adressiert. Das Framework und seine Komponenten werden anhand von Szenarien evaluiert, die von Hol- und Bringdiensten im Einzelhandel und Haushalt bis hin zur industriellen Oberflächenbehandlung und Elektronikmontage reichen.

Acknowledgments

This work would not have been possible without the support of my mentors, colleagues, collaborators and companions. Their brilliance, guidance and criticism have shaped the way I think. For this, and for their direct and indirect contributions to this work, they deserve my highest acknowledgments.

I would like to express my deepest gratitude to my supervisor, Michael Beetz, for the intellectual inspiration and invaluable feedback he has given me over the course of my work. His vision of cognitive robotics shaped my approach to robot intelligence and was a steady guidance in a fast-changing field.

Over the last five years, my colleagues at ArtiMinds Robotics have provided me with invaluable support. I would like to thank Darko Katic for having been a good mentor, kind critic and an example in leadership and scientific thinking. I thank Rainer Jäkel for letting me stand on his shoulders and giving me feedback and direction at crucial times, and Sven Schmidt-Rohr for providing me with the trust and room to grow. I would like to thank Andreas Hermann and Gerhard Dirschl for their wealth of technical expertise and personal kindness; Claudius Kienle for being the best office mate, coauthor and brainstorming partner; Oliver Karrenbauer, Felipe Alves, Jens Ochsenmeier, Daniel Kosinski, Michael Weiß and Johannes Zahn for teaching me how to write, build and ship good software, and for joining me in the endeavor to make AI-enabled robot programming a reality; Stefan Fuß and his automation engineers for their help with experiments and all things hardware; and Florian Aumann, Sven Kiesser, Alexander Heilig, Tobias Moritz and the rest of the ArtiMinds team for their feedback and support on and off the clock.

I am grateful to Aude Billard, Tanja Schultz, Nico Hochgeschwender, Daniel Beßler and Luca Krohm for kindly agreeing to serve on my thesis committee, as well as my colleagues at the Institute for Artificial Intelligence (IAI) for their support, feedback and inspiration. I would like to thank my co-authors and scientific collaborators for their contributions to this research, especially Asil Kaan Bozcuoğlu, Andrei Haidu, Franklin Kenghagho Kenfack, Julia Dvorak, Florian Stöckl, Silvan Müller, Julian Raible and Christopher Braun; and I would like to thank my students, who have brought many insights and new perspectives to my thinking and provided invaluable help with programming and experiments, notably Martin Hetz, David Lerch, Anika Gieringer, Urs Keßner, Lukas Ringle, Tobias Winter, Cristian Gorun and Xavier Bustamante.

Finally, I would like to express my gratitude to my parents, Manfred Alt and Anne Plein-Alt, for always having supported my growth and learning; as well as to Ryan, Tamara, Titus, Yael and Marta for being anchors in a dynamic time.

Karlsruhe, 25 October 2024
Benjamin Alt

This work has received funding from the German Ministry of Education and Research (BMBF) as part of the projects ILIAS (grant #01DR19001B) and KARL (grant #02L19C255); from the German Ministry for Economic Affairs and Climate Action (BMWK) as part of the projects GANResilRob (grant #01MJ22003B), EASY (grant #01MD22002B) and VADER (grant #13IK026A); from the German state of Baden-Württemberg as part of the project RoboGrind (grant #BW1 0079/01); from the German Research Foundation DFG as a part of the EASE Collaborative Research Center (CRC #1320); and from the European Union as part of the project euROBIN (grant #101070596).

Contents

List of Figures ix

List of Tables xiii

1 Introduction 3
- 1.1 AI-Enabled Robot Programming . . . 5
- 1.2 Contributions . . . 11
- 1.3 Outline . . . 12

2 A Neurosymbolic Robot Program Representation 15
- 2.1 A Question of Representation . . . 15
- 2.2 Neurosymbolic Robot Programs: Overview . . . 20
- 2.3 Source Program Representations . . . 27
- 2.4 Differentiable Shadow Programs . . . 34
- 2.5 Differentiable Motion Planning . . . 54
- 2.6 Related Work . . . 64
- 2.7 Discussion . . . 79
- 2.8 Conclusion . . . 85

3 First-Order Robot Program Parameter Optimization 87
- 3.1 Optimization of Robot Program Parameters via Inversion of Differentiable Shadow Programs . . . 89
- 3.2 Lifelong Learning and Optimization in Stochastic Environments . . 109
- 3.3 Joint Optimization of Task Parameters and Motion Trajectories . . 129
- 3.4 Discussion . . . 144
- 3.5 Conclusion . . . 146

4 Interactive AI-Enabled Robot Program Synthesis 149
- 4.1 Knowledge-Driven Robot Program Synthesis . . . 152
- 4.2 An Interactive Robot Programming Assistant . . . 186
- 4.3 Prompt-based Program Synthesis with Large Language Models . . 206
- 4.4 Discussion . . . 215
- 4.5 Conclusion . . . 217

5 A Framework for Neurosymbolic Robot Programming 219
5.1 A Framework for Neurosymbolic Robot Programming 220
5.2 Related Work 230
5.3 Discussion 233
5.4 Conclusion 235

6 Conclusion 237
6.1 Summary 237
6.2 Discussion 239
6.3 Outlook 243
6.4 Conclusion 245

Bibliography 247

Appendix 299

A Differentiable Shadow Programs: Predicted Trajectories 301

B Dialogue Transcripts 303

C Prompts 305

D List of Publications 311

List of Abbreviations 315

List of Figures

1.1 Robot programming as bidirectional communication 4
1.2 Examples for retail fetch-and-place and industrial peg-in-hole tasks 6
1.3 Neurosymbolic robot programming 11

2.1 Neurosymbolic Robot Programs 16
2.2 A dual representation of neurosymbolic robot programs 25
2.3 Task tree of a peg-in-hole task and examples for two industrial peg-in-hole applications 32
2.4 Structure of a robot source program for a peg-in-hole task 32
2.5 Batched tensor representation of trajectories 36
2.6 Stacked GRU sequence-to-sequence neural architecture 39
2.7 Shadow skill DCG for a spiral search 41
2.8 Prior, predicted and ground truth Cartesian spiral trajectories . . . 41
2.9 Ground truth, prior and posterior Cartesian end-effector trajectories 43
2.10 An NRP for an industrial peg-in-hole task 49
2.11 An NRP for a tactile probe search task 50
2.12 The DGPMP2-ND motion planner 58
2.13 Collision model by cylindrical approximation for a UR5 arm 60
2.14 Computation of the DGPMP2-ND collision error vector for a robot link 60

3.1 The SPI program parameter optimization algorithm 88
3.2 Optimization of force-controlled contact with SPI 95
3.3 Empirical process metrics for different SPI task objectives, relative to oracle and random baselines 98
3.4 Spiral search motions optimized by SPI for different task objectives 99
3.5 Exemplary baseline and optimized spiral trajectories 100
3.6 VR human demonstration for a household pick-and-place task . . . 101
3.7 Optimization of pick-and-place program parameters with respect to a human VR demonstration 103
3.8 Posterior trajectory during optimization of pick-and-place program parameters with respect to a human VR demonstration 104
3.9 Real-world execution of a fetch-and-place task optimized with respect to a VR human demonstration 104

3.10 Closed-loop SPI for lifelong robot program parameter optimization in nonstationary environments 111
3.11 Robot program and hardware setup for search-based valve assembly 115
3.12 Optimization of spiral search parameters in the presence of stationary process noise 116
3.13 Robot program and hardware setup for THT assembly 120
3.14 Optimization of probe search patterns for nonstationary noise processes (linear drift and Brownian motion) 123
3.15 The SPI-DP program parameter and motion trajectory optimizer . 130
3.16 SPI-DP performs joint parameter and trajectory optimization by double-loop gradient descent over the shadow program DCG . . . 132
3.17 RGB and depth images of a human demonstration of a household pick-and-place task 134
3.18 10 human demonstrations and 4 exemplary trajectories planned by SPI-DP for a household pick-and-place task; real-world execution of the optimized robot program 135
3.20 4 exemplary trajectories planned by SPI-DP for placing a wine glass into a cupboard; real-world execution of the optimized robot program 136
3.21 Collision, prior trajectory and start pose factors during DGPMP2-ND optimization for four different target poses, given the same human demonstration 137
3.19 Source program for a household pick-and-place task 138
3.22 Trajectory planned by SPI-DP for a poka-yoke QA task; real-world execution of the optimized robot program 139
3.23 Source program for a poka-yoke QA task 139

4.1 The MetaWizard family of robot program synthesis systems 151
4.2 High-level overview of the MetaWizard system for knowledge-driven program synthesis 153
4.3 Semantic maps of VR and real-world supermarket environments . 156
4.4 The semantics of events, tasks and situations modeled by SOMA . . 160
4.5 Event timeline of a VR NEEM 165
4.6 Grounded object designators and ARTM robot program for “inverse” peg-in-hole 170
4.7 Object designator grounding with RoboSherlock 171
4.8 VR human demonstrations for force-controlled fetch-and-place and peg-in-hole 173
4.9 Laboratory and realistic supermarket environments 175
4.10 Simulated and real-world execution of a generated robot source program for force-controlled fetch-and-place in a lab environment 176
4.11 Simulated and real-world execution of a generated robot source program for force-controlled peg-in-hole in a lab environment . . . 178

4.12 Simulated and real-world execution of a synthesized robot program for a supermarket fetch-and-place task in a realistic environment . 179
4.13 Simulated and real-world execution of a synthesized robot program for a force-controlled peg-in-hole task in a realistic environment . . 180
4.14 MetaWizard2 co-creates executable robot programs by interactive dialogue with a human programmer 187
4.15 The MetaWizard2 metatask representation 190
4.16 MetaWizard2 grounds symbols in its knowledge base by natural-language interaction with a human programmer 193
4.17 The RoboGrind experiment for robot program synthesis for a wind turbine sanding task 194
4.18 Scanned point cloud and annotated defect regions of a wind turbine blade section and planned tool path for sanding 195
4.19 Measured force trajectories for different controller parameterizations 197
4.20 Surface roughness before and after sanding for two different, filled blade segments 198
4.21 Hardware setup for wind turbine refabrication with MetaWizard2 . 199
4.22 Simulated and real-world experiment setup for gear deburring with MetaWizard2 200
4.23 Simulated and real-world experiment setup for sanding of rotational molds with MetaWizard2 201
4.24 The MetaWizardLLM robot program synthesis system 207
4.25 The CAD grounding module of MetaWizardLLM 209
4.26 Experiment setup and generated underspecified plan for a gear assembly task 211
4.27 Execution of the robot program generated by MetaWizardLLM . . 212

5.1 Lifecycle of an industrial robot program 220
5.2 The BANSAI conceptual framework for neurosymbolic robot programming 222
5.3 An explanation user interface for the SPI program optimization workflow 224
5.4 SPI XUI: Visualization of dataset and data quality 225
5.5 SPI XUI: Visualization of optimized trajectories in guided and expert modes 226
5.6 A software framework for AI-enabled robot programming 230

A.1 Ground-truth, prior and posterior Cartesian end-effector trajectories predicted by shadow skill $\bar{p}$ for a spiral search task 301
A.1 Ground-truth, prior and posterior Cartesian end-effector trajectories predicted by shadow skill $\bar{p}$ for a spiral search task 302

List of Tables

3.1 Optimization of contact motions for different skill frameworks and surfaces . . . 96
3.2 Comparison of sequential transfer learning with meta learning alternatives for NRP training . . . 119
3.3 Failure rates for different regularizers and stochastic processes . . . 124
3.4 Success rates and cycle times for force-controlled poka-yoke testing of screw holes before and after optimization . . . 138

4.1 Quantitative program synthesis results for a force-sensitive fetch-and-place task . . . 175
4.2 Quantitative program synthesis results for a force-controlled peg-in-hole insertion task . . . 177
4.4 Root MSE between the planned tool path and the detected surface point cloud . . . 197
4.3 Evaluation of the surface roughness after filling (R_{a1}) and after sanding (R_{a2}) for four different parameter sets . . . 198

List of Code Listings

2.1 Differentiable implementation of a Cartesian spiral generator . . . 38
2.2 Forward pass through a differentiable shadow skill 42
2.3 Differentiable inverse kinematics for serial N-degree of freedom (DoF) manipulators 46
2.4 Forward pass through a differentiable shadow program 53
2.5 The DGPMP2-ND optimization loop 63
3.1 SPI iteratively optimizes the input parameters `x` of a DCG `p` with respect to task objectives `phi` by gradient descent 91
4.1 Example for a KnowRob knowledge base in RDF Turtle syntax . . . 154
4.2 Dynamic knowledge assertion and retraction in the KnowRob knowledge base 157
4.3 Predicates defined with KnowRob's projection and retrieval operators 157
4.4 Definition of a `soma:PickingUp` task in terms of pre-, runtime and postconditions 161
4.5 Definition of an `artm:InsertHoleOntoPeg` task in terms of pre-, runtime and postconditions 162
4.6 Semantic reasoning routines in the Knowrob query language . . . 162
4.7 The predicate `interprets_to` enables the abstraction from a given action to the intended task via the Prolog unification algorithm . . 167
4.8 Abstraction from a demonstrated sequence of action to a candidate task sequence via the Prolog unification algorithm 168
4.9 PyCRAM description of a designator for "inverse" peg-in-hole . . . 169
4.10 Metatask definition of a sanding task via Prolog predicates in the semisymbolic knowledge base 190
4.11 Transcript of a dialogue between MetaWizard2 and a human programmer for a deburring task 200
4.12 High-level pseudocode for the MetaWizardLLM program synthesis and task grounding steps 208
B.1 Transcript of a dialogue between MetaWizard2 and a human programmer for a wind turbine blade refurbishing task 303
B.2 Transcript of a dialogue between MetaWizard2 and a human programmer for sanding a rotational mold 303

B.3 Transcript of a dialogue between MetaWizardLLM and a human programmer for a gear assembly task 304
C.1 Prompt for extracting numerical parameters from Llama3 305
C.2 Prompt for parsing chosen options with Llama3 306
C.3 Generated prompt for gear assembly with MetaWizardLLM 307

CHAPTER 1

Introduction

From the inception of the field, one of the central visions driving research and development in robotics has been that of an embodied universal assistant, capable of performing open-ended tasks in real-world environments. Modern robot hardware – accurate manipulators with redundant kinematics, high-resolution 3D cameras, flexible and precise end-effectors – provide robots with wide-ranging physical abilities to interact with their environment and make this vision appear, increasingly, within our grasp. As improved *physical* capabilities permit engineers to apply robots to increasingly complex tasks, however, the demands on robot software increase correspondingly. Robots must be endowed with the *cognitive* abilities to make sense of unstructured environments, plan motions to solve challenging manipulation problems or learn from past experience to improve their own behavior over time. At the same time, humans must have a way to instruct robots what to do, at a level of abstraction that allows the expression of increasingly abstract goals – "clear the table" rather than "put the blue cup into the dishwasher" – for increasingly complex tasks, in a way that remains simple and concise. Robot programming involves the dual problem of instructing the robot *how* to use its sensors and actors to perform a task, as well as specifying *what* the robot ought to do – what goals it should pursue and what constraints it must respect.

Programming is bidirectional communication between the human programmer and the programmed system: The programmer communicates an idea to the system in the form of a program, describing a problem to be solved (the *what*) and a strategy to achieve them (the *how*), with the aim of letting the system perform the work of actually solving the problem. The system, in turn, performs work to produce a result, which is communicated back to the programmer and can be interpreted by them in light of the program (Abelson et al. (1996) and Sussman (2005); see Figure 1.1). In the context of robot programming, the programmer provides the robot with a program containing some representation of the task at hand as well as knowledge, such as a set of robot skills, that enable the robot to solve it; they (or someone else) then observe the behavior of the robot and interpret it, using the program to understand what the robot has just done, and how it has done

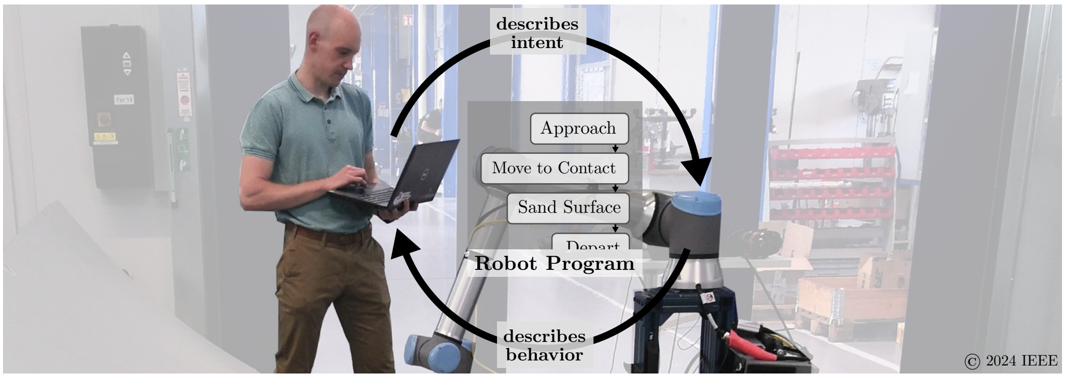


Figure 1.1: Robot programming as bidirectional communication between programmer and robot (Alt et al., 2024c).

it (Knuth, 1984). For complex tasks, humans cannot fully specify the intended robot behavior. Instead, real-world robot programs are generally *underspecified*, requiring some degree of intelligence on the part of the robot to "understand" the human intent, infer missing information from past experience or available sensory data, and plan a sequence of low-level actions to materialize the intended behavior in the world (Nyga et al., 2018). Conversely, interpreting the observed behavior of an intelligent robot is challenging for human programmers, as the robot's behavior is the result of cognitive processes such as learning or planning that operate on representations that afford tractable computation, but may not afford human interpretation. Particularly in embodied domains such as robotics, where the programmed system interacts with the physical world, a causal, or at least mechanistic, understanding of robot behavior is crucial to ensure safety and foster trust – doubly so in applications in which the robot faces novel situations that have not been part of its training regime. Beyond understanding, safe operation of robots requires humans to be in *control* of robot behavior, able to predict and modify it to prevent and correct errors.

Research question ▷

This work, then, addresses the following fundamental research question: **How can robots be programmed to tractably solve complex tasks in real-world environments, while leaving humans in control of robot behavior?**

One core observation made in this work is that the difficulty in programming artificially intelligent robots arises from the fact that humans and robots require seemingly incompatible program representations: Implicit, subsymbolic representations have proven essential for capable robot learning and planning, but explicit, symbolic representations facilitate human interpretability and control. Based on this observation, this work proposes a new paradigm of AI-enabled robot programming which rests on the fundamental hypothesis that **bridging the apparent divide between implicit and explicit program representations can enable robots to learn and optimize their behavior, while allowing humans to understand and control it.** This work presents a framework which realizes this paradigm by com-

Working hypothesis ▷

bining technologies that facilitate data-driven learning and optimization with a fundamentally human-centric model of robot programming.

1.1 AI-Enabled Robot Programming

The challenges facing robot programmers in complex real-world manipulation tasks – endowing robots with an understanding of what they ought to do, and how they can go about doing it – can be addressed by leveraging AI as an integral part of the programming process. From a programmer's perspective, AI promises to simplify programming via learning and optimization, but also imposes new challenges, such as ensuring the safety of robot behavior resulting from AI-generated robot programs.

1.1.1 Learning and Self-Optimization

Recent advances in deep learning (DL) have enabled breakthrough improvements in a wide array of domains ranging from machine translation (Scao et al., 2023) and creative writing (Wang et al., 2024a) to image generation (Betker et al., 2023), computational law (Chalkidis et al., 2020), medical diagnostics (Swanson et al., 2023) and academic research itself (Truhn et al., 2023). There is mounting evidence that the algorithms and data structures which enable disembodied digital systems to represent vast bodies of latent knowledge, and learn to use that knowledge to solve complex problems, can also provide the foundations to endow embodied digital systems – robots – with powerful cognitive abilities. Learning, the "ability for a system to improve its performance over time through the acquisition of knowledge or skill" (Vernon, 2022) has been identified as a core component of artificially intelligent systems (Russell and Norvig, 2021). All general-purpose robot cognitive architectures incorporate learning mechanisms (Laird et al., 1987; Vernon et al., 2011; Sun, 2017; Kotseruba and Tsotsos, 2020; Peller-Konrad et al., 2023; Beetz et al., 2023). Advances in deep unsupervised and Reinforcement Learning (RL), combined with general-purpose neural architectures, have shown to be applicable to problems in robotics such as manipulation, navigation and task planning (Kroemer et al., 2021). They promise an avenue toward robots capable of autonomously learning *how* to perform complex tasks by inferring task semantics and dynamics from data, robustly perceiving and understanding their environment, and planning long-horizon manipulation sequences to achieve their goals. At the same time, recent advances in multimodal learning and generative AI permit digital systems to parse and output natural language with near-human performance (Sejnowski, 2023) and jointly reason over data combining multiple modalities, such as vision and language (Li et al., 2024). Multimodal DL models promise natural human-robot interaction through language or demonstration, allowing for novel interactive programming paradigms that permit humans to intuitively specify *what* tasks are to be performed.

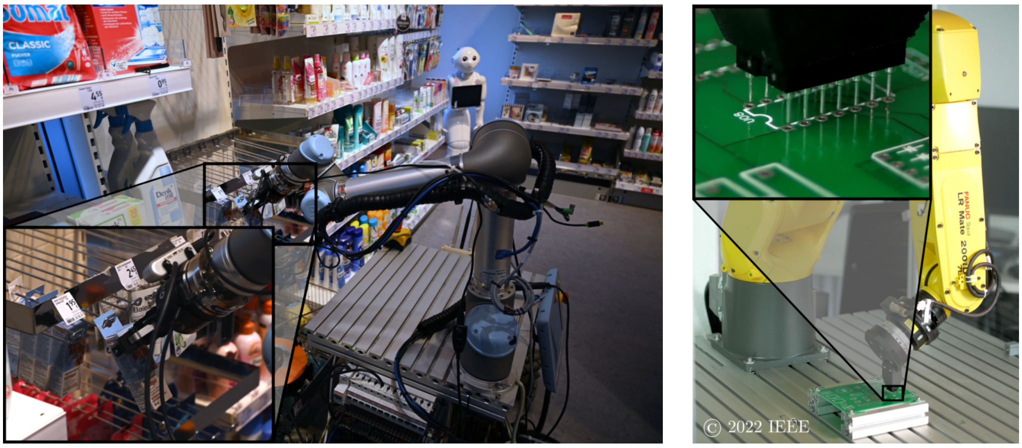


Figure 1.2: Examples for retail fetch-and-place (left, Alt et al. (2023)) and industrial peg-in-hole tasks (right, Alt et al. (2022b)), which serve as running examples in this work.

1.1.1.1 Case Study: Retail Assistance

Consider a robotic shopping assistant for people with restricted mobility, such as the elderly, that fetches objects from supermarket shelves and places them in a shopping basket (see Figure 1.2, left). Under real-world conditions, this is a deceptively complex task. Typical supermarkets feature a variety of different products, each of which have different appearances, affordances and physical properties. To competently solve this task, a robot must be capable of robustly perceiving objects of various sizes and materials and grasping objects of different shapes and weights. Moreover, retail environments are semi-structured: While the locations of most objects are a priori known, objects are frequently misplaced and often not immediately reachable. Retail fetch-and-place requires perception and manipulation capabilities that are robust against uncertainty, occlusions and the presence of unexpected objects. Lastly, retail environments are dynamic, with humans, shopping carts or pets moving in the scene. Dynamic environments require flexible cognitive mechanisms for replanning motions and actions in the moment. Programming an assistance robot for retail fetch-and-place requires endowing it with perception, cognition and action abilities that are highly generalizable and can be applied to a large variety of objects and surroundings, but afford sufficiently precise perception and manipulation for the concrete objects and environment at hand (Alt et al., 2023). DL approaches permit agents to learn perception, motion and planning skills for complex manipulation tasks in semi-structured environments, taking the burden of explicitly programming such skills off the human programmer.

1.1.1.2 Case Study: Electronics Assembly

Service robotics tasks such as retail fetch-and-place are challenging due to the unstructured nature of the environment. In industrial robotics, the core challenges often lie in the task itself. Many industrial tasks have low tolerances which are due to product requirements (such as maximally permissible gap sizes in automotive assembly tasks). Printed circuit board (PCB) assembly serves as an illustrative use case, as sensitive components must be placed with submillimeter tolerances (see Figure 1.2, right). Particularly for through-hole technology (THT) processes, components must not only be placed so that the pins align with the holes on the PCB, but any contact, search or insertion motions must not cause forces exceeding given thresholds to avoid bending the pins or damaging the surface of the board. Beyond manufacturers' tolerances, industrial applications generally have high robustness requirements with respect to cycle time and task success. In high-volume processes, subsecond delays, such as waiting for I/O signals or searching for a hole on a PCB, can cause large costs to the producer. In PCB assembly, recovery from errors is impossible or incurs time costs, as damaged components must be discarded. Programming intelligent industrial robots chiefly consists of solving multicriterial, constrained optimization problems such as minimizing cycle time and maximizing the probability of task success subject to smoothness or collision constraints. DL approaches permit robots to learn models of robot tasks from production data (Alt et al., 2021). With learned models, robots can automatically generate behavior to solve manipulation problems or optimize their own behavior over time to improve efficiency and robustness (Alt et al., 2022b).

1.1.2 The Role of the Human Programmer

Deep learning owes its success to the leveraging of highly general learning algorithms and data structures that give rise to implicit, latent representations of knowledge and skills (Sutton, 2019). The capabilities of DL models have been shown to scale with the amount of computation and data available for training and inference (Kaplan et al., 2020). Recent large models are capable of solving challenging robot programming tasks, such as program synthesis for manipulation in unstructured environments (Zitkovich et al., 2023). Integration of DL into robot programming profoundly changes both the roles of the human and the robot in robot programming: Instead of mechanistically specifying robot behavior, the programmer acts as a teacher and guide, providing a general-purpose learning algorithm with data and constraints. Instead of automating the human programmer out of the programming process, practical AI-enabled robot programming requires the human to assume a crucial role in robot programming. Empirical studies of neural scaling laws suggest that model performance increases as a power law with diminishing returns relative to model size, dataset size, and computational resources (Kaplan et al., 2020; Hoffmann et al., 2022; Thompson et al., 2022). Embodied domains face the additional challenge of acquiring real-world training data, limited by physical constraints. Integrating the human into the learning process can help provide higher-level semantic examples (Mosqueira-Rey et al., 2023), reduce the amount of required data (Monarch, 2021), and guide learners toward solutions aligned with human preferences (Christiano et al., 2017). The scalable and sustainable building and deployment of learning systems requires principled ways of leveraging human expertise as a resource to support and guide the learning process (Brooks, 2019).

1.1.3 Control

When robot behavior is, at least in part, a learned function of current sensory input, past experience and latent knowledge, the programming of intelligent robots requires robust mechanisms to ensure that that behavior is *safe*, particularly when robots operate in proximity to humans. In the context of intelligent systems, safe behavior can be defined by the following characteristics (Rueß and Burton, 2022):

1. **Correctness:** The behavior of the system must *comply with its specification*. In the case of a retail assistance robot, this implies that the robot reliably performs the tasks it is instructed to perform, and exclusively these tasks: If the robot is instructed to pick up three cans of soup from a shelf and place them in a shopping cart while avoiding collisions with other objects or itself, correct robot behavior implies that after completion of the task, three cans of soup have been transferred from the shelf into the cart, and no collisions occurred in the process.

2. **Intent:** The specification of the system's behavior must *reflect the intent of the user*. For a retail assistant, this implies the need for a mechanism to translate the desire of a human user to have three cans of soup transferred from a shelf to a shopping cart into a program that accurately captures this intent, and permits the robot to take actions to fulfill it.

3. **Acceptability:** The behavior of the system must be *acceptable to the human user*. In particular, the system must not cause unacceptable harm (Rueß and Burton, 2022). A retail assistance robot, for example, must not injure humans or damage itself or objects in its environment while performing a task.

Recent AI safety research has shown that AI systems whose behavior is governed by subsymbolic, implicitly represented policies face significant safety challenges (Raji and Dobbe, 2023). In particular, placing guarantees on system behavior – ensuring correctness – is challenging, as deriving formal proofs about the behavior of deep neural networks may be computationally intractable (Cooper and Marques-Silva, 2023). The alignment problem – ensuring that implicitly represented policies encode the intents of human users – has likewise proven challenging on algorithmic (Ngo et al., 2023) and philosophical grounds (Gabriel, 2020), even for comparatively small DL systems (Shah et al., 2022). In response to these challenges, AI safety research has proposed that AI systems must exhibit two central characteristics: They must be *explainable* to human users, and they must afford humans *control* over the system.

Explainability Explainability has been identified as a central property of safe AI systems. Gilpin et al. (2018) define explainability via the concepts of *interpretability* and *completeness*: An interpretable AI system affords the description of "the internals of [the] system in a way that is understandable to humans" – interpretable systems afford *explanations* – and an explanation is is "more complete when it allows the behavior of the system to be anticipated in more situations" (Gilpin et al., 2018). Explainability fosters AI safety by allowing humans to understand, verify or certify that the system's behavior is correct and respects the user's intent. Moreover, explainability has been considered a requirement for AI use in applications with physical safety implications such as industrial robotics (Sofianidis et al., 2021) and to foster acceptance of intelligent robots by human colleagues (Theis et al., 2023). Due to their reliance on subsymbolic, implicit representations, robot programs generated by deep learning methods are generally not explainable (Burkart and Huber, 2021). For many practical robot applications, however, it is sufficient for the resulting robot behavior to be explainable, even if the mechanisms by which it was generated are not. A framework for programming intelligent robots should then ensure that the resulting robot behavior is explainable, even if it was generated by opaque learning, optimization or synthesis mechanisms. Chapter 5 proposes such a framework.

Control In the AI safety literature, AI control refers to "ensuring that AI systems try to do the right thing, and in particular that they don't competently pursue the wrong thing" (Christiano, 2021). While black-box approaches for controlling existing AI systems have been proposed (Bowman et al., 2022), the degree of control afforded by an AI system is largely determined by architectural choices made during the design and implementation of the AI system: "How to design AI systems such that they do what their designers intend" (Bostrom, 2017). For embodied systems such as robotic retail assistants or industrial robots operating in real-world environments, the AI control problem takes on a physical dimension with immediate safety implications for the humans or objects they interact with. From this perspective, it becomes paramount that the methods by which intelligent robots are programmed enable humans to ensure that they do "the right thing". Explainable systems greatly facilitate control, both by human programmers during the development process (Burkart and Huber, 2021) as well as by automatic or manual validation and verification before or during deployment or operation (Hurault and Marques-Silva, 2023). In robot applications, a crucial aspect of control is the ability of humans to influence the behavior of the AI system after deployment, particularly in response to errors or unforeseen circumstances. A framework for programming intelligent robots should therefore allow for reprogramming or reparameterization at runtime by human users. Chapter 5 introduces a robot programming paradigm which allows for the learning and optimization of robot programs while allowing humans to understand and control robot behavior. On the often-cited continuum between AI capability and control (Yampolskiy, 2022; Jensen et al., 2023; Wolf et al., 2024), it proposes an equilibrium which allows for data-driven, first-order, DL-based learning and optimization, while using an explainable program representation for execution.

1.1.3.1 AI Control in Service and Industrial Robotics

Many service robots, such as retail assistants, share physical spaces with humans. This imposes strict safety requirements on the programs governing the robot's behavior, as hurting humans must be avoided under all circumstances. To safely deploy robotic shopping assistants, robot programmers must be able to make hard guarantees about the robots' behavior. To this end, they must be able to interpret the policies driving the behavior of the robot, which is challenging, if not intractable, with most DL-based policy representations. Beyond the immediate safety aspects, interpretability of the rules governing the robot's behavior is essential for fostering trust, which in turn is a requirement for the adoption of embodied, intelligent assistants in the immediate surroundings of humans. For these reasons, safely programming and deploying robots in real-world service contexts requires a novel paradigm of robot programming that permits human involvement in the programming process for scaffolding and safety assurance while at the same time leveraging powerful, subsymbolic learning and planning approaches.

Like service applications, many industrial applications have high safety requirements due to the use of fast industrial end effectors, possibly dangerous tools such as lasers, or the need for safe human-robot collaboration. These requirements imply AI-based robot programming solutions to provide a high degree of control over robot behavior. Human programmers must be able to modify industrial robot programs in a targeted way, e.g. when changes to the hardware setup are made in the context of maintenance, and human auditors must be able to ensure the safety of the robot program after initial creation and modification. Moreover, industrial robot applications in high-risk industries such as the aerospace or automotive sectors must undergo strict certification before production can begin. To facilitate certification and safe maintenance after deployment, large corporations often impose company-wide robot programming standards (Akcay, 2016). Safety certification and compliance with industry standards require robot programs to afford some degree of explanation, and to afford making some verifiable statements about robot behavior.

1.1.4 A New Robot Programming Paradigm

It appears, then, that the subsymbolic, implicit representations that facilitate learning and self-optimization of robot behavior seem ill-suited to human understanding and control. A good programming paradigm should afford the highest degree of capability possible without sacrificing control, and vice versa: "[I]t should be easy to conceive programs, it should be easy to convince oneself that a program is correct and that the machine working under its control will indeed produce the desired result" (Dijkstra, 1971). A framework for programming intelligent robots should

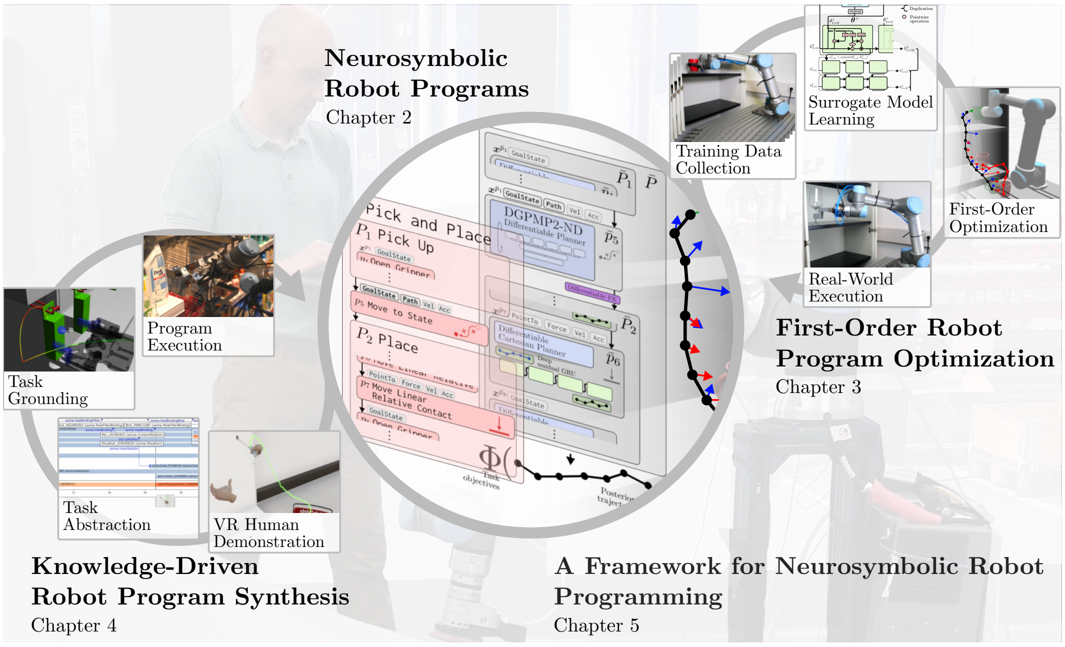


Figure 1.3: Neurosymbolic robot programming: Program synthesis and optimization algorithms operating on a neurosymbolic program representation permit to integrate AI assistance into the robot programming process, while leaving the human programmer in control of robot behavior.

combine the advantages of both subsymbolic end-to-end learning and textual programming, while avoiding as many of their shortcomings as possible. This work introduces *neurosymbolic robot programming*, a robot programming framework that leverages a neurosymbolic program representation as well as AI algorithms for program synthesis and program optimization operating on it.

1.2 Contributions

Neurosymbolic robot programming approaches the problem of creating, deploying and operating safe and capable intelligent robots from two central premises: That programming as an activity and the program as an artifact are highly useful concepts in AI creation and human-AI interaction, and that a practical resolution of the capability-control-tradeoff requires bridging the representational divide between human and machine. This work introduces neurosymbolic robot programming from a conceptual perspective, proposes a framework of algorithms and data structures to realize it in the context of robotic manipulation, and validates it on several real-world applications from industrial and service robotics. It makes the following contributions:

1. **Neurosymbolic Robot Programs (NRPs)**, a robot program representation that combines the advantages of both symbolic and subsymbolic, particu-

◁ *Representation*

larly neural, program representations. NRPs are a *dual* representation that enforces an isomorphism between a symbolic program and its subsymbolic, differentiable "shadow". They combine the compositionality and human interpretability of symbolic programs with the learnability and optimizability of neural representations: The symbolic representation affords explainability and human interaction, while its subsymbolic counterpart enables self-learning and first-order optimization.

Optimization algorithms ▷

2. **A first-order robot program optimizer** capable of optimizing NRPs with respect to a wide range of task objectives, subject to motion-level constraints such as collision-freeness. Shadow Program Inversion (SPI) (Alt et al., 2021) and its variants perform iterative gradient-based optimization over learned models of robot programs. It enables efficient optimization without requiring real-world program rollouts, as well as the interpretation of optimization results by human experts. SPI is evaluated in several real-world industrial and household use cases such as mechanical assembly, industrial quality control and table cleaning.

Program synthesis systems ▷

3. **A family of hybrid symbolic-subsymbolic robot program synthesis systems** that combine Knowledge Representation & Reasoning (KR&R) over explicit sources of knowledge with learned, neural, subsymbolic modules. MetaWizard (Alt et al., 2023) and its variants facilitate the bootstrapping of program structures to achieve tasks given high-level human specifications, and refine these program skeletons through interactive dialogue with human domain experts. The proposed systems are evaluated in the context of retail and industrial robotics on surface finishing, mechanical assembly and shopping assistance tasks.

Framework ▷

4. **A framework for AI-enabled robot programming** that combines the proposed neurosymbolic robot program representations as well as the algorithms for program synthesis and optimization operating on it, and integrates them into a programming framework that balances the need to achieve high degree of capability with the need for retaining control of robot behavior. It takes a holistic view of the process of developing, deploying and operating robots and realizes a robot programming workflow that provides AI assistance at all stages of the programming process.

1.3 Outline

The remainder of this work is organized as follows:

Chapter 2 presents NRPs, a dual representation of robot programs that bridges the gap between symbolic, explicitly represented programs and neural, subsymbolic, learned policies. It introduces the concept of *Shadow Programs* in the context of surrogate models, and presents a mathematical formulation of a robot program as a parameterized function predicting robot behavior. It then presents an architecture for robot programs as differentiable computation graphs (DCGs) and illustrates how the combination of neural networks and differentiable programming techniques can realize Shadow Programs as differentiable, predictive models of robot programs. It presents Differentiable Gaussian Process Motion Planning for N-DoF Manipulators (DGPMP2-ND), a differentiable collision-free path planner, and its integration as an algorithmic prior for bootstrapping smooth, kinematically feasible and collision-free robot motions.

Chapter 3 introduces SPI, a first-order optimizer for parameterized robot programs, which leverages NRPs to jointly optimize program parameters with respect to a wide range of task objectives, as well as variants of SPI that enable lifelong learning and joint parameter and trajectory optimization. A comprehensive evaluation of SPI and its variants on tasks ranging from industrial peg-in-hole, search and assembly tasks to household fetch-and-place tasks is presented.

Chapter 4 presents the MetaWizard family of program synthesis systems. It motivates and explores the use of both symbolic and neural methods for KR&R on human expertise and task knowledge, as well as intuitive modalities for interacting with AI-enabled program synthesis algorithms, such as via virtual reality (VR) demonstrations or natural-language dialogue. MetaWizard is evaluated in three real-world applications comprising retail fetch-and-place tasks, wind turbine refabrication and industrial peg-in-hole assembly.

Chapter 5 integrates the proposed representations and algorithms into a principled framework for AI-enabled robot programming. It proposes a conceptual framework for robot programming aimed at resolving the capability-safety-tradeoff in a way that facilitates the solution of complex applications, as well as a software framework for learning, synthesizing, optimizing and maintaining neurosymbolic robot programs.

Chapter 6 provides a summary of this work, contextualizes it within the field of AI-enabled robot programming from both applied and scientific perspectives, and proposes avenues of further research.

CHAPTER 2

A Neurosymbolic Robot Program Representation

A central thesis of this work is that explicitly represented robot programs can serve as a bridge between robot and human intelligence, and facilitate AI-enabled programming through robot learning and optimization, while leaving humans in control of robot behavior. Chapter 1 introduces robot programming as bidirectional communication between programmer and robot. In this communication, the robot program serves a dual role: To *instruct* the robot what to do and how to do it, and to *inform* the programmer what the robot is doing and how it is doing it. This bidirectional communication hinges on a shared data structure – the robot program – that describes robot behavior. This chapter motivates the use of neurosymbolic representations for robot programming and introduces NRPs, a robot program representation that combines the advantages of symbolic and subsymbolic programs (see Figure 2.1).

2.1 A Question of Representation

A core tenet of this dissertation is that the way robot programs are represented informs and constrains the ways in which robots can be programmed, and that a well-designed program representation is key to the realization of a robot programming framework that satisfies the needs of real-world robotics applications.

2.1.1 Symbolic and Subsymbolic Representations

Symbolic representations As a central claim of classical cognitive science is that the human brain is a physical symbol system, operating on symbolic representations of the perceived environment, possible actions, and knowledge about the world and itself (Fodor, 1980; Newell, 1994; Deacon, 1998). Newell (1980) defines *symbols* as entities "which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure)." Rougier (2009)

◁ *Symbols*

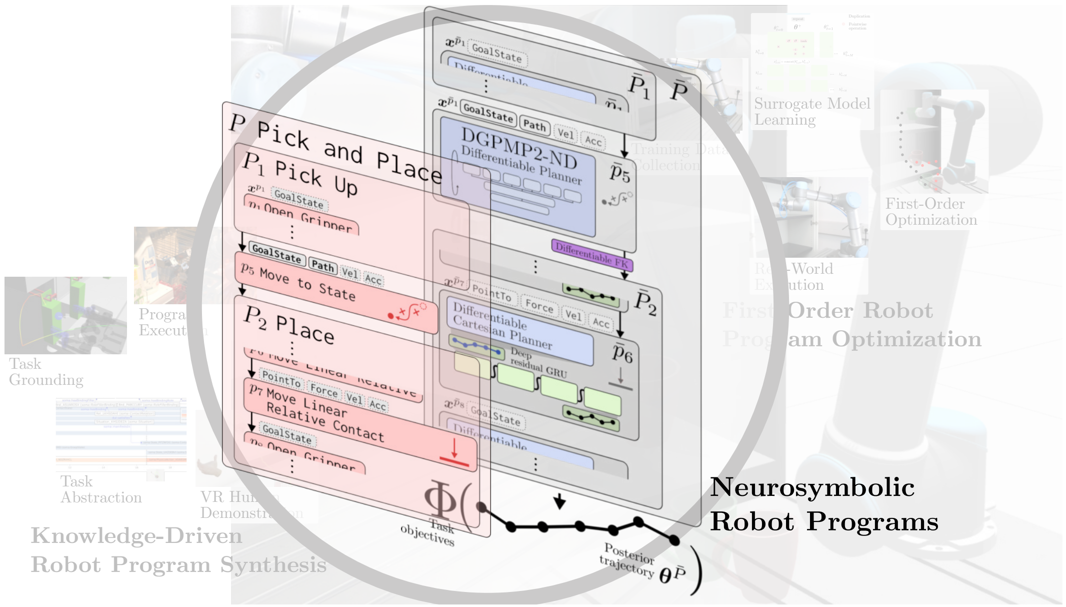


Figure 2.1: Neurosymbolic Robot Programs (NRPs) combine a symbolic, skill-based program representation (left, red) with a neurosymbolic, differentiable surrogate (right, grey). NRPs afford first-order learning and optimization while being composable and editable by human programmers.

proposes a more concrete definition: “Symbols are [...] defined as a functional regularity where a signifier stands for a signified but this function is grounded on an arbitrary conventional rule established by some entity.” Symbols are patterns, or “regularities”, that signify other entities: Things in the real world, such as cups or cats, or abstract notions, such as efficiency or safety. The relation between signifier and signified is “grounded on an arbitrary conventional rule established by some entity” (Rougier, 2009): The semantics of symbolic representations follow rules of interpretation on which all producers or consumers of a given symbolic representation must agree. The archetypical example of a symbolic representation is natural language (Chao, 1968): Words are symbols for real-world or abstract concepts. The word “table” is a symbol which our brains can ground in the abstract concept of a table, i.e. an item of furniture with a flat surface that affords the placing of objects on it, or in a particular physical entity that we perceive with our senses, such as the desk that is currently in front of me. Words are combined to symbolic structures such as sentences to represent complex concepts (Newell, 1980; Chaudhuri et al., 2021; Rosenbloom, 2023). Pitt (2022) offers a concise definition: “Symbolic structures [...] have semantically evaluable constituents.” Symbols are typically considered *atomic* in that they cannot be further subdivided; *discrete* in that they can be enumerated; *static* in that the signifier does not change, though the signified may change; and *arbitrary* in that the value of the signifier does not carry meaning in itself (Blank et al., 1992).

Symbolic structures ▷

Symbolic representations and algorithms operating on them have been the focus of AI research throughout most of the 20th century. “Classical AI”, in which learning, inference, optimization and related cognitive tasks are realized chiefly via symbol manipulation, rested on the Physical Symbol System Hypothesis (PSSH): “A physical symbol system has the necessary and sufficient means for general intelligent action” (Newell, 1980). The PSSH has been critiqued from many directions (Brooks, 1990; Russell and Norvig, 2021), and I refer to the literature for detailed accounts of its criticisms (Rosenbloom, 2023; Nilsson, 2007). Programming languages, i.e. symbolic, textual representations, have been the default choice for representing programs, both in robotics and in general computing. From an AI perspective, symbolic program representations have several advantages. High-level programming languages are syntactically sufficiently similar to natural language to facilitate the composition and editing of programs by human programmers, while simultaneously facilitating algorithmic synthesis or optimization by virtue of their rigid grammar (Summers, 1977; Alur et al., 2018; Sobania et al., 2023). The composition of symbol structures permits the efficient use of recursive divide-and-conquer algorithms to computationally manipulate programs (Smith, 1985; Alur et al., 2017) or formally verify them by theorem proving (Manna and Waldinger, 1971; D’Silva et al., 2008).

◁ *Symbolic program representations*

Subsymbolic representations As opposed to symbolic representations, which are “generally characterized by hard-coded, explicit rules operating on discrete, static tokens”, subsymbolic processing is “associated with learned, fuzzy constraints affecting continuous, distributed representations” (Blank et al., 1992). Subsymbolic representations of percepts or knowledge underpin connectionist or hybrid approaches to AI (Vernon, 2022), which emphasize that “to build a system that is intelligent it is necessary to have its representations grounded in the physical world” (Brooks, 1990). The Physical Grounding Hypothesis (PGH) proposes that “the world is its own best model” (Brooks, 1990), and that physical symbol systems will necessarily misrepresent the real world due to the approximation errors and oversimplifications introduced by atomic, discrete, static and arbitrary symbols.

Subsymbolic representations are often characterized as *distributed*, in that information is represented by the interplay between multiple signifiers, rather than individual symbols; *continuous*; and *emergent*, in that representations form over time (Blank et al., 1992). Unlike symbolic representations, in which the signifiers are arbitrary, their concrete representation of information is often a function of the use of the representation (Blank et al., 1992). Deep neural networks (DNNs) are archetypical examples for subsymbolic representations. The representation of information is distributed across the continuous-valued weights of the network, and emerges during training. The precise nature of this distributed representation reflects the training algorithm and objective function.

The division between symbolic and subsymbolic representations is not clear-cut; rather, there is a continuum between the two, and most representations lie somewhere between the extremes (Blank et al., 1992). Smolensky (1988) proposes a hierarchical model of representation, in which subsymbolic representations form the representational foundations from which symbolic representations are constructed: They are "cognitive descriptions built up of entities that correspond to *constituents* of [...] symbols." Program representations such as Dynamic Movement Primitives (DMPs) and similar modular representations are subsymbolic in the sense that robot behavior is represented as continuous-valued functions; however, at the program level, they act as symbols, affording e.g. sequential composition to form more complex program graphs.

Subsymbolic program representations avoid the need to specify a priori a set of symbols and a shared convention of interpreting them. Given a generic data structure, such as a neural network architecture or a parameterized ordinary differential equation (ODE), the representation of robot behavior manifests automatically, e.g. by running a learning algorithm on a dataset of human demonstrations. Subsymbolic program representations afford grounding of physical percepts and learning of visuomotor policies in a biologically plausible way (Brooks, 1990; Rombouts et al., 2013; Pogodin et al., 2021). However, subsymbolic program representations lack the interpretability of symbolic programs (Rudin, 2019; Chaudhuri et al., 2021) and establishing hard guarantees about the represented behavior, let alone formally verifying behavior, is challenging (Narodytska, 2018; Shi et al., 2023).

2.1.2 Explicit and Implicit Representations

When discussing AI systems and the algorithms and data structures they operate on, a distinction must be made between explicit and implicit representations (Rougier, 2009). A representation is more explicit, the more immediately readable it is (Kirsh, 1990). The amount of time (Kirsh, 1990) or computation (Kirsh, 2006) required to extract the represented information serves to quantify the explicitness of a representation. The distinction between explicit and implicit representation therefore involves the "consumer" of the representation, such as the human programmer reading or writing a robot program, or the robot executing a program. To a robot controller, assembly language is an explicit program representation: The instructions it must execute are immediately readable, and the only additional computation required is loading them into the appropriate processor registers. To a human programmer, the robot behavior represented by assembly language is less explicit, as she must consult a "mental model" of the robot controller to understand the behavioral semantics of a given piece of assembly code. To the same programmer, assembly code represents robot behavior more explicitly than a deep neural policy, whose weights alone provide next to no indication of the represented behavior.

With regard to robot programs, explicit representations have several advantages (Gray and Rumpe, 2022). First, they permit programmers to describe intended behavior in a syntactically concise way, permitting other programmers or robot end users to understand and modify robot behavior. Moreover, explicit representations more precisely encode the semantics intended by the programmer, as less latent knowledge is required by the reader to decode the information, which could possibly falsify the intended semantics. By the same reasoning, explicit program representations can be parsed by computers to help identify inconsistencies, find bugs, or even apply formal verification methods. A significant shortcoming of explicit representations is that it is challenging to find a representation that minimizes the need for latent knowledge while remaining concise (Levesque and Brachman, 1987). Moreover, implicit program representations avoid the need for humans to design good representations for programs a priori, by substituting it with suitable learning algorithms and general-purpose data structures: In neural policies, the (latent) program representation emerges by gradient descent, and the policy can be used without the user's knowledge of how motions or objects are represented.

2.1.3 Neurosymbolic Representations

Ideally, then, AI-enabled robot programming operates on a program representation that combines the properties of symbolic and subsymbolic representations, particularly of explicit symbolic representations such as textual or graphical programs, and implicit subsymbolic representations such as neural networks. Neurosymbolic representations provide such a combination. A *neurosymbolic program* is a "program that uses neural components and either symbolic components or symbolic compositions" (Chaudhuri et al., 2021). In the context of programs, a *symbolic component* is "a function that comes with a symbolic implementation, or [...] a symbolic specification of its functionality" (Chaudhuri et al., 2021): Symbolic components can be processed by symbol systems. *Composition* of multiple components, such as functions in a program, is symbolic if "certain requirements hold at the interface of the components being composed" (Chaudhuri et al., 2021). In other words, composition is performed via interfaces that satisfy hard, symbolic (e.g. logical) constraints, and that have explicit semantics.

◁ *Neurosymbolic program*

Consider a function that computes inverse kinematics (IK), i.e. takes an end-effector pose as input and returns a corresponding robot joint configuration. A symbolic IK component `f`, for example, could accept one single input parameter, `X`, that is a symbol representing a pose, and output a symbol `theta` representing the associated joint configuration. `f` is a symbol system with explicit semantics: For every pose `X`, a user will know that they will obtain the corresponding `theta`, because the behavior of `f` has been symbolically specified; a user will also be able to compose `f` with other components, as the semantics of `X` and `theta` are known. A neural IK component f, on the other hand, takes a tensor x as input and outputs a tensor θ; the semantics of x and θ are implicit in the function and arise during training. Neural networks afford symbolic composition by human programmers or symbolic program synthesis systems only if semantics are made explicit and described symbolically.

Neurosymbolic program representations permit the use of subsymbolic learning algorithms and neural representations at the component level, while still affording symbolic planning of and reasoning about programs at the composite level. Moreover, if neurosymbolic programs impose some symbolic constraints on the input-output relationship of neural components, humans can compose and parameterize such neurosymbolic components, and test their (learned) behavior against those symbolic constraints. For robot programming, a neurosymbolic program representation can facilitate learning and first-order optimization of robot behavior as well as symbolic program synthesis, while permitting human programmers to understand and modify the resulting programs. In the following sections, a neurosymbolic robot program representation for AI-enabled robot programming is proposed.

2.2 Neurosymbolic Robot Programs: Overview

The following paragraphs introduce Neurosymbolic Robot Programs (NRPs), a novel representation of robot programs that associates a symbolic, skill-based program representation with a neurosymbolic DCG to afford both symbolic planning and first-order learning and optimization. NRPs have been the subject of several publications (Alt et al., 2021; Alt et al., 2022b; Alt et al., 2025). The following sections introduce NRPs in greater detail. NRPs form the basis for the program optimization and synthesis algorithms presented in Chapters 3 and 4.

2.2.1 Desiderata for a Neurosymbolic Program Representation

The vision of AI-enabled robot programming outlined in Chapter 1 – enabling humans to endow robots with the abilities to solve complex manipulation tasks in challenging environments – imposes several desiderata on the program representation used to realize it. Some requirements arise directly from the objective of enabling the creation, deployment and operation of intelligent robots, while others facilitate the realization of the programming framework in practical applications.

Affordance of learning The ability to adapt their behavior to novel tasks or environments based on experience is a core property of intelligent systems (Russell and Norvig, 2021). Many of aspects of robot programming benefit from learning, the adaptation of robot programs to incremental changes in the environment, or the optimization of skills based on human demonstrations. Pre-programming all possible mappings from e.g. sensory inputs to robot behavior for all possible goals is intractable. Consequently, a program representation for AI-enabled robot programming must afford learning of programs, or parts of programs, from data reflecting real-world interaction between the robot and its environment.

Affordance of planning and optimization at runtime The deployment of intelligent robots in real-world environments requires robots to be able to adapt to novel situations as they arise. A robotic retail assistant, for example, may have to manipulate products that have fallen over or have been placed by customers in locations in which they do not belong. Other use cases require robots to adapt to changing task requirements. In low-volume, high-mix production scenarios, industrial robots often have to manipulate new parts or part variants on the fly, without explicit prior programming for these exact parts. A general-purpose neurosymbolic robot program representation must permit task or motion planners to modify the robot's behavior both offline (e.g. for task-level planning of manipulation strategies) or online (e.g. for planning collision-free motions). Moreover, a general-purpose neurosymbolic program representation must support planning and optimization jointly at the motion and task level: Task-level objectives such as task success may impose constraints on a motion planner, such as keeping a grasped object upright during manipulation. For all applications, the robot program must produce kinematically feasible motions; for most applications, motions must be collision-free and smooth. These motion-level constraints in turn influence a task-level planner, as some motion types may not be possible under the given constraints.

Generalization across tasks Robots are universal manipulators: Kinematic configurations with many degrees of freedom allow robots to solve a wide array of tasks without requiring mechanical reconfiguration. To leverage the flexibility promised by robots, a general-purpose robot program representation must be able to represent a wide variety of tasks, if not, in principle, all possible tasks. In industrial applications, this is a prerequisite for efficient high-mix, low-volume production, where robots must switch between tasks or task variants with only minimal hardware and software reconfiguration. Service domains such as household or retail assistance require robots to be able to perform a variety of tasks, depending on user input and the state of the environment. As general-purpose programming languages have been instrumental in facilitating the widespread adoption of digital computers (Nofre et al., 2014), general-purpose robot program representations can facilitate the widespread adoption of robots as universal manipulators.

Generalization across hardware The conceptual separation of computer hardware and the software running on it has been a central paradigm of modern computing (Dijkstra, 1971). Modern programming languages encapsulate hardware behind layers of abstraction, allowing for the sharing of programs across platforms. Robot hardware extends from the computational hardware to the kinematics and dynamics of the manipulator, the end effector as well as peripheral hardware such as vision or force sensors. Modern industrial robot workcells include complex combinations of sensors and actuators, combining robots from different manufacturers and kinematic configurations. The ability to program complex robot cells using the same program representation not only simplifies the process of developing new robot workcells, but also facilitates maintenance, as replacing or adding hardware components requires only minimal changes to the existing robot software.

Human explainability The explainability of robot programs has been shown to be a major bottleneck in the adoption of AI-enabled robot programming (Peres et al., 2020; Agostinho et al., 2023; Theis et al., 2023), and is crucial for AI adoption in industrial robotics, where robot programs must undergo auditing and certification before being cleared for deployment (Tong and Lei, 2017). While it would be desirable to be guarantee explainability for both the program (the rules governing the robot's behavior) and the metaprogram (the planners and learning algorithms which generate the program), this is considered intractable for complex programs and metaprograms with a high degree of generalization (see Yampolskiy (2022) and Section 1.1.3). This work focuses on explainability of the program: An explainable program, that can be audited and certified, can be used in real-world applications, but incurs the cost of re-auditing when the policy is changed by e.g. a planner.

Human editability In both service and industrial applications, it is crucial for humans to exert influence over the robot's behavior and to change the robot program. The degree to which human users can influence the behavior of an AI system has been shown to be positively correlated with user satisfaction and positive attitudes toward AI (Morrison et al., 2023). Beyond psychological factors, human editability is a practical necessity in industrial applications, where the modification of robot programs by workers or engineers is a crucial part of robot deployment and maintenance (Alt et al., 2024a).

Symbolic composition Most general-purpose textual or graphical program representations afford symbolic composition by encapsulating complex algorithms or data structures behind symbols, which can be reused and recombined. Symbolic composition has been one of the core factors enabling humans to design, deploy and maintain complex software systems (Perlis, 1996). Likewise, there is mounting neurobiological evidence that the presence of symbolic representations in the human brain and the evolution of cognitive processes which involve symbolic compositions are central components of human intelligence (Pulvermüller et al., 2014; Do and Hasselmo, 2021).

Hierarchical structure The decomposition of high-level tasks into hierarchies of subtasks is one of the core principles of task-level programming. It permits planners to efficiently solve complex, high-level tasks by hierarchically solving simpler, more circumscribed low-level subtasks. Beyond automatic planning, hierarchical task models allow human programmers to bootstrap complex robot behavior from simple, primitive skills (Kortenkamp et al., 2016).

Interoperability Particularly in industrial applications, robots rarely operate in isolation. Typical industrial robot workcells are heterogeneous both with respect to hardware and software, and can be operational for several years. This translates to a high need for interoperability with the different program representations typically deployed on industrial robot workcells, such as manufacturer-specific robot programming langugages. A program representation which augments, rather than replaces, legacy robot software ecosystems can be retrofitted to existing robot workcells without requiring complete reprogramming, decreasing deployment costs and overhead considerably.

Neither purely symbolic nor purely subsymbolic program representations satisfy these requirements, leading to limited applicability for a range of real-world applications. Purely symbolic, explicitly represented robot programs such as textual or task model-based programs only afford learning to a limited degree, as the learning capacity of compact symbol systems is constrained; they only afford a limited degree of optimization at runtime, as they cannot represent the continuous, nonlinear interactions of the robot with a changing environment; and they are typically limited to a given robot manufacturer. Under the deep end-to-end learning paradigm, neural, subsymbolic policies implicitly represent robot behavior, which is typically neither human-interpretable nor human-editable, and which does not afford symbolic composition by default. A suitable program representation combines the benefits of both representational extremes while avoiding their respective shortcomings. The following paragraphs introduce such a representation.

2.2.2 A Dual Program Representation with a Differentiable Surrogate

The field of explainable artificial intelligence (XAI) has introduced various approaches to address the gap between the implicit, subsymbolic representations used by many AI models for learning and representations that enable formal analysis or human interpretation (see Section 2.6). One such approach is the use of *surrogate models*, which construct an interpretable surrogate of an opaque DL model that performs identically to the original, but is mechanistically interpretable (Burkart and Huber, 2021). This allows for the use of the original model for efficient inference, while the surrogate model can be used to obtain explanations when needed, while typically requiring more computational resources. NRPs apply the concept of surrogate models to robot programming by associating a symbolic *source program* with a differentiable, partially neural *shadow program*, and enforcing an isomorphism between them. NRPs are a *dual program representation*: They combine two representations, each of which can be dynamically converted into the other, and permit algorithms for program synthesis or optimization to operate on either, depending on the requirements of the algorithm. The use of surrogate

◁ *Surrogate model*

◁ *Dual representation*

models overcomes the the apparent dichotomy between self-learning and human sovereignty via an additional layer of indirection. By performing learning and optimization not on the robot program itself, but on its differentiable, subsymbolic shadow, powerful deep learning and gradient-based optimization methods can be used without requiring the user to write differentiable robot code or interpret the parameters of neural networks. Instead, the user can program a robot in a program representation familiar and understandable to them – and the optimized robot behavior is, in turn, displayed to the user in that same representation.

2.2.2.1 Processes, Programs, Skills and Tasks

For the purpose of formally defining NRPs, the terms “program”, “process”, “task”, “skill” and “parameter” must be formally defined. This work adopts the ISO/CEN 19439 definition of a ***process*** as a “partially ordered set of activities that can be executed to achieve some desired end-result in pursuit of a given objective” (Cutting-Decelle et al., 2007). ***Programs*** represent processes by *modeling* them, with the threefold purpose of instructing a machine to perform them (as a *prescriptive* model), communicate them to the programmer and other humans (*descriptive* model), and making predictive statements about them (*predictive* model). The predictive view on programs permits to establish a mathematical definition of programs as functions, which map a set of program parameters and the initial state of the robot and its environment to a robot motion and the corresponding outcomes in the world.

Process ▷

Program ▷

Skill ▷

A ***skill*** is a “predefined robot’s capability that can be parameterized to solve a specific goal” (Pantano et al., 2022). A majority of common robot program representations are skill-based, such as Movement Primitives (MPs) (Schaal, 2006; Paraschos et al., 2013; Ratliff et al., 2018; Seker et al., 2019), Task and Motion Planning (TAMP) operators (Kaelbling and Lozano-Perez, 2011; Silver et al., 2021) or Behavior Trees (BTs) (Colledanchise and Ögren, 2018). The definition of a skill proposed in (Pantano et al., 2022) also covers other representations typically not considered “skill-based”, such as most modular robot programming languages (Ajaykumar et al., 2021), the predefined function libraries used by many program synthesis approaches (Fan et al., 2024; Liang et al., 2023; Vemprala et al., 2023) and even parameterized neural policies (Klimek et al., 2017; Verma et al., 2024). A skill refers to any parameterized function that generates some goal-oriented robot behavior. For the purpose of this work, a Conditional Neural Movement Primitive (CNMP) (Seker et al., 2019) trained to represent a wiping action, a URScript (Universal Robots, 2018b) function to execute a spiral search motion, or a TAMP operator for closing a robotic gripper can be considered skills.

Task ▷

A ***task*** is “an ordered ensemble of [skills] and depicts a concrete representation of steps in a workflow to solve a specific goal” (Pantano et al., 2022). Complex tasks typically involve sequential chaining of several skills. Real-world tasks are inherently hierarchical: A pick-and-place task, for example, can be decomposed into the subtasks *Pick* and *Place*; *Pick*, in turn, can be decomposed into *Approach*, *Grasp*

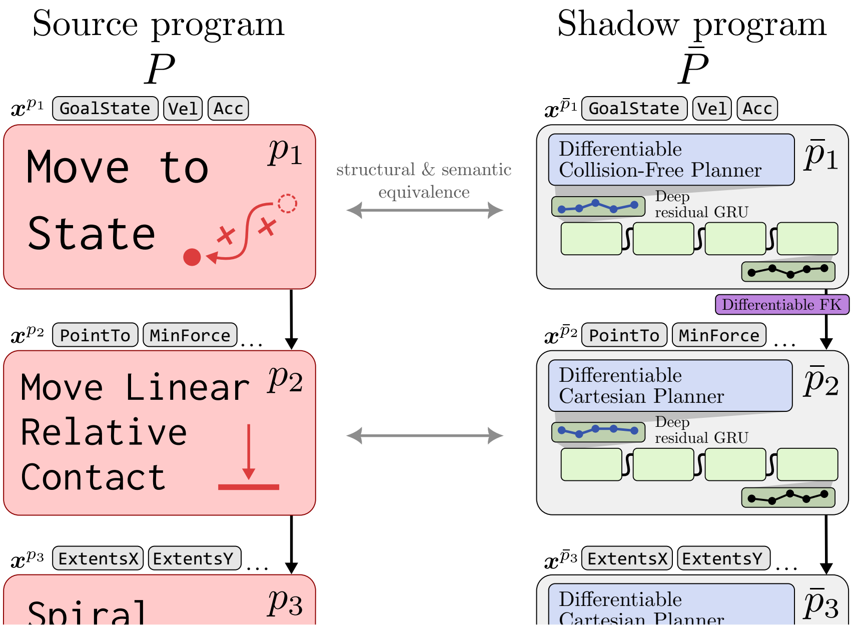


Figure 2.2: NRPs are a dual program representation, associating a skill-based robot program (*source program*) P with a corresponding DCG $\bar{P}$ (*shadow program*).

and *Depart*; each of which can be solved by a sequence of motion and perception skills. Many skill-based representations such as textual programming languages or graphical task models (Jäkel, 2013) afford hierarchical task composition. From a programmer's perspective, a hierarchical conception of tasks is an instance of symbolic composition, which ensures modularity and eases understanding and maintenance of complex robot programs (Chaudhuri et al., 2021); from a cognitivist perspective, a hierarchical task definition mirrors the hierarchical organization of sensorimotor control circuits in the human brain (Newell, 1994; Flanagan et al., 2006).

◁ *Parameter*

Parameters "configure a [s]kill for a specific [t]ask" (Pantano et al., 2022). In the example of a pick-and-place task, an *Open Gripper* skill may be parameterized with the width of the gripper opening for a parallel gripper, or a set of joint angles for a humanoid hand. The parameterization of skills depends on the skill representation, the semantics of the skill and the robot hardware: The same spiral motion can be generated by a DMP parameterized with a set of basis function weights (Pervez and Lee, 2018), or e.g. a URScript function parameterized with the spiral extents and distance between the spiral arms (Alt et al., 2021).

2.2.2.2 A Dual Program Representation

◁ *Robot program*

In the context of this work, then, a ***robot program*** denotes a *model of a task*, hierarchically and sequentially composed of subtasks, which are ultimately composed of sequentially chained, parameterized skills. This definition covers most parameterized, modular program representations currently in use. NRPs are robot programs — they are hierarchical models of tasks, which are in turn composed of parameterized skills. The core representational difference between NRPs and other program representations is that NRPs have a dual representation, offering two different views onto the same robot program. The NRP *source program* and *shadow program* are, in fact, the same program – they are semantically equivalent, but represented differently. Programmers or algorithms working with NRPs can choose whether to use the source or shadow representation, and seamlessly convert between the two.

Source program ▷

The ***source program representation*** can be any hierarchical task model, composed of parameterized skills. The term "source program" reflects the fact that it can be any existing program representation – and that the corresponding shadow program is constructed from it. A robot programmer using NRPs can choose any source program representation they want, provided it is a hierarchical task model, composed of parameterized skills. Chapters 3 and 4 provide examples for different source program representations, including DMPs, Cognitive Robot Abstract Machine (CRAM) plans, URScript, or task models in a graphical robot programming language. In terms of the ternary conception of programs as descriptive, prescriptive and predictive models, the source program primarily acts as a descriptive and prescriptive model. It is typically interpretable and human-editable, serving as a method for human programmers to describe to the robot the task it should solve, and for the robot to communicate to the human programmer how it is solving the task.

Shadow program ▷

The ***shadow program representation***, in turn, is a novel, differentiable program representation, presented in detail in Section 2.4. Shadow programs, too, are hierarchical task models, composed of parameterized skills. Unlike the source program representation, the shadow program representation acts primarily as a predictive model. It is differentiable and partially neural, making it ideal for learning and optimization. The unique benefits of NRPs stem from the fact that the descriptive, prescriptive and predictive roles do not have to be played by the one and the same internal representation. The source program representation can be descriptive and prescriptive, without having to be suitable for optimization and learning; the shadow program representation can be differentiable and subsymbolic, without having to be interpretable (descriptive) or executable (prescriptive). The dual representation used by NRPs sidesteps the capability-control-tradeoff by allowing users to switch seamlessly between source and shadow program representations, depending on the user's needs. This duality rests on two properties, *structural* and *semantic* equivalence, which are enforced by design.

Structural equivalence

A shadow program $\bar{P}$ is ***structurally equivalent*** to its corresponding source program P if for each skill p_i in P, there is a corresponding *shadow skill* $\bar{p}_i$ in $\bar{P}$, and if for each parameter x_{p_i} of p_i, $\bar{p}_i$ has a corresponding parameter $\bar{x}_{\bar{p}_i}$.

These correspondences are the only constraints on either representation – the internal representations used to implement skills or parameters, the computational mechanisms by which skills are combined, the methods used to implement control flow etc. can vary substantially between source and shadow programs. The following sections as well as Chapters 3 and 4 will show that the three listed correspondences suffice to automatically construct a shadow program for a given source program, apply learning and optimization algorithms to the shadow program, and map the results back to the source program so that human programmers can interpret and possibly modify them.

Semantic equivalence

$\bar{P}$ and P are ***semantically equivalent*** if they are models for the same task. Given the same parameters, the robot actions and their consequences on the world predicted by $\bar{P}$ are identical to, or closely approximate, the robot actions and actual real-world consequences resulting from execution of P.

Semantic equivalence permits shadow programs to act as surrogates, facilitating computations such as data-driven learning or first-order optimization that the source program does not afford.

The structural and semantic equivalence properties make NRPs a *dual* program representation by offering two different "views" on the same program: Users or algorithms can interact with the program in its source or shadow representation, depending on whether interpretability and human-interpretability or learning and subsymbolic optimization are required. NRPs as a dual program representation are illustrated in Figure 2.2.

2.3 Source Program Representations

The following paragraphs provide a formal definition of robot programs as predictive models of tasks. It delimits the class of program representations that can represent source programs in the context of NRPs.

2.3.1 A Formal Definition of Robot Skills

A proper definition of NRPs requires a more concrete definition of skills than parameterized, atomic robot actions (Pantano et al., 2022). Chapter 3 proposes a learning and optimization algorithm for skills, which requires the definition of a skill as a parameterized function producing a robot motion that, in turn, influences the environment. Particularly in dynamic environments, some aspects of the environment will be different at every skill execution: In a retail setting, the same shelf may contain different products on different days, e.g. due to promotions; in industrial applications, the pose of a feature on a workpiece, such as connector pins or screw holes, may stochastically vary between individual workpieces due to manufacturing tolerances. For these reasons, this work conceives of a skill as a *parameterized stochastic process*: Given a parameter vector $\boldsymbol{x} \in \mathbb{R}^N$ and initial robot and world state θ_0, a skill p is a stochastic process which, when sampled, gives rise to a trajectory $\boldsymbol{\theta}$. The notions of “parameter vector”, “trajectory” and “executing a skill” merit further definition.

Parameter vector ▷

The length N and contents of the ***parameter vector*** $\boldsymbol{x}$ are skill dependent. A DMP for table wiping, for example, may have a parameter vector of length 12, containing one real-valued weight for each of 12 Gaussian kernel functions (Ijspeert et al., 2002). An force-controlled probe search skill from an industrial robot skill library may have a parameter vector of length 32, corresponding to 16 2D positions in a plane describing the search pattern (Alt et al., 2021). Aside from the domain of $\mathbb{R}$, no additional requirements are placed on the input vector, allowing NRPs to represent a wide variety of source programs.

Trajectory

A ***trajectory*** $\boldsymbol{\theta} \in \mathcal{S}^M$ is a sequence of states $\theta \in \mathcal{S}$, with a fixed temporal sampling interval between successive states. The trajectory length M is not assumed to be fixed, but may vary, even between executions of the same skill.

The trajectories generated by search skills, for example, often exhibit widely varying lengths, as the feature the robot is searching for may be found immediately in some executions, but very late or not at all in others.

State space ▷

Like the contents of the parameter vector $\boldsymbol{x}$, the state space $\mathcal{S}$ is, in principle, skill-, hardware- and application-dependent and represents of the robot configuration and world state at the moment the skill is executed. $\mathcal{S}$ will typically be a subspace of $\mathbb{R}^N$. For simple skills defined in joint space, $\mathcal{S} = \mathcal{C}$, the configuration space of the robot; for simple skills defined in Cartesian space, $\mathcal{S}$ is the robot’s workspace. More complex skill representations or applications may require additional information to represent the initial state, such as force or torque values at the end effector and the current task frame for force-controlled skills (De Schutter et al., 2007), or a voxelized representation of the environment (Alt et al., 2025). For the purpose of defining robot programs as predictive models of tasks, it suffices to assume *some* representation of the current state of the robot (and, possibly, its environment) at the beginning of a skill. Section 2.4 will illustrate some technical constraints on possible state representations.

D **Skill (formal definition)**

A skill can be modeled as a discrete-time continuous-valued stochastic process: $p := \{\Theta_t\}$ with time index $t \in \{1, 2, ..., M_p\}$ and state space $\mathcal{S}$. The law of $\{\Theta_t\}$ is parameterized by $\boldsymbol{x}^p$, the parameter vector for p, and conditional on θ_0^p as well as the current environment H: The distribution of trajectories is a partial function of the skill's parameters and the current state of the robot, as well as those task-relevant aspects of the environment which are not captured by θ_0^p. Following Alt et al. (2022b), those aspects of the environment are grouped under a random variable H.

Spiral search is an intuitive example: The skill parameters define the shape of the resulting spiral trajectories, and the current robot state defines the absolute pose of the spiral in the workspace. $\{\Theta_t\}$ is, however, not exclusively a function of the skill parameters and start state, but also depends on the environment, the dynamics of the robot and other factors. For a force-controlled spiral search skill, the resulting trajectories will deviate considerably from the the pre-planned planar spiral; if a hole is found, the trajectory will dip into it, and the remaining motion will be constrained by the size of the hole.

◁ *Skill execution*

Executing or *evaluating* a skill p results in some robot behavior, which is dependent on the parameterization of the skill and the state of the robot and environment. In the proposed model, this corresponds to sampling from the stochastic process, resulting in a trajectory $\boldsymbol{\theta}^p$. Due to the possibly dynamic nature of the environment, successive evaluations of the same skill will produce different trajectories, even for the same parameterization $\boldsymbol{x}^p$. As the true distribution of $\{\Theta_t\}$ is generally unknown, it is impossible to evaluate p in a computer – the execution of a skill requires actual interaction of a robot in an environment, unless both the robot and environment are simulated.

This mathematical conception of a skill is sufficiently flexible to cover any parameterized mechanism engendering robot behavior, including both deterministic, pre-planned motions and non-deterministic behavior such as force control or neural policies. The concrete representation of the skill, the robot hardware and the environment are abstracted away and folded into the stochastic process $\{\Theta_t\}$. While it may be tempting to further specify the exact contents of $\boldsymbol{x}^p$ and $\boldsymbol{\theta}^p$ for additional mathematical rigor, such specification is neither required nor possible. The provided notation suffices to define a skill p as a mapping from real-world parameters to a distribution of trajectories. The conception of skills as parameterized stochastic processes is a mathematical abstraction that allows to concisely describe NRP source programs: ***Any skill-based robot program representation whose skills can be interpreted as parameterized discrete-time continuous-valued stochastic processes can serve as the source program representation of NRPs.*** It allows for source programs as diverse as textual URScript programs, DMPs or Hierarchical Task Networks (HTNs). The fact that p can generally not be evaluated in a computer, let alone differentiated, motivates Neurosymbolic Robot Program (NRP) as a dual representation, as the shadow representation facilitates such operations.

2.3.1.1 Examples

URScript functions The URScript programming language is the platform-specific programming language of the robot manufacturer Universal Robots (Universal Robots, 2018b). URScript has a syntax similar to Python, offers general-purpose computing mechanisms such as arithmetic operators, string manipulation, threading and network communication, but also an application programming interface (API) containing an array of motion primitives. One example is the function `movel(pose, a, v, t, r)`, which takes a target `pose`, tool acceleration `a`, tool speed `v`, optional movement time `t`, and optional blending radius `r`, and executes a linear end-effector motion in the workspace. `movel` fits the definition of a skill: It is parameterized, with a parameter vector $\boldsymbol{x}^p$ comprising a real-valued representation of the parameters `pose`, `a`, `v`, `t` and `r`; when executed, the robot follows a trajectory $\boldsymbol{\theta}^p$. While `movel` can be considered deterministic from a control perspective – it issues the same control commands when called multiple times with the same parameters – $\boldsymbol{\theta}^p$ will be subject to small deviations at every execution due to physical phenomena such as wear and tear of the robot's joints or slight changes in the lengths of arm segments due to temperature (Raible et al., 2023b). In more extreme circumstances, larger deviations from the expected trajectory are possible, e.g. due to collisions with the environment.

DMPs DMPs are nonlinear dynamical systems, which are parameterized to generate goal-directed or periodic robot behavior. Consider the DMP formulation proposed by Ijspeert et al. (2013). It proposes a damped spring model, which can be parameterized to act as a single point attractor or a limit cycle attractor, depending on the choice of several constants and a *forcing function* f. Ijspeert et al. (2013) realize f as a normalized linear combination of exponential basis functions with weights w_i: The weights w_i are parameters determining the behavior of the system. Viewed as a black box, DMPs can be modeled as a parameterized, discrete-time, continuous-valued stochastic process: They generate a robot trajectory which is contingent on a set of parameters. The mathematical formulation of DMPs is deterministic, suggesting a stochastic process with zero variance. As their execution on real robot hardware is subject to the same stochastic variation as e.g. the `movel` URScript function, however, a probabilistic model is still warranted.

ARTM skills The ArtiMinds Robot Task Model (ARTM) is a commercial, industrial robot program representation which underlies a manufacturer-independent, graphical programming framework (Schmidt-Rohr et al., 2013). The ArtiMinds Robot Task Model (ARTM) is an extension of Generalized Manipulation Strategies (Jäkel, 2013). An ARTM skill is a parameterized set of constraint functions, which, when used with a constraint-based motion planner, yield a planned *prior trajectory* $\tilde{\boldsymbol{\theta}}^p$. On the basis of this prior trajectory, along with the skill type and skill parameters, a set of platform-specific compiler backends generate code for the desired robot platform, such as a Fanuc industrial manipulator, and peripheral devices, such as grippers or force-torque sensors. `Insert Moment`, an exemplary ARTM skill for force-controlled, zero-torque peg-in-hole insertion, accepts 8 parameters: `Tool`, `Bias`, `PointFrom`, `PointTo`, `WreExt`, `GoalWreExt`, `WreCenter`, `GoalWreCenter`, as well as a velocity `Vel` and acceleration `Acc`. The poses `PointFrom` and `PointTo` define a relative motion in the `Tool` coordinate frame; `WreCenter` (wrench center) and `WreExt` (wrench extents) define the runtime constraints of the skill, i.e. the end-effector wrench region the force controller will attempt to reach during the execution of the skill; and `GoalWreCenter` and `GoalWreExt` parameterize the skill's stop condition, i.e. an end-effector wrench region in which the skill will stop. Like URScript functions and DMPs, ARTM skills can be modeled as parameterized stochastic processes. While the prior trajectories $\tilde{\boldsymbol{\theta}}^p$ will be the same for identical parameterizations, the real-world robot trajectories are subject to variation. The path followed by the robot will depend on the shapes of the peg and hole, and the forces experienced during the motion are contingent on material properties such as surface friction or the rigidity of the materials.

2.3.2 Skill-based Robot Programs

A robot program is a model of a task, sequentially and hierarchically composed of subtasks, which are, in turn, composed of skills (Pantano et al., 2022). As for robot skills themselves, skill-based robot programs are represented in a variety of ways. Again, a common abstraction serves to identify a class of program representations which can serve as source program representations for NRPs.

D **Robot program**

For the purpose of this work, a **robot program** is a sequence of subprograms; a **subprogram** is a sequence of subprograms or skills. A robot program is denoted as P, with parameter vector $\boldsymbol{x}^P$. The ith subprogram in a program is denoted as P_i, with parameter vector $\boldsymbol{x}^{P_i}$. The jth skill in a subprogram is denoted as p_j, with parameter vector $\boldsymbol{x}^{p_j}$.

An example for a robot program solving a peg-in-hole task is shown in Figure 2.4. A program is a model of a task; like the corresponding task tree (Figure 2.3), its root represents the program as a whole. It is composed of two subprograms, `Search` and `Insert`, that solve the subtasks of searching for a hole on a workpiece, and inserting a peg into this hole. `Search`, in turn, is composed of three skills: `Approach`, `Move to Contact`, and `Spiral Search`. Robot programs are hierarchies of subprograms, nested to arbitrary depth; the leaf nodes are primitive skills.

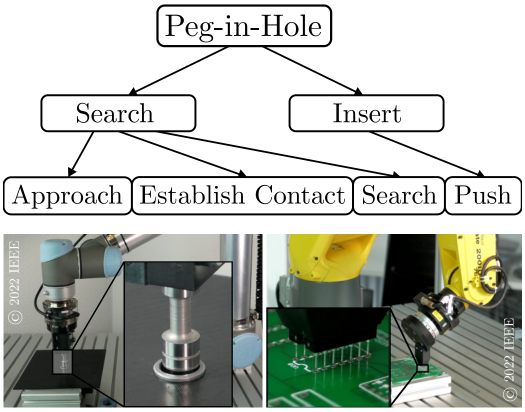


Figure 2.3: Task tree of a peg-in-hole task and two industrial peg-in-hole applications: Pneumatic valve assembly (left), PCB assembly (right) (Alt et al., 2022b). Tasks are hierarchically and sequentially decomposable into subtasks and skills.

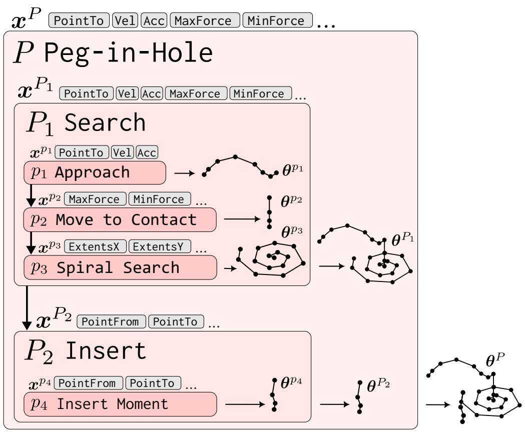

x^P PointTo Vel Acc MaxForce MinForce ...
P Peg-in-Hole
x^{P_1} PointTo Vel Acc MaxForce MinForce ...
P_1 Search
x^{p_1} PointTo Vel Acc
p_1 Approach
θ^{p_1}
x^{p_2} MaxForce MinForce ...
p_2 Move to Contact
θ^{p_2}
θ^{p_3}
x^{p_3} ExtentsX ExtentsY ...
p_3 Spiral Search
θ^{P_1}
x^{P_2} PointFrom PointTo ...
P_2 Insert
x^{p_4} PointFrom PointTo ...
p_4 Insert Moment
θ^{p_4}
θ^{P_2}
θ^P

Figure 2.4: Structure of a robot source program for a peg-in-hole task, hierarchically and sequentially composed of parameterized subprograms and skills.

Program parameters

The **program parameters** $\boldsymbol{x}^P$ are the concatenated parameter vectors of its constituent subprograms; the parameter vector of a subprogram, in turn, is the concatenated parameter vectors of its constituent subprograms and skills.

Program execution ▷

The *execution* of a robot program proceeds from top to bottom (see Figure 2.4). When a robot program is executed, its subprograms are executed in order. When a subprogram is executed, its skills are executed in order. Each skill p_j yields a trajectory $\boldsymbol{\theta}^{p_j}$; each subprogram P_i yields a trajectory $\boldsymbol{\theta}^{P_i}$, which is the concatenation of the trajectories of its constituent skills; the program P yields trajectory $\boldsymbol{\theta}^P$, the concatenation of the trajectories of its constituent subprograms.

As robot programs are typically executed in environments under uncertainty, the probabilistic definition of a robot skill as a trajectory-generating stochastic process can be extended to robot programs.

Robot program (probabilistic definition)

From a probabilistic perspective, a robot program P can be defined as a discrete-time continuous-valued stochastic process $P := \{\Theta_t\}$ with time index $t \in \{1, 2, ..., M_P\}$ and state space $\mathcal{S}$. The law of $\{\Theta_t\}$ is parameterized by the program parameters $\boldsymbol{x}^P$ and conditional on the initial state θ_0^P and environment H.

2.3.2.1 Examples

URScript programs As URScript functions are robot skills, there is a large class of URScript programs that fit the proposed program definition and can be represented as NRPs. Most industrial robot programs are sequences of function calls, with functions in turn composed of skills such as `movel`. PolyScope, Universal Robots' proprietary graphical programming interface, provides an interface by which program hierarchies can be composed, which results in structures similar to the program shown in Figure 2.4. This visual representation is internally mapped to textual URScript code.

Behavior trees A large class of BT-based robot programs can be represented as NRPs. BTs are a trees with parameterized skills at the leaves, which engender robot behavior (Colledanchise and Ögren, 2018). BTs are hierarchical and can be composed of sub-trees nested to arbitrary depth. BTs composed of Action and Sequence nodes can be trivially represented as NRPs. Fallback nodes can be represented by the same mechanism as `for` or `while` loops in URScript (see Section 3.2.3 of Chapter 3).

ARTM programs ARTM programs are hierarchically composable graphs of robot skills. ARTMs are directed acyclic graphs (DAGs) of subprograms (`Hierarchies`) or skills (`Templates`); repeating behavior is realized via a dedicated `Loop` hierarchy type. ARTMs are structurally equivalent to the program structure described in Section 2.3.2 and illustrated in Figure 2.4.

2.3.3 Discussion

The definition of NRPs' source program representation is purposefully loose. It fits a large class of skill-based robot program representations, including textual and graphical representations. In the NRP dual representation, the shadow program representation is automatically constructed from the source program; because many skill-based robot program representations can serve as source programs, NRPs can represent a wide variety of program representations currently used by robot programmers. As most hierarchically and sequentially composed sequences of parameterized skills can serve as a source program, NRPs support the "retrofitting" of existing robot programs with the data-driven optimization capabilities of NRPs. However, the structural and semantic equivalence between source and shadow programs effectively limits the expressivity of NRPs to that of the source program representation, as the source program is executed on the robot, and the shadow program is trained to be a predictive model of the resulting behavior. In practice, this restriction can be mitigated by the choice of a sufficiently flexible source program representation. The ARTM, for example, has general-purpose motion skills such as `Move to State` or `Path Force`, which execute collision-free planned motions or force-controlled freeform motions, respectively, and are parameterized with low-level sampled paths. In Section 3.3.2, NRPs are used in the AI-based planning and optimization of such freeform constrained motion skills. Similarly, URScript, other textual program representations or MPs support near-arbitrary motions, trading off intuitive parameterization for flexibility. First-order optimization over shadow programs (see Chapter 3) automatically generates such motion-level parameterizations to achieve task-level objectives.

2.4 Differentiable Shadow Programs

Section 2.3 established that most hierarchically and sequentially composed sequences of parameterized skills can serve as source programs for NRP. While these representations are generally interpretable and designed for human-machine interaction, they do not all afford learning or optimization. As noted in Section 2.3.1, real-world robot skills cannot be evaluated in a computer; while the program is a data structure represented in a computer, its execution involves interaction with the physical world outside of the machine. Any learning or optimization of the source program itself must be zero-order and model-free; the notion of "differentiating a robot program" for first-order optimization is ill-defined if evaluating that program requires running it on real robot hardware. The unique property of the NRP representation is that it enables first-order, model-based optimization of robot programs, using a *learned model of the program*.

Section 2.2 introduced NRPs as a dual program representation, which represents a robot program both in its original form (the *source program* representation) as well as in a differentiable *shadow program* representation which can serve as a model of the program. The shadow program representation facilitates learning, evaluating and optimizing programs without requiring physical robot hardware.

Shadow program ▷

The shadow program must ensure structural and semantic equivalence to the source program (see Section 2.2.2.1). Like the source program representation, it is modular and skill-based, and permits the sequential and hierarchical composition of complex programs from primitive skills. Structural equivalence is ensured by affording the same symbolic composition mechanisms as the source program representation; semantic equivalence is ensured by introducing learnable components, which can be trained to be models of the source program.

2.4.1 Shadow Skills

Shadow skill ▷

NRPs associate a *shadow skill* to each skill in the source program. For a given skill p, the corresponding shadow skill $\bar{p}$ associates a parameter vector $\boldsymbol{x}^{\bar{p}}$ and initial world state θ_0 with a posterior trajectory $\boldsymbol{\theta}^{\bar{p}}$. There is a one-to-one mapping between source and shadow skills: For every source skill p_i, there is a corresponding shadow skill $\bar{p}_i$ (see Figure 2.10).

The raison d'être for shadow skills is to serve as differentiable surrogates for robot skills for program learning and optimization. For this reason, they must be implemented purely in software – it is impossible to differentiate a physical robot and its environment. Unlike p, which is *executed*, $\bar{p}$ is a function which is *evaluated*. Where we only have indirect means to obtain the real-world trajectory $\boldsymbol{\theta}^p$ by measurement, $\bar{p}$ *returns* $\boldsymbol{\theta}^{\bar{p}}$. This enables computation with $\boldsymbol{\theta}^{\bar{p}}$ and the skill $\bar{p}$ itself; the shadow program representation represents the abstract notion of a robot skill as a DCG that can be differentiated using automatic differentiation, allowing for implementation of otherwise ill-defined operations such as "inversion of a skill" or "backpropagation through a skill".

2.4.1.1 Tensor Representation of States and Trajectories

The representation of a robot skill as a DCG necessitates a representation of parameters, states and trajectories as tensors over which differentiable operations can be defined. This section introduces a tensor representation of parameters, states and trajectories which is used for the remainder of this work. A shadow skill's parameter tensor $\boldsymbol{x}^{\bar{p}}$ is simply the corresponding skill's parameter vector $\boldsymbol{x}^p$. The representation of states and trajectories requires additional care.

◁ *State space*

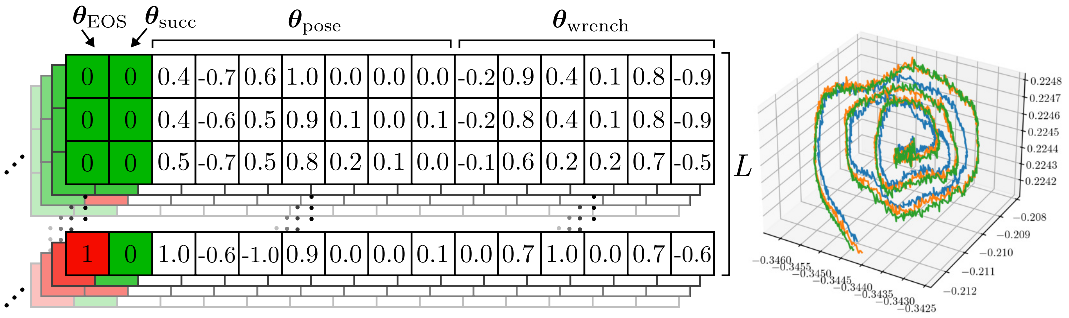
As motivated in Section 2.3.1, the state space $\mathcal{S}$ is skill-, application- and hardware dependent. In this work, three spaces are considered:

1. **Cartesian end-effector poses $\mathcal{P}$:** The Cartesian position and orientation of the robot's tool center point (TCP) in the workspace. Represented as the concatenation of a 3D position vector and a 4D normalized unit quaternion. Other pose representations such as dual quaternions would also be possible, provided they are unambiguous and interpolate smoothly.

2. **Configuration space $\mathcal{C}$:** The N-dimensional joint configuration of the robot, including the gripper, if any.

3. **End-effector wrenches $\mathcal{W}$:** The six-dimensional forces and torques at the TCP, in local coordinates.

Which combination of these three spaces (or *components*) span the a skill's state space $\mathcal{S}$ depend on the skill type and the needs of the application. A shadow skill for the `movel` URScript function, for example, would have a state space $\mathcal{S}_{\text{movel}} = \mathcal{P}$. A DMP for table wiping would have state space $\mathcal{S}_{\text{wipe}} = \mathcal{P}$ or $\mathcal{S}_{\text{wipe}} = \mathcal{C}$, depending on whether a Cartesian or joint-space DMP is used. A force-controlled insertion skill `Insert Moment` has state space $\mathcal{S}_{\text{insert}} = \mathcal{P} \times \mathcal{W}$. Whether or not to include a component in a skill's state representation ultimately depends on any downstream computation being performed. If e.g. a skill's parameters are optimized to produce smooth motions, the skill's state space should include $\mathcal{C}$. This work adopts the convention that skills are defined either in Cartesian or configuration space, but not



$\boldsymbol{\theta}_{\text{EOS}}$	$\boldsymbol{\theta}_{\text{succ}}$	$\boldsymbol{\theta}_{\text{pose}}$							$\boldsymbol{\theta}_{\text{wrench}}$					
0	0	0.4	-0.7	0.6	1.0	0.0	0.0	0.0	-0.2	0.9	0.4	0.1	0.8	-0.9
0	0	0.4	-0.6	0.5	0.9	0.1	0.0	0.1	-0.2	0.8	0.4	0.1	0.8	-0.9
0	0	0.5	-0.7	0.5	0.8	0.2	0.1	0.0	-0.1	0.6	0.2	0.2	0.7	-0.5
1	0	1.0	-0.6	-1.0	0.9	0.0	0.0	0.1	0.0	0.7	1.0	0.0	0.7	-0.6

Figure 2.5: Batched tensor representation of trajectories, by example of a spiral search skill. Left: 3 exemplary real-world spiral search trajectories executed on a UR5e industrial manipulator.

both. If downstream computations require the other representation, differentiable forward kinematics (FK) and IK operations are available (see Section 2.4.4.1).

Trajectory tensors ▷

As motivated in Section 2.3.1, a trajectory is a time series of states. Trajectories are variable-length: Different skills will produce trajectories of different length, as will subsequent executions of the same skill. As batch-wise parallel computation can greatly improve inference speeds in DCGs, a fixed-length representation of trajectories is desirable to enable batching. For this reason, the shadow skill DCG pads trajectories to a common length L by repeating the last state. To enable downstream computations to distinguish padding from regular states, an *end-of-sequence (EOS) token* is prepended to each state on the trajectory, which is 0 for regular states on the trajectory and 1 for padding.

Many skill representations have the notion of *success* of a skill, which typically denotes that all runtime conditions of the skill were met during execution and that all postconditions of the skill held after completion. Shadow skills extend each state on a trajectory by a *success token*, which is 1 for all states during which the skill execution is deemed successful, and 0 otherwise. Figure 2.5 illustrates the proposed tensor representation of trajectories.

2.4.1.2 Shadow Skill Learning

Posterior trajectory ▷

Shadow skills relate skill inputs $\boldsymbol{x}^{\bar{p}}$ and initial state θ_0 to the *posterior trajectory* $\boldsymbol{\theta}^{\bar{p}}$. The dual nature of NRPs requires shadow skills to be semantically equivalent to their corresponding source skills. For this reason, $\boldsymbol{\theta}^{\bar{p}}$ must reflect the ground-truth trajectory $\boldsymbol{\theta}^{p}$. Due to the fundamentally stochastic nature of many physical processes, real-world robot behavior cannot be modeled with perfect accuracy. $\boldsymbol{\theta}^{\bar{p}}$ can be, at best, a very good approximation of $\boldsymbol{\theta}^{p}$, and $\bar{p}$ can only be an approximate model of p. Nevertheless, it facilitates precise robot program optimization in complex, force-sensitive tasks, as evidenced by experiments in Chapter 3.

The need for $\boldsymbol{\theta}^{\bar{p}}$ to accurately reflect real-world robot behavior implies that parts of the shadow skill DCG must be *learnable*. A straightforward way to implement this is by using a differentiable autoregressive model, such as a recurrent neural network, trained to minimize some prediction loss $\mathcal{L}$ between the posterior and ground-truth trajectories:

$$\mathcal{L}(\boldsymbol{\theta}^{\bar{p}}, \boldsymbol{\theta}^{p}) = \mathrm{BCE}(\boldsymbol{\theta}^{\bar{p}}_{\mathrm{EOS}}, \boldsymbol{\theta}^{p}_{\mathrm{EOS}}) + \mathrm{BCE}(\boldsymbol{\theta}^{\bar{p}}_{\mathrm{succ}}, \boldsymbol{\theta}^{p}_{\mathrm{succ}}) + \mathrm{MSE}(\boldsymbol{\theta}^{\bar{p}}_{\theta}, \boldsymbol{\theta}^{p}_{\theta}), \tag{2.1}$$

◁ *Trajectory prediction loss*

the sum of the binary crossentropies between the predicted and ground-truth EOS and success tokens, respectively, as well as the mean squared error (MSE) between predicted and ground-truth states.[1]

Bootstrapping the posterior trajectory given $\boldsymbol{x}^{\bar{p}}$ and θ_0 is a challenging learning problem: The model must implicitly learn to plan low-level motion trajectories from a sparse set of parameters, and to predict the expected interaction of the robot with its environment. Recent publications in the field of informed machine learning have highlighted the potential of combining explicit, differentiable implementations of algorithms with learnable components (von Rueden et al., 2021). Some approaches such as differentiable particle filters (Jonschkowski et al., 2018; Wen et al., 2021) or differentiable physics engines (Degrave et al., 2019; Toussaint et al., 2018; Lutter et al., 2021) implement known algorithms or (physical) laws as DCGs, which allows for the first-order learning of physical parameters by iterative optimization. Other approaches include parameterizing algorithms at runtime with neural networks (Yang et al., 2023), integrating neural networks with ODE solvers (Chen et al., 2018) or directly modeling the solution curves of ODEs in the network architecture (Biloš et al., 2021). Such *algorithmic priors* promise to dramatically reduce the complexity of the learning problem – instead of learning a generative predictor of a (usually physical) process, the system must only learn to regress those aspects of the real-world process which deviate from the prior. Algorithmic priors have been shown to reduce the required amount of training data and to improve model accuracy, particularly under limited data regimes (von Rueden et al., 2021).

As the availability of high-quality training data is limited in physical application domains such as robotics, and as many manipulation tasks in service and industrial robotics require a high degree of reliability and precision, the shadow skill architecture leverages a differentiable path planner as an algorithmic prior whose outputs are adapted by a sequence-to-sequence neural architecture. Given skill parameters $\boldsymbol{x}^{\bar{p}}$, the differentiable planner bootstraps a *prior trajectory* $\tilde{\boldsymbol{\theta}}^{\bar{p}}$. $\tilde{\boldsymbol{\theta}}^{\bar{p}}$ is then adapted by a recurrent neural network to produce the posterior trajectory $\boldsymbol{\theta}^{\bar{p}}$, which approximates the real-world ground-truth robot behavior $\boldsymbol{\theta}^{p}$.

◁ *Prior trajectory*

2.4.1.3 Differentiable Cartesian Priors

Both Cartesian and joint-space algorithmic priors are considered. The present Section focuses on Cartesian priors; Section 2.5 introduces a collision-free differentiable motion planner, which is capable of generating $\mathcal{C}$-space trajectories satisfying

---

[1] An early formulation of NRPs used a loss function that split the trajectory prediction loss into position and orientation components (see Alt et al. (2021) for details).

collision-freeness, smoothness and other constraints. For skills whose behavior is defined in Cartesian space, e.g. force-controlled skills with constraints specified in the task frame formalism (Bruyninckx and De Schutter, 1996) or similar conventions, or for simple Cartesian motion skills such as the URScript `movel` primitive introduced in Section 2.3.1, Cartesian planners can bootstrap suitable prior trajectories.

Example: Spiral Search The ARTM skill `Spiral Search Relative` takes parameters `MinForce`, `MaxForce`, `MinDepth`, `MaxDepth`, `ExtentsX`, `ExtentsY`, `PathIncrement` as well as velocity `Vel` and acceleration `Acc` parameters. It executes an Archimedean spiral motion in the XY plane relative to start state $\theta_0^{\bar{p}} \in \mathcal{P} \times \mathcal{W}$, where the total size of the spiral along the X and Y axes is given by `ExtentsX` and `ExtentsY` and the scaling factor `PathIncrement` determines the number of windings. During execution, the end effector pushes against the workpiece with a force greater than `MinForce`. If the end effector drops by more than `MinDepth` along the local Z axis, the motion ends and is deemed successful; if the spiral is executed without such a drop or if the maximum force `MaxForce` or maximum depth `MaxDepth` is exceeded during the motion, the motion ends and the skill is considered to have failed. A differentiable Cartesian planner for spiral search can be implemented by sampling an Archimedean spiral[2]:

```python
def archimedean_spiral(start: Tensor = 0.0,
                       end: Tensor = 1.0,
                       path_increment: Tensor = 0.01) -> Tuple[Tensor, Tensor]:
    a = path_increment / (2 * pi)
    angle = tensor(0.0)
    angle_max = 0.5 / a
    max_length = angle_max * sqrt(1 + angle_max**2) + log(angle_max + sqrt(1 +
    angle_max**2))
    time = 0
    points = []
    times = []
    while time <= end:
        radius = angle * a
        point = stack([radius * cos(angle), radius * sin(angle)])
        current_length = angle * sqrt(1 * angle**2) + log(angle + sqrt(1 +
            angle**2))
        time = current_length / max_length
        if time + eps > start:
            if len(points) == 0 or norm(point - points[-1]) > 0.05:
                points.append(point)
                times.append(time)
        angle = angle + path_increment
    return stack(points), stack(times)
```

Listing 2.1: PyTorch-based differentiable implementation of a Cartesian spiral generator.

The resulting sampled path is differentiable with respect to the `PathIncrement` parameter. It is scaled according to `ExtentsX` and `ExtentsY` and re-sampled with

---

[2] This work uses Python syntax for (pseudo-)code. Unless otherwise specified, functions and datatypes are Python builtins or provided by PyTorch (Paszke et al., 2019). Imports are omitted for brevity.

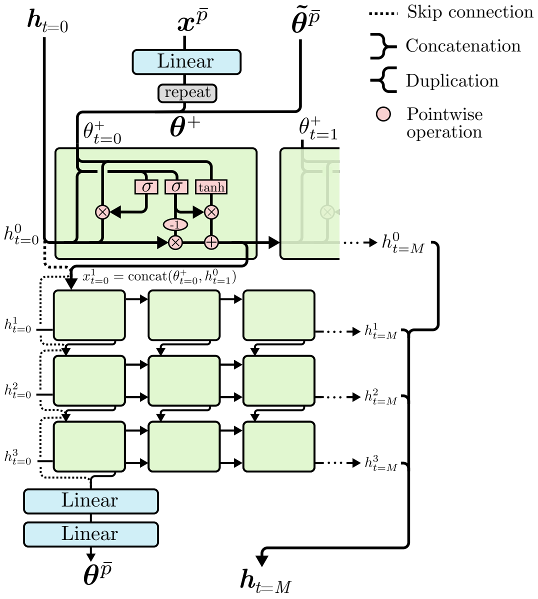


Figure 2.6: Stacked Gated Recurrent Unit (GRU) sequence-to-sequence neural architecture. The visualization of individual GRUs has been adapted from Olah (2015).

a trapezoidal velocity profile based on `Vel` and `Acc`, using a differentiable PyTorch implementation of the trajectory resampling algorithm provided by the Orocos Kinematics and Dynamics Library (KDL) (Smits, 2020). The resampled spiral is then translated to the correct absolute workspace pose given $\theta_0^{\bar{p}}$. Similar differentiable priors can be constructed for linear, circular, and other Cartesian motions (see Section 3.1.3).

For some skills, such as Cartesian free-space motions, the differentiable prior suffices to approximate the posterior trajectory; in this case, $\tilde{\boldsymbol{\theta}}^{\bar{p}} \approx \boldsymbol{\theta}^{\bar{p}} \approx \boldsymbol{\theta}^{p}$, and the shadow skill does not need a learnable component at all. This is the case for free-space motions defined in Cartesian space, such as the URScript function `movel`, or DMPs which do not interacting with the environment. Note, however, that the approximation will be imperfect at best: The real-world dynamics of the robot will deviate from the pre-programmed behavior, as a robot's dynamics will always show slight variations, e.g. due to wear and tear, payload-induced deformation of links or similar factors (Raible et al., 2023b). For some applications, however, submillimeter accuracy is not required and NRPs can be used for model-based optimization without training. One such application, the optimization of approach motions for a household fetch-and-place task, is shown in Section 3.1.3.

2.4.1.4 Skill-Level Neural Adapters

Intelligent robots interact with the physical environment to achieve their objectives. From stocking supermarket shelves to assembling PCBs, most practical use cases for intelligent robots require robot skills that react dynamically to their environment, be it via force control, visual servoing, or other types of dynamic robot-environment interaction. A DNN permits to learn this robot-environment interaction: If the training dataset contains a sufficiently diverse sample of skill executions in different environments, or variations of the same environment, then the trajectory predicted by the model will be different for every parameterization and start state. More importantly, if the shadow skill was finetuned for one specific robot and environment, knowledge about this particular robot and environment will be implicitly represented in the weights of the DNN – and the trajectory predicted by the shadow skill will not only be conditioned on the skill inputs and start state, but also on the learned, latent representation of the environment, robot kinematics and dynamics. To realize prediction of expected real-world trajectories in a data-efficient way, a predictor for the posterior trajectory $\boldsymbol{\theta}^{\bar{p}}$ can be conditioned on the prior $\tilde{\boldsymbol{\theta}}^{\bar{p}}$. This leads to a two-part DCG, in which the prior $\tilde{\boldsymbol{\theta}}^{\bar{p}}$ is first bootstrapped by a differentiable planner, and the planning results are then modified by a *neural adapter* to reflect the learned real-world dynamics and environment interactions.

Neural adapter ▷

The proposed DNN architecture for neural adapters is shown in Figure 2.6. It consists of a four-layer stacked GRU (Cho et al., 2014) with residual connections, linear input and output layers, scaled exponential linear unit (SELU) activations (Klambauer et al., 2017) and dropout layers after each GRU (Srivastava et al., 2014). The choice of GRUs as recurrent networks is motivated by their parameter efficiency and comparatively good ability to model correlations across long time horizons (Yang et al., 2020b). A GRU hidden size of 128 has proven sufficient for all tested skills; rather than the width of the hidden state, prediction accuracy is more dependent on the number of stacked GRUs. 4 have been empirically found sufficient for most skills. Dropout layers improve generalization, particularly on small datasets (Srivastava et al., 2014). A dropout rate of 0.2 has been empirically found to yield best results. SELU activations have been shown to prevent vanishing or exploding gradients (Klambauer et al., 2017), facilitating the training of deep architectures and permitting the construction of the end-to-end differentiable, deep shadow program DCGs introduced in Section 2.4.

The posterior trajectory $\boldsymbol{\theta}^{\bar{p}}$ not only depends on the prior and the latent representation of the environment implicit in the model's weights, but also on the skill inputs. Consider the `Spiral Search Relative` skill introduced in Section 2.4.1.3: Whether the search drops into a hole, the depth to which it drops and the dynamics with which the robot brakes depend on the inputs of the skill, notably on `MinDepth`, `MinForce`, `Vel` and `Acc`, and are not reflected in the prior. As the model expects a sequence as inputs, $\boldsymbol{x}^{\bar{p}}$ is concatenated to every state of the prior $\tilde{\boldsymbol{\theta}}^{\bar{p}}$ to form the *augmented prior* $\boldsymbol{\theta}^{+}$ (see Figure 2.6).

Augmented prior ▷

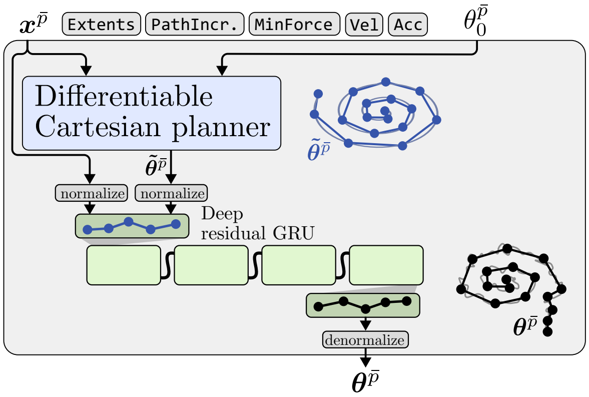

$\boldsymbol{x}^{\bar{p}}$
Extents
PathIncr.
MinForce
Vel
Acc
$\theta_0^{\bar{p}}$
Differentiable Cartesian planner
$\tilde{\boldsymbol{\theta}}^{\bar{p}}$
$\tilde{\boldsymbol{\theta}}^{\bar{p}}$
normalize
normalize
Deep residual GRU
$\boldsymbol{\theta}^{\bar{p}}$
denormalize
$\boldsymbol{\theta}^{\bar{p}}$

Figure 2.7: Shadow skill DCG for `Spiral Search Relative`, which combines a differentiable Cartesian planner with a sequence-to-sequence neural network.

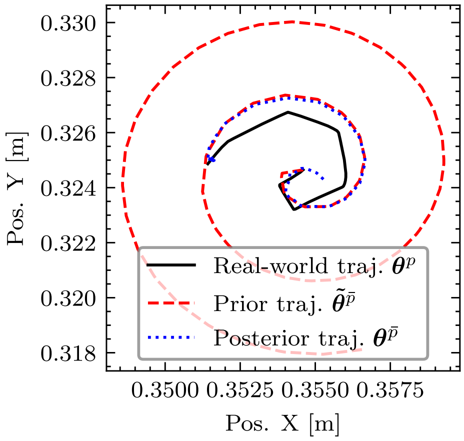


Figure 2.8: Prior (dashed), predicted (dotted) and ground truth (solid) Cartesian spiral trajectories.

2.4.2 Evaluating Shadow Skills

◁ *Shadow skill evaluation*

Both presented shadow skill formulations – purely differentiable planning-based models, and sequence-to-sequence neural models with differentiable priors – are DCGs which relate inputs $\boldsymbol{x}^{\bar{p}}$ and initial state θ_0^{p} to posterior trajectory $\boldsymbol{\theta}^{\bar{p}}$. *Evaluating a shadow skill* means performing a *forward pass* from leaves ($\boldsymbol{x}^{\bar{p}}$ and $\theta_0^{\bar{p}}$) to root ($\boldsymbol{\theta}^{\bar{p}}$). A shadow skill can then be viewed as a differentiable function $\bar{p} : \mathbb{R}^N \times \mathcal{S} \to \mathcal{S}^M$, that permits the computation of $\frac{\partial \bar{p}(\boldsymbol{x}^{\bar{p}}, \theta_0^{\bar{p}})}{\partial \boldsymbol{x}^{\bar{p}}}$ and $\frac{\partial \bar{p}(\boldsymbol{x}^{\bar{p}}, \theta_0^{\bar{p}})}{\partial \theta_0^{\bar{p}}}$, the gradients of $\bar{p}$ with respect to the leaves (or any other node in the DCG).

◁ *Normalization*

Input normalization and denormalization Shadow skill DCGs with neural adapters contain *normalization* and *denormalization* operations before and after the DNN (see Figure 2.7). Normalizing the inputs of neural networks to the ranges $[-1, 1]$ or $[0, 1]$ is considered best practice to prevent the first nonlinearity in the network from saturating immediately: The SELU activation used in both neural skill architectures is near-constant for values less than -5 and linear for values greater than 0. For this reason, inputs are normalized to the range $[-1, 1]$, which will yield input values close to the nonlinear region of SELU. Moreover, the inputs and states contain values at very different scales: Consider `Spiral Search Relative`, whose `ExtentsX` may be very small (on the order of 0.001-0.003 m), but whose `MaxForce` may be three orders of magnitude larger (5-8 N). In the experiments presented in Chapter 3, inputs to networks (both $\boldsymbol{x}^{\bar{p}}$ and $\theta_0^{\bar{p}}$ or $\tilde{\boldsymbol{\theta}}^{\bar{p}}$) are normalized to the range $[-1, 1]$ using min-max-normalization over the range of the training dataset, and network outputs are again denormalized using the same scale. Denormalization of outputs permits downstream computations, such as the differentiable forward or inverse kinematics (see Section 2.4.4.1) or user-defined objective functions (see

```python
def forward_skill(p: DCG, x: Tensor, theta_0: Tensor) -> Tensor:
    theta_prior = p.differentiable_planner(x, theta_0)
    if p.has_dnn():
        x, theta_prior = normalize(x, theta_prior)
        theta = p.dnn(x, theta_prior)
        theta = denormalize(theta)
    else:
        theta = theta_prior
    return theta
```

Listing 2.2: Forward pass through a differentiable shadow skill.

Chapter 3) to be expressed in terms of their "natural" domains: A differentiable IK module, for example, may expect a joint configuration in radians, as opposed to the dimensionless, normalized output space of a neural network.

Forward pass The forward pass through a shadow skill $\bar{p}$ (in pseudocode: `p`) is shown in Listing 2.2. Additional details such as differentiable IK and FK operations are omitted. An invocation of the differentiable planner, given skill inputs `x` and start state `theta_0`, yields prior trajectory `theta_prior`, which is normalized and passed through the DNN. The denormalized posterior trajectory `theta` is returned. `forward_skill` preserves gradients, permitting the derivation of functions of `theta` with respect to `x`.

2.4.3 Training Shadow Skills

The shadow skill $\bar{p}$ is trained to minimize the trajectory prediction loss (Equation 2.1) on a dataset $\mathcal{D}$ consisting of past executions of the corresponding source skill p. $\mathcal{D}$ contains 3-tuples $(\boldsymbol{x}^p, \theta_0^p, \boldsymbol{\theta}^p)$, representing the skill inputs, initial states and ground-truth trajectories, respectively. The neural network parameters φ of $\bar{p}$ are trained to minimize the expected trajectory prediction loss over $\mathcal{D}$:

Shadow skill training loss ▷

$$\varphi^* = \arg\min_{\varphi} \sum_{(\boldsymbol{x}^p, \theta_0^p, \boldsymbol{\theta}^p) \in \mathcal{D}} \mathcal{L}(\bar{p}(\boldsymbol{x}^p, \theta_0^p), \boldsymbol{\theta}^p) \tag{2.2}$$

The training mechanism reflects current best practices. The AdamW optimizer (Loshchilov and Hutter, 2018) is used with a relatively low initial learning rate ($\alpha =$ 5e-5) and a linear learning rate scheduler. Dropout is activated during training, but deactivated during evaluation. The number of epochs depends on the size of the training dataset, though training typically converges in under 1000 epochs. In line with empirical studies on the impact of batch sizes on network performance, small-to-medium batch sizes of 32 or 64 provide a good trade-off in convergence speed and achieved accuracy (Masters and Luschi, 2018; He et al., 2019a; Kerley et al., 2023).

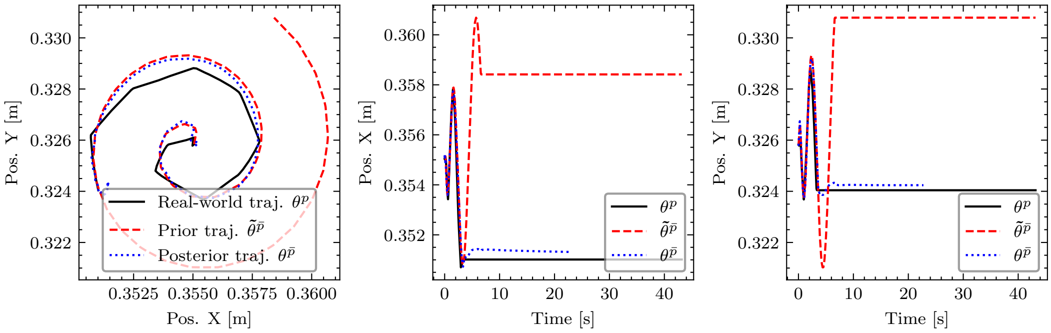


Figure 2.9: Ground-truth (solid), prior (red) and posterior (blue) Cartesian end-effector trajectories predicted by shadow skill $\bar{p}$ for a spiral search task.

Precomputed prior trajectories Shadow skills with a sequence-to-sequence architecture compute the prior trajectory $\tilde{\boldsymbol{\theta}}^{\bar{p}}$ as part of their forward pass. Depending on the implementation of the differentiable planner, bootstrapping $\tilde{\boldsymbol{\theta}}^{\bar{p}}$ may be computationally expensive and can dominate the training time. From a pure training perspective, evaluating the differentiable planner is not required: In the DCG, the differentiable planner is upstream of the neural network and does not contain learnable parameters. For this reason, at training time, the differentiable prior can be bypassed and the neural network can be provided with a pre-computed prior trajectory $\tilde{\boldsymbol{\theta}}^{\bar{p}}$. The training dataset $\mathcal{D}$ is then extended to contain 4-tuples $(\boldsymbol{x}^p, \theta_0^p, \tilde{\boldsymbol{\theta}}^{\bar{p}}, \boldsymbol{\theta}^p)$, which comprise the pre-computed priors $\tilde{\boldsymbol{\theta}}^{\bar{p}}$ for $\boldsymbol{x}^p$ and θ_0^p. This incurs a one-time additional computational cost at dataset creation time, but saves a repeated computational cost at training time, as the differentiable planner would otherwise be evaluated at every epoch. When the trained shadow skill is evaluated, the differentiable prior cannot generally be bypassed, as the inputs and initial states will generally be different from those present in the dataset.

Data collection $\mathcal{D}$ is collected by executing the source skill p for a range of inputs $\boldsymbol{x}^p$ and start states θ_0^p and measuring the resulting trajectories $\boldsymbol{\theta}^p$. $\boldsymbol{x}^p$ and θ_0^p should be sampled uniformly over the domain of p. Both the input and state domains as well as the distribution of $\{\Theta_t\}$, the trajectory-generating stochastic process that incorporates the stochastic characteristics of the environment, must be the same at data-collection time as at evaluation time. If the trained shadow skill $\bar{p}$ is executed with inputs and start states outside of its training data domain, its prediction accuracy will be considerably reduced (Wang et al., 2023a); likewise, the posterior trajectories $\boldsymbol{\theta}^{\bar{p}}$ will reflect the distribution of the training data, and cannot reflect changes to e.g. the environment at runtime without further information. Section 3.2 showcases how continuous finetuning can ensure that $\bar{p}$ reflects dynamically changing environments. In a different line of research, my colleagues and I explore the use of multimodal Transformer architectures to condition $\bar{p}$ on a representation of the current environment at runtime (Kienle et al., 2024).

Example: Spiral search Figure 2.9 shows an exemplary posterior, prior and ground-truth trajectory for a shadow skill for an ARTM `Spiral Search Relative`. Details on the data collection setup and hyperparameters are provided in Section 4.3.2 and Appendix A. Figure 2.9 illustrates the roles of the prior (here, the Cartesian differentiable spiral motion planner) and the neural adapter. The neural adapter predicts real-world semantics such as search success as well as the dynamics of failed or successful search, in which case the search ends before the spiral is completed. At $t = 4$s in Figure 2.9, the workpiece drops into the hole and the search ends. Note that the ground-truth trajectory is not a perfect spiral, due to the robot-specific implementation of the `Spiral Search Relative` ARTM skill. On ABB robots, the spiral is split into linear segments, giving the resulting trajectory an angular appearance. The posterior trajectory is regularized by the prior, causing it to ignore robot-specific low-frequency noise such as the ABB-specific angularity of spiral motions. It strongly deviates from the prior when (here, correctly) predicting that the robot has found the hole. Figure A.1 shows additional trajectories for different parameterizations and hole positions.

2.4.4 Shadow Programs

Differentiable shadow skills represent robot skills as trainable DCGs that are differentiable end-to-end. When combined with differentiable operations, they form DCGs representing complex robot behavior arising from the combination of several skills that can fulfil complex, real-world tasks. By virtue of structural and semantic equivalence, shadow skills act as differentiable, predictive models of robot skills. Shadow programs are differentiable, predictive models of source programs; they must therefore ensure semantic and structural equivalence at the *task level*. Shadow programs ensure *structural equivalence* via composition of shadow skills to complex DCG that mirror the composite structure of robot programs and the hierarchical nature of tasks. The mechanisms by which this is achieved are described in Sections 2.4.4.1, 2.4.4.2 and 2.4.4.3. Shadow programs achieve *semantic equivalence* with their corresponding source programs by being trained to reflect task-level semantics, arising from the execution of several skills in the environment. Section 2.4.4.4 introduces a mechanism for learning execution semantics that span multiple skills.

2.4.4.1 Differentiable Kinematics

Shadow skills can be defined in either Cartesian or configuration space, and the tensor-based state representation introduced in Section 2.4.1.1 supports both modalities. The combination of shadow skills to complex DCGs representing programs necessitates gradient-preserving mechanisms to convert between Cartesian and configuration space. For this reason, differentiable FK and IK modules are introduced. Based on these modules, Section 2.4.4.2 introduces a general-purpose mechanism for chaining shadow skills and propagating gradients through shadow programs.

Differentiable forward kinematics The formulation of a differentiable FK for serial manipulators is straightforward. FK is a function $f_\varphi : \mathcal{C} \rightarrow \mathcal{P}$: It maps every joint configuration $\theta \in \mathcal{C}$ to exactly one Cartesian end-effector pose $q \in \mathcal{P}$, given some parameters φ describing the robot's kinematics. The Denavit-Hartenberg (DH) (Denavit and Hartenberg, 1955) and Modified DH (Craig, 2004) conventions represent a robot's kinematics by its link lengths as well as rotational and translational joint offsets. FK then reduces to a series of matrix multiplications, which are differentiable operations with respect to the input joint angles θ. Most common differentiable kinematics libraries are based on variations of this approach (Meier et al., 2022; Mölschl et al., 2023). Alternatives to DH-based FK, such as the product-of-exponentials formula (Brockett, 1984), are likewise differentiable in principle, though no differentiable implementations exist. NRPs use the differentiable FK proposed by Meier et al. (2022). Based on the Unified Robot Description Format (URDF) description of the robot's kinematic chain, a tree of differentiable rigid bodies is instantiated, which are connected by prismatic, revolute, continuous or fixed joints. Given a joint configuration θ, the workspace poses of the links are recursively updated, starting from the robot base. I refer to Meier et al. (2022) for details.

Differentiable inverse kinematics Unlike FK, IK is not a function: For a given Cartesian pose q, a kinematic chain can have either no IK solutions, if q is not reachable; one single solution; multiple solutions; or infinitely many solutions, in the case of kinematically redundant manipulators (DeMers and Kreutz-Delgado, 1997). Provided q is in the workspace of the robot, all IK algorithms face the fundamental problems of selecting the "best" redundant configuration for use in downstream computations; generating smooth $\mathcal{C}$-space trajectories for given Cartesian trajectories without oscillating between redundant configurations; or handling transitions between disconnected regions of $\mathcal{C}$-space, if no smooth $\mathcal{C}$-space trajectory exists.

Haug (2021) provides a mathematical theory of differentiable IK. He defines the *regular manipulator configuration space* of an N-DoF serial manipulator as an N-dimensional differentiable manifold with one or more maximal, disjoint, singularity free, path-connected components. Within these components, smooth control is possible and gradients are well-defined. Their boundaries, however, are sets of singularities, implying that IK for serial manipulators is only *locally* differentiable, within the confines of individual singularity-free components. Singularity avoidance and graceful handling of unstable gradients due to transitions between disjoint $\mathcal{C}$-space components remains an open engineering challenge.

From a computational perspective, a differentiable, closed-form analytical IK is desirable, as analytical IK formulations typically incur less computational overhead than numerical solvers. However, analytical IK solutions do not exist for redundant manipulators. Various numerical solvers have been proposed (Xie et al., 2022). Jacobian-based methods such as the Jacobian pseudoinverse (Dulęba and Opałka,

2013) or the extended Jacobian inverse (Janiak and Tchoń, 2008) are compute-efficient but numerically unstable near singularities (Buss, 2004). Higher-order optimization algorithms such as Levenberg-Marquardt (LM) (Goldenberg et al., 1985) or limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) (Liu and Nocedal, 1989) exploit gradient information to approximate the target configuration and tend to converge quickly, while remaining comparatively stable in the neighborhood of singularities (Xie et al., 2022).

This work proposes a differentiable IK based on second-order L-BFGS optimization over the differentiable FK outlined above, to iteratively approach the IK solution θ^* from an initial configuration θ^{init} known to lie within the same disjoint, singularity-free $\mathcal{C}$-space component as θ^*. L-BFGS is chosen as it is considered more robust than LM and exhibits faster convergence for initial configurations farther away from the solution (Xie et al., 2022). A PyTorch implementation is provided below.

```
def inverse_kinematics(q: Tensor,
                       max_num_iter: int,
                       theta_init: Tensor) -> Tensor:
    theta = theta_init

    def optim_closure(theta_g: Tensor) -> Tuple[Tensor, Tensor]:
        q_curr = forward_kinematics(theta_g)
        loss = sum((lstsq(q, q_curr) - eye(4))**2)
        grad = autograd.grad(
            inputs=theta_g,
            outputs=loss,
            ...
        )[0]
        return loss, grad

    theta_g = autograd.Variable(theta, requires_grad=True)
    optim = LBFGS(max_iter=max_num_iter)

    with set_grad_enabled(True):
        theta_g, loss = optim.step(theta_g, optim_closure)

    return theta_g
```

Listing 2.3: Differentiable inverse kinematics for serial N-DoF manipulators.

It uses `torch.optim.LBFGS`, the PyTorch implementation of L-BFGS, in conjunction with the `torch.autograd` automatic differentiation engine (Paszke et al., 2017) to construct a DCG at runtime. When some downstream computation requires the gradients of θ^* (`theta_g`) with respect to e.g. q or any variable upstream of q, `autograd` can backpropagate through the IK procedure.

The IK problem for trajectories requires special care since successive end-effector poses on a trajectory must be mapped to the same singularity-free $\mathcal{C}$-space component to avoid discontinuities. In this work, this is addressed by using the IK solution θ_i^* of the ith point on a trajectory as the initial configuration θ^{init} for the $i+1^{\text{th}}$ point, successively constructing a contiguous $\mathcal{C}$-space trajectory.

The proposed differentiable IK module leverages the computational efficiency of second-order optimization, which converges sufficiently quickly for practical use (see Sections 2.5 and 3.3). Due to the nature of the IK problem, inherent limitations remain (Haug, 2021). While proper choice of θ^{init} can ensure starting in the same connected $\mathcal{C}$-space component as θ^*, such a proper choice cannot be guaranteed, as θ^* is a priori unknown. For trajectories, the assumption that θ^*_{i+1} will be close to θ^*_i generally holds, and provides a good heuristic for the choice of θ^{init}. Like all iterative optimization-based IK solvers, the proposed L-BFGS-based solver can converge in local minima, even within the same connected $\mathcal{C}$-space component. Random restarts may mitigate this issue (Xie et al., 2022).

2.4.4.2 Composite Program Structure

In order for shadow programs to act as differentiable surrogates of robot programs, they must be structurally equivalent to their source programs. As a consequence, the shadow program DCG is sequentially and hierarchically composed in a manner that mirrors the sequential and hierarchical composition afforded by most skill-based robot program representations (see Section 2.3):

> D A ***shadow program*** $\bar{P}$ is a DCG of shadow programs and shadow skills linked by differentiable operations. Its leaves are the program inputs $\boldsymbol{x}^{\bar{P}}$ and initial state $\theta_0^{\bar{P}}$. Its root is the posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$. $\bar{P}$ is structurally and semantically equivalent to a source program P.

This recursive definition implies that shadow programs can be hierarchically nested to arbitrary depth. It also implies that shadow programs and shadow skills have the same signature: Both translate a set of input parameters and a robot or world state to a posterior trajectory. Per the definition of structural equivalence (see Section 2.2.2.1), for each subprogram P_i in the source program P, there is a corresponding shadow (sub-)program $\bar{P}_i$ in $\bar{P}$; for each skill p_j in P, there is a corresponding shadow skill $\bar{p}_j$ in $\bar{P}$; and for each input parameter x_k^P in $\boldsymbol{x}^P$, there is a corresponding parameter $x_k^{\bar{P}}$ in $\boldsymbol{x}^{\bar{P}}$. Semantic equivalence implies that for a given set of inputs $\boldsymbol{x}^P$ and initial state θ_0^P, the posterior $\boldsymbol{\theta}^{\bar{P}}$ predicted by $\bar{P}$ approximates the ground-truth trajectory $\boldsymbol{\theta}^P$.

Figure 2.10 illustrates a NRP for a peg-in-hole task and highlights the one-to-one correspondence between the source program and its differentiable shadow. Like shadow skills, shadow programs are DCGs, permitting the computation of $\frac{\partial \boldsymbol{\theta}^{\bar{P}}}{\partial \boldsymbol{x}^{\bar{P}}}$, the gradient of the posterior trajectory with respect to the program inputs, via automatic differentiation. The hierarchical composition into a DCG is achieved by assigning the last state of the posterior trajectory $\boldsymbol{\theta}_M^{\bar{P}_i}$ of shadow (sub-)program $\bar{P}_i$ to the initial state $\theta_0^{\bar{P}_{i+1}}$ of the next shadow (sub-)program $\bar{P}_{i+1}$. Besides realizing the seemingly trivial condition for physically plausible trajectories that the succeeding skill begins where the preceding skill ended, this mechanism enables gradient computation and backpropagation of errors through the entire program graph. Most robot applications have interdependencies between skills: In peg-in-hole tasks such as that illustrated in Figure 2.3, the behavior of search skills critically depends on the preceding approach motions. A suboptimally parameterized approach will require a larger search region to find a hole for insertion. In service robotics, grasping is a canonical example: The appropriate parameterization of a grasping skill, such as the initial joint configuration of a robotic hand, depends on the approach motion for the grasp; an approach motion, in turn, can only be collision-free if the preceding motion did not end in a collision. In the shadow program DCG, the inherent statefulness of robot manipulation programs is captured by the literal passing of the state between successive subprograms. This permits the joint learning (see Section 2.4.4.4) and optimization (see Chapter 3) of program parameters across complex program hierarchies, taking inter-skill dependencies into account.

2.4.4.3 Control Flow

NRP shadow programs map sequential control flow by passing the robot state from one skill or subprogram to the next, and by concatenating predicted posterior trajectories along the time dimension. Other control flow structures such as conditional branching (`if/else`) or loops (`for`, `while`) require additional consideration.

Conditional branching Differentiable programming with conditional branching via `if/else` or `switch/case` statements requires particular care. Each branch is a separate differentiable subgraph; the branches eventually merge back into one single DCG. Which subgraph is evaluated during the forward pass depends on the branch condition and the value of the corresponding variables at the time the branch condition is evaluated. While gradients can still be backpropagated through the evaluated subgraph to leaf nodes upstream of the branch condition, gradients cannot be computed for the subgraphs corresponding to the branches that were not evaluated. When performing gradient descent over a DCG with conditional branches, updating variables upstream of the branch condition may lead to different branches being evaluated at different iterations of gradient descent, leading to (possibly) vastly different values at the root of the DCG. This translates into possibly vastly different loss values and oscillating optimizer behavior.

Conditional branching is crucial for error handling routines, in which a subprogram is executed conditional on the detection of known or unexpected errors at runtime. The shadow program DCG in Figure 2.10 contains a subprogram for an `Error Handler` ARTM control structure. `Error Handler` executes one subprogram (the `if` branch) if the preceding subprogram terminated in an error state, and another subprogram (the `else` branch) otherwise. In Figure 2.10, the `Spiral Search Relative` skill is only executed if the preceding contact motion terminated successfully, i.e. made contact with the surface. In the corresponding shadow subprogram ($\bar{P}_1$ in Figure 2.10), the shadow skill $\bar{p}_3$ modeling the search motion is only executed if the success label of the start state $\theta_0^{\bar{P}_1}$ is not below 0.5; otherwise, $\theta_0^{\bar{P}_1}$ is passed directly to the subsequent subprogram. The predicted posterior trajectory

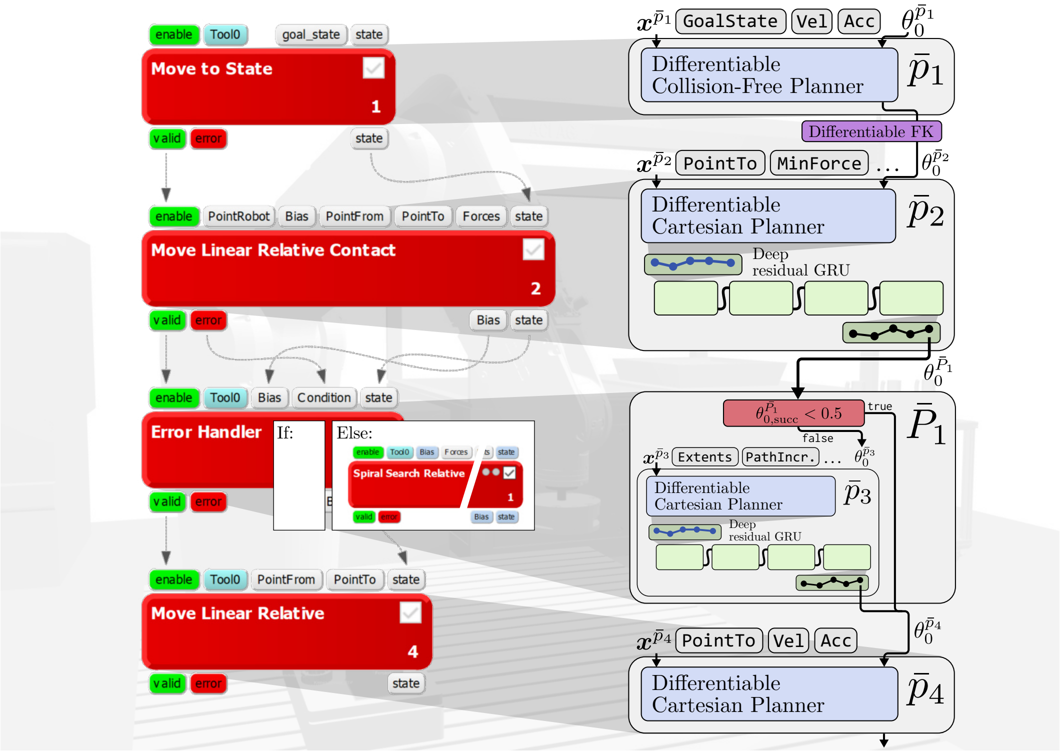


Figure 2.10: A NRP for a peg-in-hole task. The source program (left) is represented in an industrial skill-based program representation (ARTM, Schmidt-Rohr et al. (2013)); the shadow program (right) is represented as a DCG. Structural and semantic equivalence enables bidirectional conversion between the two representations. Control flow structures such as conditional branching (`Error Handler`) are mapped to corresponding structures in the DCG ($\bar{P}_1$). The depicted shadow program DCG contains shadow skills with and without neural adapters (green).

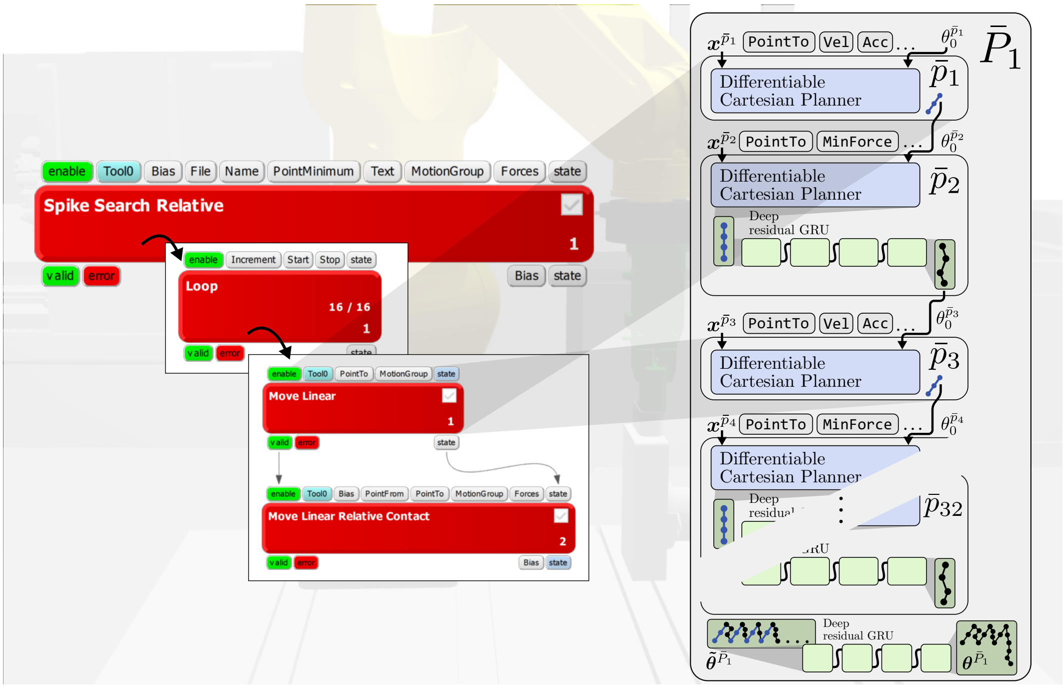


Figure 2.11: A NRP for a tactile probe search task. A `Spike Search Relative` ARTM skill is mapped to a shadow subprogram ($\bar{P}_1$) containing an “unrolled” sequence of shadow skills ($\bar{p}_1$, ..., $\bar{p}_{32}$). Task-level semantics must be learned at the hierarchy level. Neural adapters (green) permit learning of task-level semantics at any level in the program.

$\boldsymbol{\theta}^{\bar{P}}$ for the entire shadow program contains only the predicted trajectory for the branch that was evaluated during the forward pass. This mechanism generalizes to all branching control structures where the branch condition is a function of the start state. Other branch conditions are, in principle, possible, but require additional care, as all variables evaluated in the branch condition must be available as part of the forward pass. It is possible, for instance, to construct a branch condition based on e.g. sensor values, provided these values are injected into the DCG during the forward pass.

Loops Figure 2.11 shows the shadow DCG for probe search, a search strategy by which the robot repeatedly touches (“probes”) a surface until a feature, such as a hole, is detected. Variations of probe search are commonly used in PCB assembly and similar tasks requiring the manipulation of small, sensitive parts under uncertainty. In the ARTM source program representation, the `Spike Search Relative` skill implements probe search. It consists of a `Loop` control flow structure, which executes a predefined probe pattern by repeatedly moving to the next probe position (via a `Move Linear` skill) and touching the surface (`Move Linear Relative Contact`). Loops are mapped to the shadow program DCGs by *unrolling*, i.e.

sequential chaining of M copies of the shadow subprogram for M loop iterations. Neural network weights are shared between the copies, but gradients are not. The predicted posterior trajectory for the looped subprogram then consists of the concatenated posterior trajectories of the shadow subprograms in the unrolled sequence. Experiment 3.2.3 evaluates first-order optimization over robot programs involving loops in greater detail.

2.4.4.4 Task-Level Neural Adapters

In the context of manipulation tasks, the ground-truth motions performed by the robot will generally differ from planned motions, as the robot reacts to contact forces with objects, avoids collisions or detects failure and aborts a motion. At the skill level, sequence-to-sequence neural networks provide a mechanism for learning these dynamics. At the task level, a similar learning mechanism is required. For complex real-world source programs, the assumption that the posterior trajectory of a (sub-)program is merely the concatenation of its constituent skills or subprograms must be relaxed.

Consider the robot program in Figure 2.11 (left), which illustrates an ARTM source program for finding a hole on a surface by repeated force-controlled probing. It approaches each point in the sequence and executes a force-controlled downward motion (“probe”) with a given depth along the local Z axis, stopping the current probe and moving to the next point if it hits the surface with a contact force greater than `MinForce`, and aborting the search altogether when `MaxForce` is exceeded. The search is considered unsuccessful if it is aborted, or if all probes have made contact with the surface. If a probe fails to make contact, the search is considered successful and no further probes are executed. The source program for probe search consists of a sequence of linear approach and contact motions. The program-level semantics, such as early termination and the determination of program success, is handled by additional logic in the respective source program representation (e.g. URScript).

Probe search illustrates two cases in which the execution semantics of the source program do not correspond to sequential execution of the constituent skills. First, the search terminates early, either when a permissible force threshold is exceeded or when the hole is found. In this case, some skills in the sequence will not be executed at all, and the resulting ground-truth trajectory $\boldsymbol{\theta}^P$ is not identical to the concatenation of the trajectories $\boldsymbol{\theta}^{p_i}$ of all constituent skills. Second, the success of the program is determined by some non-trivial logic over the success of the last executed skill: If the skill was successful, it made contact, and the search was unsuccessful; if it was unsuccessful, but did not exceed force limits, it did not make contact, and the search was successful; if it exceeded force limits, the search was unsuccessful. NRPs aim to be a universal neurosymbolic program representation; for this reason, such program-level semantics must be supported in principle by the architecture.

The representation of dynamic changes to the control flow of programs, such as early termination of a search program, poses severe challenges for automatic differentiation. Modern differentiable programming frameworks such as PyTorch perform automatic differentiation over an evaluation trace constructed during the forward pass; the DCG over which automatic differentiation is performed contains only those operations on the control flow path taken during execution. The counterfactual, i.e. the situation in which the search had not been aborted, is not represented; neither are the operations which led to the chosen control flow path. In other words, automatic differentiation "is blind with respect to any operation, including control flow statements, which do not directly alter numeric values" (Baydin et al., 2018). For probe search, the control flow partly depends on features of the environment, such as the position of the hole. As with shadow skills, shadow programs must incorporate some learned representation of the interactions of the robot with the environment *at the program level*. Likewise, representing logical operations in DCGs is challenging. Boolean operators are, in principle, not differentiable. While several approaches for differentiable logics such as Differentiable Fuzzy Logics (van Krieken et al., 2022) or vectorized first-order logic rules (Rocktäschel, 2018) have been proposed, the field of differentiable logics remains pre-paradigmatic, and no general-purpose mechanism for automatic differentiation of logical operators exists.

NRPs propose a general-purpose, learning-based approach to reflect both environment dependent control flow as well as task-level success semantics. The core idea of adapting a prior trajectory via a sequence-to-sequence model to reflect ground-truth behavior that deviates from the prior has been introduced in the context of shadow skills in Section 2.4.1. As shadow skills and shadow programs share the same signature, the principle can be directly transferred to the program level, with the concatenation of the posteriors $\boldsymbol{\theta}^{\bar{P}_i}$ of the contained skills or subprograms acting as prior $\tilde{\boldsymbol{\theta}}^{\bar{P}}$ for the containing shadow program. Figure 2.11 (right, bottom) illustrates the use of such a *neural adapter* at the task level. The stacked GRU architecture described in Section 2.4.1 is used. The overall posterior $\boldsymbol{\theta}^{\bar{P}}$ has success and EOS tokens which reflect the program-level semantics: The neural adapter sets the EOS token to 1 when the motion ends, and adapts the success token to reflect the success semantics of the search overall, rather than the constituent contact motions. The dependence of the robot behavior on the environment is implicitly represented by the neural adapter and, by way of its forward pass, reflected in differentiable computations which form part of the DCG. Due to the infinitely composable nature of shadow programs, neural adapters can be inserted at any level of the program hierarchy. As the neural components of shadow skills do at the skill level, neural adapters give computational meaning to the notion of "differentiating through robot behavior" at the task level.

Task-level neural adapter ▷

Forward pass The forward pass through a shadow program $\bar{P}$ is summarized in pseudocode in Listing 2.4. For each subprogram $\bar{P}_i$ (`subp`) in $\bar{P}$, a forward pass

```
def forward(p: DCG, x: Tensor, theta_0: Tensor) -> Tensor:
start_state = theta_0
thetas_sub = []
for subp in p:
    x_subp = extract_inputs(subp, x)
    if subp.is_skill():
        theta_sub = forward_skill(subp, x_subp, start_state)
    else:
        theta_sub = forward(subp, x_subp, start_state)
    start_state = theta_sub[-1]
    thetas_sub.append(theta_sub)

theta_prior = concatenate(thetas_sub)

if p.has_dnn():
    x, theta_prior = normalize(x, theta_prior)
    theta = p.dnn(x, theta_prior)
    theta = denormalize(theta)
else:
    theta = theta_prior
return theta

```

Listing 2.4: Forward pass through a differentiable shadow program.

is performed and the resulting trajectories $\boldsymbol{\theta}^{\bar{P}_i}$ (`theta_sub`) are concatenated to form the prior trajectory $\tilde{\boldsymbol{\theta}}^{\bar{P}}$ (`theta_prior`). If $\bar{P}$ has a neural adaptation layer, it is evaluated on normalized program inputs and $\tilde{\boldsymbol{\theta}}^{\bar{P}}$, and the posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$ (`theta`) is returned. Note that due to the composite structure of $\bar{P}$, the forward pass is identical at every hierarchy level and is implemented by recursive calls to `forward`. At the skill level, `forward_skill` is used (see Listing 2.2).

2.4.5 Discussion

The presented shadow program architecture permits the nesting of shadow subprograms and skills to arbitrary depths, as well as the construction of shadow programs for source programs with non-sequential control flow. Neural adapters permit the learning of execution semantics at the program level. Given a source program, the corresponding differentiable shadow program can be automatically constructed by traversing the source program and instantiating the corresponding DCG. The modular shadow program architecture permits the development of compact parsers and generators for automatic shadow program instantiation. In the context of this work, a parser-generator-toolchain was developed to automatically create differentiable shadow programs for ARTM source programs, as well as for the automatic mapping between ARTM and shadow program (input) parameters, intermediate states and trajectories. Similar toolchains can be developed for any source program representation that meets the criteria outlined in Section 2.3.

Both at the (shadow) skill and program level, DNNs are used to map prior to posterior trajectories. The stacked GRU architecture is trained using supervised learning on a sequence-to-sequence translation problem (see Equation 2.1). While the used architecture permits the learning of complex, long trajectories at the primitive (see Experiment 3.1.3.2) and composite level (see Experiment 3.2.3.3), learning motions with dynamic sampling characteristics remains a challenge. The proposed architecture implicitly assumes label trajectories to be sampled at a fixed interval, and that posterior and prior trajectories share roughly the same dynamics. Sequence-to-sequence learning from heterogeneously sampled datasets may be addressed by incorporating Dynamic Time Warping (Salvador and Chan, 2007) or similar techniques. Moreover, the shadow program architecture requires differentiable priors to be available at the skill level. In this work, linear, circular and spiral Cartesian differentiable motion planners have been developed, and Section 2.5 introduces a differentiable $\mathcal{C}$-space planner. For very complex motions, it may be preferable to directly bootstrap the posterior trajectory using e.g. a generative neural network (Garza et al., 2024). Finally, the state space of NRP is limited to poses, joint configurations, end-effector wrenches and the gripper state, effectively constraining NRP to tactile manipulation. Kienle et al. (2024) extend the NRP state space to images of the environment, enabling the application of neurosymbolic robot programming with NRPs to visuotactile manipulation problems.

2.5 Differentiable Motion Planning

Flexible manipulation tasks typically require motions to be collision-free: When placing glasses into a kitchen cupboard, for example, collisions with other objects in the cupboard and the cupboard itself must be avoided. When working in close proximity with humans, collision-freeness is a safety-critical requirement.

Collision-free motion planning is a long-standing topic of research in robotics. Planners can be broadly divided into sampling-based and trajectory optimization approaches. Sampling-based planners such as Rapidly Exploring Random Trees (LaValle, 1998), Probabilistic Roadmaps (Kavraki et al., 1996) or Expansive Space Trees (Barraquand and Latombe, 1991) generate collision-free joint trajectories by repeatedly sampling random configurations in $\mathcal{C}$-space, performing collision checking, and connecting collision-free configurations to form a path. Sampling-based planners can quickly explore large regions of $\mathcal{C}$-space, making them well-suited for complex environments with high-dimensional configuration spaces (Elbanhawi and Simic, 2014). Trajectory optimization-based planners, on the other hand, find a collision-free path by optimizing a cost function that encodes constraints such as collision avoidance, smoothness, and time efficiency. Given an initial guess, the candidate trajectory is iteratively refined to minimize the cost function while satisfying the constraints. By optimizing over the entire trajectory, rather than sampling configurations individually, trajectory optimization-based planners can find solutions that are more efficient and better satisfy the constraints. However, they may be less robust to changes in the environment and can require more computational resources than sampling-based planners.

Given the wide range of available planners, collision-free planning has become an integral part of robot programming, and several skill-based robot programming frameworks support collision-free planned skills. The ArtiMinds Robot Programming Suite (RPS) (Schmidt-Rohr et al., 2013), an integrated development environment (IDE) for the ARTM program representation, provides high-level skills for grasping, insertion or ungrasping which comprise collision-free approach or depart motions. To represent such skills and enable their optimization subject to collision-freeness and other motion-level constraints (see Section 3.3), NRP shadow programs must be endowed with the capacity to generate collision-free trajectories while ensuring differentiability through the planner.

◁ *DGPMP2-ND*

This section introduces DGPMP2-ND, a differentiable collision-free motion planner that generates trajectories that adhere to motion constraints such as collision-freeness, smoothness, joint limits and accuracy at key points on the trajectory. DGPMP2-ND extends and modifies Differentiable Gaussian Process Motion Planning (DGPMP2) (Bhardwaj et al., 2020) by implementing differentiable collision checking for three-dimensional collision worlds and N-DoF serial kinematics, adding joint limit constraints, and incorporating a factor that rewards similarity to a human demonstration (Alt et al., 2025).

2.5.1 Differentiable Gaussian Process Motion Planning

DGPMP2 is a trajectory optimization-based planner that casts planning as maximum a posteriori (MAP) inference on a differentiable factor graph (Bhardwaj et al., 2020). It is based on Gaussian Process Motion Planning 2 (GPMP2) (Mukadam et al., 2018), which represents continuous-time trajectories $\boldsymbol{\theta}(t) : t \rightarrow \mathcal{S}$ as a sample from a Gaussian process (Mukadam et al., 2016). Motion planning is cast as the minimization of a cost functional $\mathcal{F}[\boldsymbol{\theta}(t)]$ subject to

$$\mathcal{G}_i[\boldsymbol{\theta}(t)] \leq 0, i = 1, ..., m_{\text{ineq}} \tag{2.3}$$

$$\mathcal{H}_i[\boldsymbol{\theta}(t)] = 0, i = 1, ..., m_{\text{eq}}, \tag{2.4}$$

with inequality constraints $\mathcal{G}_i$, such as joint limits, and equality constraints $\mathcal{H}_i$, such as trajectory start and end states. In typical applications, $\mathcal{F}[\boldsymbol{\theta}(t)]$ encodes smoothness and a measure of collision-freeness. Trajectories are modeled as samples from a Gaussian process (GP):

$$\boldsymbol{\theta}(t) \sim \mathcal{GP}(\boldsymbol{\mu}(t), \mathcal{K}(t, t')) \tag{2.5}$$

with mean $\boldsymbol{\mu}(t)$ and covariance $\mathcal{K}(t, t')$. In this section, $\boldsymbol{\theta}(t)$ denotes a trajectory as a function of a continuous-valued time index t. Evaluating $\boldsymbol{\theta}(t)$ for equidistantly spaced values of t yields the discrete-time, vectorized representation of $\boldsymbol{\theta}$ introduced in Section 2.3.1. In the context of DGPMP2, $\boldsymbol{\theta}$ denotes the *support states* that parameterize $\boldsymbol{\theta}(t)$ (Mukadam et al., 2018). Given a sampled trajectory $\boldsymbol{\theta}$, the GP then “defines a prior on the space of trajectories” (Mukadam et al., 2018), with probability density

$$p(\boldsymbol{\theta}) \propto \exp\left\{-\frac{1}{2}||\boldsymbol{\theta} - \boldsymbol{\mu}||^2_{\mathcal{K}}\right\}, \tag{2.6}$$

where $||\cdot||^2_{\mathcal{K}}$ denotes the Mahalanobis distance with respect to $\mathcal{K}$. The advantage of such a Gaussian prior is that $\boldsymbol{\theta}(t)$ can be sampled and re-sampled at varied temporal resolutions. I refer to Mukadam et al. (2018) for a description of the construction of the GP.

DGPMP2 solves the motion planning problem by MAP inference. It defines a series of binary events e_i at times t_i, which denote whether or not some condition holds: $e_0 = 1$, for example, may signify that $\boldsymbol{\theta}(t_0)$ is in collision, and $e_i = 0$ for all i that the trajectory is collision-free. Given such collision events $\boldsymbol{e}$, the posterior density of $\boldsymbol{\theta}$ is

$$p(\boldsymbol{\theta}|\boldsymbol{e}) = \frac{p(\boldsymbol{\theta})p(\boldsymbol{e}|\boldsymbol{\theta})}{p(\boldsymbol{e})} \propto p(\boldsymbol{\theta})p(\boldsymbol{e}|\boldsymbol{\theta}) \tag{2.7}$$

where $p(\boldsymbol{e}|\boldsymbol{\theta})$ is the likelihood that $\boldsymbol{\theta}$ is collision-free (Mukadam et al., 2018).

A MAP estimator will find a collision-free trajectory $\boldsymbol{\theta}^*$:

$$\begin{aligned} \boldsymbol{\theta}^* &= \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\boldsymbol{e}) \\ &= \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta})l(\boldsymbol{\theta};\boldsymbol{e}) \end{aligned} \tag{2.8}$$

where $l(\boldsymbol{\theta};\boldsymbol{e}) \approx p(\boldsymbol{e}|\boldsymbol{\theta})$ is the likelihood of states $\boldsymbol{\theta}$ given events e_i for each state θ_i in $\boldsymbol{\theta}$. It is defined by the distribution

$$l(\boldsymbol{\theta};\boldsymbol{e}) = \exp\left\{-\frac{1}{2}||\boldsymbol{h}(\boldsymbol{\theta})||^2_{\boldsymbol{\Sigma}_{\text{obs}}}\right\}, \tag{2.9}$$

where $\boldsymbol{h}(\boldsymbol{\theta})$ is the obstacle cost for each state on $\boldsymbol{\theta}$, and the covariance $\boldsymbol{\Sigma}_{\text{obs}}$ is a tunable hyperparameter defining the scale and direction of the collision error. To perform iterative MAP optimization, $\boldsymbol{\theta}$ and $\boldsymbol{e}$ are expressed as a factor graph. The trajectory prior (2.6) can be factored as

$$p(\boldsymbol{\theta}) \propto f_0^p(\boldsymbol{\theta}_0)f_N^p(\boldsymbol{\theta}_N)\prod_{i=0}^{N-1} f_i^{gp}(\boldsymbol{\theta}_i, \boldsymbol{\theta}_{i+1}) \tag{2.10}$$

with start and goal state priors

$$f_i^p(\boldsymbol{\theta}_i) = \exp\left\{-\frac{1}{2}||\boldsymbol{\theta}_i - \boldsymbol{\mu}_i||^2_{\boldsymbol{\mathcal{K}_i}}\right\}, i = 0 \text{ or } N, \tag{2.11}$$

for start and end states $\boldsymbol{\mu}_0$ and $\boldsymbol{\mu}_N$ and respective covariances $\mathcal{K}_0$ and $\mathcal{K}_N$. The GP factor $f_i^{gp}(\boldsymbol{\theta}_i, \boldsymbol{\theta}_{i+1})$ places a GP prior on successive states (see Mukadam et al. (2018) for details). MAP inference (2.8) corresponds to the minimization problem

$$\begin{aligned}\boldsymbol{\theta}^* &= \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\boldsymbol{e})\\ &= \arg\min_{\boldsymbol{\theta}} -\log(p(\boldsymbol{\theta})l(\boldsymbol{\theta};\boldsymbol{e}))\\ &= \arg\min_{\boldsymbol{\theta}} \frac{1}{2}||\boldsymbol{\theta}-\boldsymbol{\mu}||^2_{\mathcal{K}} + \frac{1}{2}||\boldsymbol{h}(\boldsymbol{\theta})||^2_{\Sigma}.\end{aligned} \tag{2.12}$$

At iteration j, the cost function $\boldsymbol{h}$ is linearized around the current trajectory estimate $\boldsymbol{\theta}^j$ using a Taylor expansion $\boldsymbol{h}(\boldsymbol{\theta}) = \boldsymbol{h}(\boldsymbol{\theta}^j) + \boldsymbol{H}\delta\boldsymbol{\theta}$ with $\boldsymbol{H} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{\theta}}\big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^j}$. A Gauss-Newton optimizer then solves the linear system

$$(\mathcal{K}^{-1} + \boldsymbol{H}^T\Sigma^{-1}\boldsymbol{H})\delta\boldsymbol{\theta} = -\mathcal{K}^{-1}(\boldsymbol{\theta}^j - \boldsymbol{\mu}) - \boldsymbol{H}^T\Sigma^{-1}\boldsymbol{h}(\boldsymbol{\theta}^j) \tag{2.13}$$

to find the update $\delta\boldsymbol{\theta}$ (Bhardwaj et al., 2020):

$$\boldsymbol{\theta}^{j+1} = \boldsymbol{\theta}^j + \delta\boldsymbol{\theta} \tag{2.14}$$

The reformulation of the planning problem as nonlinear least squares and the sparse structure of the system permits the use of highly efficient solvers. GPMP2 outperforms other trajectory optimization-based solvers such as CHOMP (Ratliff et al., 2009) or TrajOpt (Schulman et al., 2013), while allowing collision checking and execution at different levels of resolution via GP interpolation (Mukadam et al., 2018). Like all motion planners based on trajectory optimization, GPMP2 can get stuck in local minima, which can be mitigated by parallel optimization from different initializations.

DGPMP2 is a differentiable implementation of GPMP2 based on the observation that the operations performed by GPMP2 for motion planning form a DCG (Bhardwaj et al., 2020). Alongside the planning module, which corresponds to the original GPMP2, the authors introduce a learnable module which generates a set of planner parameters, such as the GP covariance $\mathcal{K}$ or the collision likelihood covariance Σ, by way of a convolutional neural network (CNN). The ability to set planner parameters from a learned model permits the learning of planner parameterization from past experience, mitigating the need for manual hyperparameter tuning for different applications.

2.5.2 DGPMP2-ND: Differentiable Gaussian Process Motion Planning for N-DoF Manipulators

The differentiable motion planner presented in this work builds on the planning module of DGPMP2. It generalizes DGPMP2 to N-DoF manipulators and 3D collision worlds by integrating differentiable kinematics into the planning DCG.

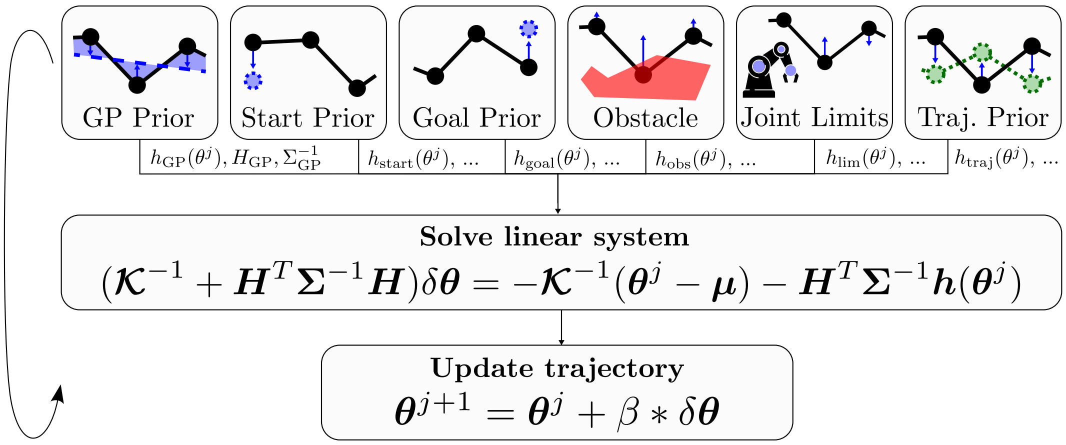

GP Prior
Start Prior
Goal Prior
Obstacle
Joint Limits
Traj. Prior
$h_{\mathrm{GP}}(\theta^j), H_{\mathrm{GP}}, \Sigma_{\mathrm{GP}}^{-1}$
$h_{\mathrm{start}}(\theta^j)$, ...
$h_{\mathrm{goal}}(\theta^j)$, ...
$h_{\mathrm{obs}}(\theta^j)$, ...
$h_{\mathrm{lim}}(\theta^j)$, ...
$h_{\mathrm{traj}}(\theta^j)$, ...
Solve linear system
$(\boldsymbol{\mathcal{K}}^{-1} + \boldsymbol{H}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{H})\delta\boldsymbol{\theta} = -\boldsymbol{\mathcal{K}}^{-1}(\boldsymbol{\theta}^j - \boldsymbol{\mu}) - \boldsymbol{H}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{h}(\boldsymbol{\theta}^j)$
Update trajectory
$\boldsymbol{\theta}^{j+1} = \boldsymbol{\theta}^j + \beta * \delta\boldsymbol{\theta}$

Figure 2.12: DGPMP2-ND plans collision-free trajectories by iterative optimization. The planning loop is differentiable end-to-end (Alt et al., 2025).

Moreover, it permits planning in both $\mathcal{C}$-space and Cartesian workspace, permitting to simultaneously respect joint limit and end-effector pose constraints.

DGPMP2-ND is illustrated in Figure 2.12. As in DGPMP2, the planning problem is cast as MAP inference over a factor graph, which is solved by iterative Gauss-Newton over the linear system (2.13). At each iteration j, each factor computes an *error* $\boldsymbol{h}(\boldsymbol{\theta})$ indicating the magnitude of the update to the current trajectory estimate $\boldsymbol{\theta}^j$, a *Jacobian* $\boldsymbol{H}$ indicating the direction of the update and a *covariance* $\boldsymbol{\Sigma}$ indicating the relative weight of each factor. Section 2.5.2.6 provides an in-depth explanation of the planning loop.

Beyond the GP, start and goal priors introduced by Mukadam et al. (2018), DGPMP2-ND introduces a *joint limits factor*, which penalizes states that violate the joint limits of the robot (see Section 2.5.2.4), as well as a *demonstration prior* (see Section 2.5.2.5) which allows for trajectory optimization with respect to a reference trajectory, such as a human demonstration. DGPMP2-ND introduces a differentiable obstacle factor, which permits planning for N-DoF kinematics with respect to complex 3D collision worlds (see Section 2.5.2.2).

2.5.2.1 Trajectory Representation

The definition of the trajectory $\boldsymbol{\theta}$ employed by DGPMP2-ND deviates from the trajectory definition provided in Section 2.4.1.1 and employed in the overall context of NRPs. Instead of an augmented state representation, which may contain Cartesian end-effector poses, $\mathcal{C}$-space states or end-effector wrenches, $\boldsymbol{\theta}$ in the context of DGPMP2-ND strictly contains states and velocities in $\mathcal{C}$-space, echoing the literature (Mukadam et al., 2016). $\boldsymbol{\theta}$ is then a tensor of size $M \times 2N$, where M is the number of states on the trajectory and N the number of DoF of the manipulator. Moreover, in the context of DGPMP2-ND, states on the trajectory are not assumed to be sampled at equidistant points in time. Rather, the dynamics are explicitly represented by including joint velocities as part of the state. DGPMP2-ND assumes the same constant-velocity model as DGPMP2 to facilitate the computation of the GP prior (see Mukadam et al. (2018) for details).

2.5.2.2 Obstacle Factor

The proposed obstacle factor is a differentiable version of the original GPMP2, which, in turn, is based on the obstacle cost function of CHOMP (Ratliff et al., 2009). The collision geometry of the robot is represented as a set of geometric primitives for efficient computation. While GPMP2 and CHOMP represent a robot link as a set of spheres, DGPMP2-ND represents it as a cylinder, oriented along the link's principal axis, which fully contains the link's collision geometry. The cylinders of adjacent links partially overlap (see Figure 2.13). Unlike the spherical representation used by DGPMP2, the cylindrical representation covers the manipulator without gaps between primitives, avoiding spurious collisions, while requiring fewer primitives in total.

Before planning, a signed distance field (SDF) of the environment is precomputed, in which each voxel contains the signed distance of the voxel center to the closest obstacle. At each planning iteration j and for every state $\boldsymbol{\theta}_i^j$ on the current trajectory estimate, the Cartesian poses and Jacobians of all links are computed via differentiable FK (see Section 2.4.4.1). For each link and each timestep, the SDF voxel is identified which intersects with the link's cylinder approximation and has the smallest distance to the closest collision object. The sum of the direction vectors to the neighboring 26 voxels, each scaled with the signed distance to the next collision object contained in that voxel, computes the collision error vector $\boldsymbol{v}_{ik}$ for the kth robot link at the ith point on the trajectory (see Figure 2.14). The collision error $\boldsymbol{h}_{\text{obs}}(\boldsymbol{\theta})$ is an $M \times L$ tensor of the hinge loss over the error vector magnitudes $\mathcal{L}_{\text{hinge}}(|\boldsymbol{v}_{ik}|), 1 \leq i \leq M, 1 \leq k \leq L$, where L is the number of links of the robot. The collision Jacobian $\boldsymbol{H}_{\text{obs}}$ is a tensor of shape $M \times L \times N$, where each entry $\boldsymbol{H}_{\text{obs},ikn}$ is proportional to the amount by which the nth joint state should be changed to maximize the collision error for the kth link at the ith point on the trajectory. The obstacle covariance $\boldsymbol{\Sigma}_{\text{obs}}$ is the $M \times L \times L$ tensor of isotropic matrices proposed in the original formulation of GPMP2 (Mukadam et al., 2018)

$$\boldsymbol{\Sigma}_{\text{obs}} = \sigma_{\text{obs}}^2 \boldsymbol{I}, \tag{2.15}$$

where σ_{obs} is the weight of the obstacle factor.

2.5.2.3 Cartesian Start and Goal Priors

In addition to the $\mathcal{C}$-space start and goal priors introduced in GPMP2 (Mukadam et al., 2018) and provided by the original DGPMP2 implementation (Bhardwaj et al., 2020), DGPMP2-ND provides Cartesian *start* and *goal prior factors*, which penalize deviations of the end-effector poses at the first or last point of the trajectory $\boldsymbol{\theta}$ from

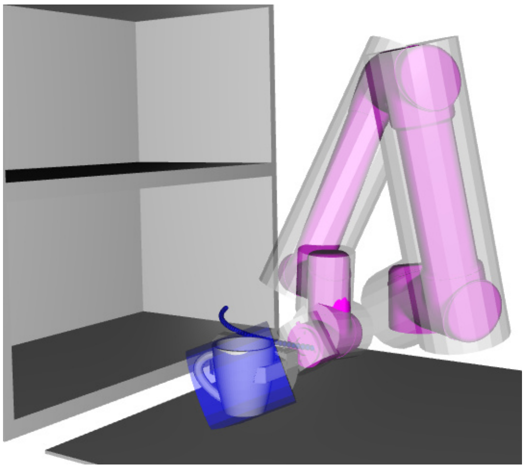


Figure 2.13: Collision model by cylindrical approximation for a UR5 arm.

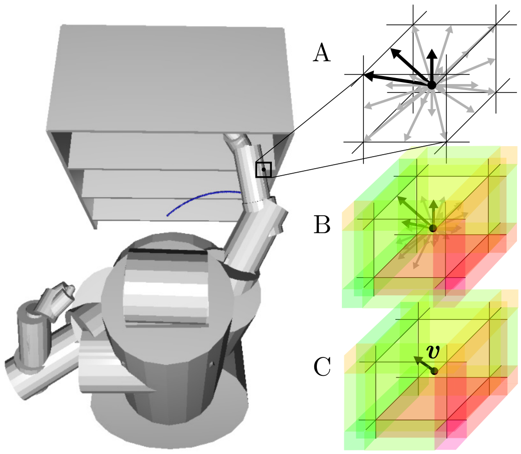


Figure 2.14: Computation of the DGPMP2-ND collision error vector $\boldsymbol{v}$ for a robot link as the sum of the direction vectors to the 26 surrounding voxels (A), rescaled by their respective collision distances (B).

given Cartesian start and end poses q_1 and q_M. They typically represent constraints on the planned motion imposed by the task, such as picking up a cup at a detected position and placing it at a destination. The Cartesian start and goal errors $\boldsymbol{h}_{\text{start}}(\boldsymbol{\theta})$ and $\boldsymbol{h}_{\text{goal}}(\boldsymbol{\theta})$ are the difference between q_1 or q_M and the differentiable FK solutions of the first and last states $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_M$, respectively, and zero for all other states on the trajectory. The Jacobians $\boldsymbol{H}_{\text{start}}(\boldsymbol{\theta})$ and $\boldsymbol{H}_{\text{goal}}(\boldsymbol{\theta})$ is the $M \times 6 \times N$ tensor of the Jacobians computed by differentiable FK for $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_M$, respectively, and the zero matrix for all other states on $\boldsymbol{\theta}$. The covariances $\boldsymbol{\Sigma}_{\text{start}} = \sigma^2_{\text{start}}\boldsymbol{I}$ and $\boldsymbol{\Sigma}_{\text{goal}} = \sigma^2_{\text{goal}}\boldsymbol{I}$ are the same isotropic matrices as (2.15) with start and goal factor weights σ_{start} and σ_{goal}, respectively (see Section 2.5.2.6).

2.5.2.4 Joint Limit Factor

To ensure that the planned trajectory $\boldsymbol{\theta}^*$ respects the joint limits of the robot, a *joint limit factor* is introduced. The error $\boldsymbol{h}_{\text{lim}}(\boldsymbol{\theta})$ penalizes states on $\boldsymbol{\theta}$ which exceed the upper and lower joint limits θ^+_{lim} and θ^-_{lim}, respectively:

$$\boldsymbol{h}_{\text{lim}}(\boldsymbol{\theta}) = \begin{cases} \boldsymbol{\theta} - \theta^+_{\text{lim}} & \text{if} \quad \boldsymbol{\theta} > \theta^+_{\text{lim}} \\ -\theta^-_{\text{lim}} - \boldsymbol{\theta} & \text{if} \quad \boldsymbol{\theta} < -\theta^-_{\text{lim}} \\ 0 & \text{otherwise.} \end{cases} \tag{2.16}$$

The joint limit Jacobian $\boldsymbol{H}_{\text{lim}}(\boldsymbol{\theta})$ is a $M \times 6 \times N$ tensor of matrices, where the ith matrix corresponds to the Jacobian for the ith state on $\boldsymbol{\theta}$. The Jacobians are diagonal matrices with entries -1 for joints exceeding θ^+_{lim}, 1 for joints falling below

$-\theta^{-}_{\text{lim}}$, and 0 otherwise. The covariance $\boldsymbol{\Sigma}_{\text{lim}} = \sigma^2_{\text{lim}}\boldsymbol{I}$ is the same isotropic matrix as (2.15) with limit factor weight σ_{lim} (see Section 2.5.2.6).

2.5.2.5 Demonstration Prior

Most planners benefit from of human guidance during the planning process. Trajectory optimization-based planners such as DGPMP2 exhibit considerably faster convergence if the initial trajectory $\boldsymbol{\theta}^0$ is close to the optimal trajectory $\boldsymbol{\theta}^*$. For many planning problems, however, a good initial trajectory is not known beforehand. For such tasks, planners may provide mechanisms for human experts to demonstrate actions, and leverage that demonstration to speed up planning or increase the quality of planned motions (Billard et al., 2008; Koert et al., 2016). To effectively incorporate human feedback into the planning process, DGPMP2-ND supports an optional *demonstration prior factor*. For every state $\boldsymbol{\theta}_i$ on trajectory $\boldsymbol{\theta}$, the corresponding Cartesian end-effector pose $\boldsymbol{p}_i$ and Jacobian $\boldsymbol{H}_i$ are computed via differentiable FK (see Section 2.4.4.1). The demonstration prior error $\boldsymbol{h}_{\text{traj}}(\boldsymbol{\theta})$ is the pointwise difference between the current Cartesian planned trajectory $\boldsymbol{p}$ and a Cartesian reference trajectory $\boldsymbol{q}$. The demonstration prior Jacobian $\boldsymbol{H}_{\text{traj}}(\boldsymbol{\theta})$ is the $M \times 6 \times N$ tensor of the Jacobians $\boldsymbol{H}_i$ along the trajectory. The covariance $\boldsymbol{\Sigma}_{\text{traj}} = \sigma^2_{\text{traj}}\boldsymbol{I}$ is the same isotropic matrix as (2.15) with demonstration prior factor weight σ_{traj} (see Section 2.5.2.6).

The computation of the pointwise difference requires the demonstration $\boldsymbol{q}$ and the planned trajectory $\boldsymbol{\theta}$ to be of the same length. The velocity of the human demonstration may be relevant to the task; when moving a vessel filled with liquid, such as a cup, the dynamics of the demonstration may be central to the task success. For this reason, rather than resampling $\boldsymbol{q}$ at the velocity of $\boldsymbol{\theta}$, the velocity and (constant) acceleration of $\boldsymbol{\theta}^0$ (the initial trajectory at the start of planning) are set so that $\boldsymbol{q}$ and $\boldsymbol{\theta}^0$ have the same number of points and states, respectively. As the number of states on $\boldsymbol{\theta}$ does not change during planning, $\boldsymbol{q}$ and the current planned trajectory $\boldsymbol{\theta}^j$ can be compared at every iteration j.

Note that multiple demonstrations can be used simultaneously to condition the planner, as additional instances of the prior demonstration factor can be added to the factor graph with different weights σ_{traj}. Also note that neither the end-effector pose at the start or the end of the initial trajectory $\boldsymbol{\theta}^0$, nor any point in between, is required to lie on $\boldsymbol{q}$. $\boldsymbol{q}$ is not required to be collision-free or smooth, either. The start and end poses, as well as the collision-freeness and smoothness of the trajectory, are enforced by the start and goal priors, the obstacle factor and the GP prior, respectively. Using the demonstration factor weight σ_{traj}, the intended degree of similarity to the human demonstration can be adjusted depending on the task. For DGPMP2-ND to benefit most from human demonstrations, the demonstration should start and end in relative proximity to the initial trajectory $\boldsymbol{\theta}^0$.

2.5.2.6 Planning Loop

The planning loop is illustrated in Figure 2.12. At each iteration, the linear system

$$(\mathcal{K}^{-1} + \boldsymbol{H}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{H})\delta\boldsymbol{\theta} = -\mathcal{K}^{-1}(\boldsymbol{\theta}^i - \boldsymbol{\mu}) - \boldsymbol{H}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{h}(\boldsymbol{\theta}^i) \tag{2.17}$$

is solved for $\delta\boldsymbol{\theta}$ with the combined Jacobian $\boldsymbol{H}$, combined inverse covariance $\boldsymbol{\Sigma}^{-1}$, inverse GP kernel matrix $\mathcal{K}^{-1}$ and combined error function $\boldsymbol{h}(\boldsymbol{\theta}^j)$. $\boldsymbol{H}$ and $\boldsymbol{\Sigma}^{-1}$ are formed by concatenating the individual factors' Jacobian and inverse covariance matrices along the row axis. $\boldsymbol{\theta}^j$ is updated along the direction of $\delta\boldsymbol{\theta}$ with update rate β:

$$\boldsymbol{\theta}^{j+1} = \boldsymbol{\theta}^j + \beta * \delta\boldsymbol{\theta} \tag{2.18}$$

Forward pass Including an iterative optimizer in the forward pass of shadow programs incurs a considerable computational overhead. The termination criterion for iterative first-order trajectory optimization should minimize the number of iterations while maximizing the likelihood of planning success. A necessary condition for planning success is collision-freeness and adherence to joint limits; moreover, the start pose error h_{start} should be as low as possible to ensure smooth integration into the overall posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$, and a low target pose error h_{goal} indicates successful completion of the motion. The *critical error* at DGPMP2-ND iteration j is then defined as

$$h_{\text{crit}}(\boldsymbol{\theta}^j) = h_{\text{start}}(\boldsymbol{\theta}^j) + h_{\text{goal}}(\boldsymbol{\theta}^j) + h_{\text{obs}}(\boldsymbol{\theta}^j) + h_{\text{lim}}(\boldsymbol{\theta}^j), \tag{2.19}$$

comprising all error terms except the GP and demonstration priors. When the critical error is zero, the planned trajectory is feasible, as it is collision-free, reachable, and integrates seamlessly into the overall posterior trajectory. To ensure fast convergence while ensuring minimization of h_{crit}, a patience mechanism is adopted, whereby the number of total inner-loop DGPMP2-ND iterations is determined dynamically. Listing 2.5 details the forward pass through the DGPMP2-ND optimizer in pseudocode. Two additional hyperparameters are introduced, which jointly determine the termination condition. The *patience* φ determines the number of remaining DGPMP2-ND iterations after the critical error stops decreasing; DGPMP2-ND terminates if h_{crit} does not decrease for φ iterations. The *threshold* ε denotes the tolerated total error h: DGPMP2-ND terminates if $h < \varepsilon$. DGPMP2-ND always terminates after a given maximum number of iterations. Patience-based early termination with respect to the critical error avoids wasted computation in situations in which the critical error cannot be reduced, e.g. if the start pose is in collision. Such situations typically result from suboptimally parameterized upstream skills.

2.5.3 Discussion

With DGPMP2-ND, NRPs gain the ability to represent collision-free motion skills, and, more generally, skills that are defined by $\mathcal{C}$-space and Cartesian constraint

```python
def forward(theta_init: Tensor, beta: float, patience: float, threshold: float,
     max_iters: float) -> Tensor:
    j = 0
    theta_j = theta_init
    theta = theta_init
    min_error = inf
    error_patience = 0

    while j < max_iters:
        h, H, K, Sigma = construct_linear_system(theta_j)

        # Compute total error and critical error
        last_error = sum(h)
        h_gp, h_start, h_goal, h_obs, h_lim, h_traj = h
        critical_error = sum(h_start, h_goal, h_obs, h_lim)

        # Decay learning rate if critical error zero
        if critical_error == 0:
            beta *= 0.1

        error_patience += 1

        # Critical error decreased
        if critical_error <= min_error:
            min_error = critical_error
            theta = theta_j         # Save new candidate (optimal) trajectory
            error_patience = 0      # Reset patience

            if last_error < threshold:
                break               # Optimizer converged, terminate

      # Critical error did not decrease for too many iterations
        if error_patience > patience:
            break                   # Patience exceeded, abort

        # Update current trajectory
        delta_theta = solve_linear_system(h, H, K, Sigma)
        theta_j += beta * delta_theta
        j += 1

    return theta
```

Listing 2.5: The DGPMP2-ND optimization loop.

minimization. In industrial and service applications, such skills are typically used for approach, depart and transfer motions. The original intent motivating the development of DGPMP2-ND was the ability to differentiate through such transfer motions to optimize downstream grasp or insertion motions (see e.g. Experiment 3.3.2.2). However, DGPMP2-ND proves to be a flexible planner in its own right, permitting the joint first-order optimization of low-level motion trajectories and high-level robot program parameters by gradient descent over the shadow DCG. Program optimization with DGPMP2-ND is covered in greater detail in Section 3.3. An evaluation of DGPMP2-ND as a standalone planner as well as a differentiable prior in the context of NRPs is provided in Section 3.3.2. An open-source implementation of DGPMP2-ND is available.[3]

2.6 Related Work

The conceptualization and development of suitable representations for expressing robot programs has been a long-standing research topic in robotics. NRPs synthesize several different traditions of robot programming research into a novel program representation, drawing from textual, skill-based, neural and hybrid representations.

2.6.1 Textual Program Representations

Textual robot programming has been the predominant mode of programming robots since the advent of the first commercial robot arms in the 1960. MHI, the first dedicated robot programming language, was developed in the early 1960s (Ernst, 1962) to support programming as bidirectional communication between the human programmer and the machine (see Figure 1.1). While its main objective was to be a “convenient programming language that permits the [...] specification of that which the hand is to do”, a secondary objective was to enable human programmers to gain “familiarity with the operation of the [robotic] system [...] by writing some simple programs for [it]” (Ernst, 1962). MHI supported primitives for perception and action, such as a `move` command which could be parameterized by e.g. a direction vector and a velocity. Primitives are combined to form complex programs by way of control flow instructions (`until`, `ifgoto`, `ifcontinue`) etc. The program representation established by MHI – “[s]tatements linked by conditional transfers determine the actions to be executed” (Ernst, 1962) – laid the foundations for all subsequent generations of textual robot program representations. Since MHI, many robot programming languages such as WAVE (Paul, 1977), VAL (Shimano, 1979) or AML (Taylor et al., 1982), and more recently the KUKA Robot Language (KRL) (Mühe et al., 2010), FANUC KAREL language (FANUC America Corporation, 2014) and URScript (Universal Robots, 2018b) have been proposed, most of which

---

[3] https://github.com/benjaminalt/dgpmp2-nd

combine a modified syntax and semantics of a general-purpose programming language such as BASIC, FORTRAN, Lisp or Python with robotics-specific extensions and a hardware-specific robot control stack (Poole, 1989).

Textual program representations are designed to be written and read by human programmers, with the intent of compactly encoding the information required to describe the robot's behavior (Ernst, 1962). Moreover, textual representations have a high degree of *expressivity* (Ajaykumar et al., 2021): Modern robot programming languages have a wide range of mathematical, logical and control flow primitives, support for functional abstraction and modular composition, as well as interfaces to general-purpose programming languages such as Python or C++, enabling them, at least in principle, to express a wide range of robot behaviors. Textual robot programs are *explicit*: A human reader or the program interpreter of a robot controller require little additional computation to extract the represented behavior from the representation.

2.6.1.1 Industrial Robot Programming Languages

The fact that textual robot programs explicitly represent robot behavior has caused textual programs to remain the predominant representations used in industrial robotics today. It is crucial to note that most industrial robot programming systems employ *hybrid* representations: Programs are displayed e.g. as graphical icons on teach pendants or in robot programming software provided by manufacturers, which can be adapted via drag-and-drop or parameterized via teach-in (Villani et al., 2018). The PolyScope programming interface for Universal Robots, that permits the programmer to create and parameterize URScript programs via a graphical user interface (GUI) on a teach pendant (Weintrop et al., 2018), is an example. One of the reasons for the ubiquity of textual program representations in industrial robotics is the fact that large manufacturing enterprises such as automotive original equipment manufacturers (OEMs) impose strict group-wide standards on their manufacturing processes, including robot programs, to ensure compatibility and safety (Hirzle et al., 2008; Akcay, 2016). Industrial robot programming standards are largely expressed in terms of textual robot program representations such as the VKRC dialect of the KRL programming language, mandating or prohibiting the use of particular language features, imposing coding guidelines and placing strict requirements on the robot controller interpreting the language (Weißmann et al., 2011). The aim of such restrictions is to ensure that robot behavior is well-understood by human programmers and workcell maintainers, that given safety and behavioral guarantees are met, and that robot programs are interoperable with the plant-wide or group-wide manufacturing execution system (MES) and other digital infrastructure. Typically, compliance of robot programming tools and generated robot programs is performed automatically at the syntactic level, e.g. by the robot controller which rejects programs syntactically incompatible with the standard, or manually by human compliance and safety experts. Industrial robotics standards favor and often require textual robot program representations because they are *documents* which can be read, versioned, archived and retrieved.

2.6.2 Movement Primitives

The representation of robot programs as text provides a straightforward way to define programs as control flow graphs of atomic primitives, which are typically implemented as low-level point-to-point or otherwise constrained motions directly on the robot controller. In textual robot programming languages, the primitives themselves are represented as symbols, whose semantics are assumed as “given”. For this reason, approaches for learning and synthesis of textual robot programs typically operate at the level of control flow and do not learn or adapt low-level motion primitives (Patton et al., 2024a; Liang et al., 2023; Chen et al., 2023; ElMaraghy and Rondeau, 1992). In contrast, Movement Primitives (MPs) represent motions as parameterized functions such as dynamic systems described by differential equations (Schaal, 2006; Ratliff et al., 2018) or probability distributions (Paraschos et al., 2013; Huang et al., 2019) whose parameters can be learned via e.g. supervised (Ijspeert et al., 2002; Billard et al., 2008; Akbulut et al., 2021a) or reinforcement learning (Akbulut et al., 2021b; Tosatto et al., 2021; Stulp and Schaal, 2011). MPs lie on the explicit side of the representational spectrum, as they encode robot motion via a parameterized function with a priori known semantics, and are subsymbolic by virtue of representing motion as continuous functions.

2.6.2.1 Dynamic Movement Primitives

A wide variety of MPs have been proposed. DMPs (Ijspeert et al., 2002; Schaal, 2006) represent movements as second-order dynamic systems, such as a parameterized damped spring model with a forcing term (Ijspeert et al., 2013). By choosing suitable values for the damper and spring constants as well as the forcing term, a range of periodic and aperiodic behaviors can be represented. If the forcing term is modeled as a weighted sum of exponential basis functions, the DMP acts as a point attractor and can be used to represent goal-directed actions such as reaching or pushing; for periodic von Mises basis functions, the DMP acts as a limit cycle attractor and can represent periodic motions such as wiping or shaking. The behavior is largely determined by the weights of the basis functions (the DMP *parameters* (Stulp et al., 2013)) as well as the start and goal states of the motion, the time constant and the damper and spring constants (the DMP *metaparameters*). Parameters and metaparameters can be learned to represent given motions using e.g. supervised learning from human demonstrations using locally weighted regression (Schaal and Atkeson, 1998; Ijspeert et al., 2013). A wide range of variants of DMP have been proposed, which use different basis functions, Gaussian Mixture Models (GMMs), GPs or neural networks as forcing terms; Saveriano et al. (2023) provide a comprehensive overview. While the original DMP formulation (Ijspeert et al., 2002) represents single-DoF motions, multiple DMPs can be coupled via a shared phase variable, allowing the representation of multi-DoF trajectories in state spaces with independent dimensions, such as the $\mathcal{C}$-space of a serial manipulator or 3D Cartesian positions (Saveriano et al., 2023). DMP formulations for Cartesian pose trajectories using quaternions (Abu-Dakka et al., 2015) or rotation matrices (Ude et al., 2014) have been proposed.

2.6.2.2 DMP Alternatives

The DMP-based MP representations cited above retain the original DMP formulation of motions as dynamic systems described by second-order differential equations with a forcing term (Ijspeert et al., 2013). Alternative approaches use different underlying representations while realizing the same function – representing low-level robot motions via a parameterized dynamics model (Saveriano et al., 2023). From a representational perspective, it is the MP formulations that cross the boundaries between symbolic and subsymbolic, explicit and implicit representations which merit detailed discussion.

Probabilistic Movement Primitives (ProMPs) (Paraschos et al., 2013) propose to address several shortcomings of DMPs by adding support for sequential and simultaneous composition, variably adjustable speed of motion, training by both RL and human demonstrations, as well as suitability for both deterministic and stochastic environments. ProMPs represent motions as distributions over trajectories, realized as a weighted mixture of Gaussian or von Mises basis functions for discrete and periodic MPs, respectively. The weight vector follows a parameterized probability distribution such that the trajectory distribution defines a hierarchical Bayesian model. This probabilistic representation permits the definition of operators for conditioning on given via-points, goal positions or velocities, as well as sequential combination (blending) and parallel combination (co-activation) (Paraschos et al., 2013).

Probabilistic Dynamic Movement Primitives (ProDMPs) (Li et al., 2023a) conceptually combine DMPs and ProMPs. They propose to generate a set of *position basis functions* offline by numerical integration of the DMP ODE. ProDMPs can closely fit a single demonstration, while supporting co-activation and blending. Li et al. (2023a) propose to extend ProDMPs by a DNN to support conditioning of the MP on additional high-dimensional inputs, such as images of the environment. The DNN predicts the mean and covariance of the weight distribution and is trained to minimize the negative log-likelihood of the ground-truth trajectories, such as human demonstrations.

Unlike ProMP and ProDMP, which model motions as dynamic systems parameterized by the weights of mixtures of basis functions, CNMPs (Seker et al., 2019) model motions as GPs. The mean and variance of the GP are predicted by a neural network given a query input vector, conditioned on prior observations. Task-specific parameters are concatenated to the query vector, making the generated behavior contingent on these parameters. CNMPs can be conditioned on high-dimensional inputs, such as environment images, and trained on human demonstrations or RL (Akbulut et al., 2021b).

ProMPs, ProDMPs and CNMPs are parameterized, motion-level representations of robot behavior. Due to their inclusion of neural components and probabilistic models, they lie on the spectrum between explicit and implicit representations. Motions are represented in part explicitly, as robot behavior is encoded in a priori specified mathematical constructs such as dynamical systems or GPs; they are also partly represented implicitly, as the concrete motion arises by sampling from probability distributions, evaluating neural networks, or a combination of both. All MPs share the common advantages of being learnable from data and affording a degree of interpretability of the represented behavior, as the robot motion is subject to the constraints imposed by the a priori defined mathematical model of motion.

2.6.2.3 From Movement Primitives to Robot Skills

MPs represent robot behavior at the motion level and satisfy the skill definition proposed in Section 2.2.2.1, though their application is rendered challenging by the fact that their parameters do not necessarily reflect task-level semantics: It is unclear, for example, how the parameters of a DMP ought to be adapted so that the represented motion satisfies the intent of the programmer. Stulp et al. (2013) explicitly add such a semantic dimension. They propose a DMP-based MP which accepts task-level parameters, such as the pose of a target object in the environment, and is trained to adapt its behavior at runtime to varying task parameters. Most state-of-the-art MP representations provide mechanisms for sequential combination (blending) and parallel superposition (co-activation) of individual MPs, permitting the representation of complex robot tasks with multiple subtasks, goals and constraints (Paraschos et al., 2013; Li et al., 2023a). Riemannian Motion Policiess (RMPs) (Ratliff et al., 2018) extend a second-order dynamic system representation of motion by a Riemannian metric, which permits to define an addition operator capable of combining movement primitives defined in different spaces such as $\mathcal{C}$-space and Cartesian task space. Cheng et al. (2019) propose the RMPflow motion planner, which sequences and parameterizes RMPs to solve hierarchical tasks while smoothly interpolating between individual motions. MPs remain, however, a skill-level representation: "The aim of MPs is to allow for composing complex robot skills out of elemental movements with a modular control architecture" (Paraschos et al., 2013). They provide a partly explicit representation of skills and underlie several state-of-the-art task-based program representations.

2.6.3 Task-Based Program Representations

The emergence of the task-level robot programming paradigm has given rise to program representations which mirror the compositional and hierarchical structure of tasks (Lozano-Perez, 1983): High-level tasks can be hierarchically decomposed into lower-level subtasks, which are solved by atomic skills (Pantano et al., 2022).

Task-based program representations structurally reflect this task hierarchy, and feature mechanisms for sequential and hierarchical composition that permit programmers, task planners or program synthesis algorithms to solve complex tasks in a divide-and-conquer, hierarchical manner. Unlike MPs, which chiefly represent *motions*, task-based program representations model *tasks*. Task-based representations typically expose task-relevant parameters such as object poses; MPs expose motion-level parameters such as velocities. Task models, then, model robot behavior at a higher level of abstraction, and rely on MPs or other motion-level representations to produce robot motions.

2.6.3.1 Task Models

Task models are graph structures which hierarchically model a task in terms of goals and constraints (Ekvall and Kragic, 2006). Planning algorithms then operate on these data structures to find a sequence of subtasks to satisfy high-level task objectives (*task planning*) or to plan low-level robot motions to satisfy the goals and constraints (*motion planning*).

Generalized manipulation strategies (Jäkel et al., 2012; Jäkel, 2013) represent a (sub-)task as a set of goals and constraints, which parameterize a sampling-based motion planner. Goals and constraints are expressed in terms of task-relevant coordinate systems, such as the opening of a bottle and the center of a cup for a pouring task, in a manner similar to the task frame formalism (Bruyninckx and De Schutter, 1996). Subtasks are combined to form *strategy graphs*, which define a partial order in which (sub-)goals have to be reached to achieve high-level tasks. To generate motions, a Constraint Satisfaction Problem (CSP) solver is combined with a Rapidly Exploring Random Tree (RRT) motion planner to plan robot trajectories which satisfy the task constraints. Strategy graphs can be learned from human demonstrations by automatic detection of contact coordinate frames and constraints, as well as pose constraints (Jäkel, 2013). Generalized manipulation strategies have evolved into the commercial ARTM industrial robot program representation, a graphical programming language with a set of industrial robot compiler backends to execute planned motions natively on robot controllers (Schmidt-Rohr et al., 2013; Jäkel and Dirschl, 2016).

Hierarchical Task Networks (HTNs) are task models designed to support hierarchical search-based planners similar to classical AI planners (Sacerdoti, 1975; Tate, 1977; Nau et al., 1999). Unlike the Planning Domain Definition Language (PDDL) and similar representations (Hoffmann, 2011), HTNs explicitly model *tasks*, *subtasks* as well as *methods*, rules to decompose tasks into subtasks (Ghallab et al., 2004). During the planning procedure, non-primitive tasks are recursively decomposed until primitive tasks are reached, which correspond to planning operators with defined pre- and postconditions. Unlike strategy graphs, which fold the CSP into a sampling-based motion planner, HTNs associate constraints with tasks as part of the planning domain, and considers them during the decomposition phase of the planning procedure (Georgievski and Aiello, 2015). Due to their reliance on search-based planners, HTNs are explicit, symbolic representations, describing tasks, goals, constraints and world states over discrete spaces. Due to their exploitation of structured domain knowledge embedded in the task constraints, HTN-based planners outperform classical AI planners for task planning (Hogg et al., 2016). HTNs have been used in conjunction with Learning from Demonstration (LfD) to represent robot programs in the context of human-robot collaboration (Hayes and Scassellati, 2016) or human-robot co-learning (Mohseni-Kabir et al., 2014). Lallement et al. (2014) propose a dedicated HTN representation and planner for robotics, which adds explicit data structures for agents along with rules for specifying “acceptable” or “unacceptable” behavior, as well as integration with a 3D simulation environment.

Several other task models have been designed to specifically support automatic planning for specialized applications or interaction modalities. Cheng et al. (2021) propose a hierarchical task model and associated planner for human-robot collaboration which represents both human and robot tasks in the same representation. Parallel (sub-)tasks are explicitly modelled, enabling an optimization-based planner to allocate robot subtasks to be executed in parallel with subtasks performed by the human, in order to minimize overall execution time. Mericli et al. (2014) propose *instruction graphs*, control flow graphs of robot primitives which are constructed by matching control structures (“if”, “while”, “until” etc.), primitives and their parameters to natural-language user inputs. Klee et al. (2015) present a planner for instruction graphs capable of deriving abstract tasks and parameter distributions from examples, and then use these distributions to autocomplete partially specified tasks.

Task and Motion Planning (TAMP) is a long-standing and highly active field of study (Garrett et al., 2020). TAMP observes that for many planning problems, task planning and motion planning cannot be performed independently: The feasibility of a task plan for example, is contingent on the feasibility of the underlying motions, such as collision-freeness and reachability (Guo et al., 2023). Likewise, some task plans may be highly efficient for some environments and robot kinematics, but highly inefficient for others. Jointly solving task- and motion-level planning problems promises improvements in planning speed as well as better plan results. Several robot program representations for TAMP have been proposed. Kaelbling and Lozano-Perez (2011) propose to represent robot tasks as a hierarchy of *plans*, each of which consists of abstract operators defined in a manner similar to STRIPS (Fikes and Nilsson, 1971). Primitive actions are planned using grasp and path planners; plans are composed of primitive actions and higher-level actions, which in turn have an associated plan. The HPN planner (Kaelbling and Lozano-Perez, 2011) then solves the planning problem recursively backwards from the goal Ghallab et al. (2004). The hierarchical nature of the plan structure is exploited to restrict the planning problem to short time horizons (Kaelbling and Lozano-Pérez, 2013). While traditional TAMP plan representations are explicit and symbolic, recent TAMP approaches have proposed a wide variety of hybrid program representations which all share a hierarchical structure, but vary widely in terms of representations of tasks, plans, primitives or planning algorithms. Paxton et al. (2017) propose to represent low-level motion policies and task-level policies as DNNs, trained to reflect task constraints expressed in Linear Temporal Logic. The planning problem is then solved via Monte Carlo Tree search. Driess et al. (2020) address TAMP via mixed-integer programming, using a trained neural network as a heuristic to guide the search over the program space. Silver et al. (2021) use a traditional Planning Domain Definition Language (PDDL)-based hierarchical plan structure and planning algorithm, but learn PDDL operators for robot skills from demonstrations.

Behavior Trees (BTs) are a graph-based representation for the behavior of artificial agents originally developed in the context of computer gaming (Colledanchise and Ögren, 2018). In the context of robotics, BTs are a popular robot program representation due to their modularity, facilitating modular reuse of behavior at different levels of abstraction (Bagnell et al., 2012). A BT is a directed tree whose leaves are *execution nodes*, and which are connected by *control flow nodes* (Colledanchise and Ögren, 2018). BTs explicitly model execution semantics as part of the representation: When a node is executed, it sends *ticks* to its children, which are in turn executed if they receive ticks. Control flow nodes comprise sequence, fallback, parallel and decorator nodes, which vary in their routing of ticks to children, and in their success and failure semantics. BTs are explicit, symbolic representations, as they model robot behavior with discrete, dedicated data structures. As a consequence, they support search-based planning of robot behavior to solve high-level tasks. Scheide et al. (2021) represent the search space of a BT as a formal grammar and use Monte Carlo Tree Search for task planning. Gugliermo et al. (2023) learn BTs from execution traces of manually written plans via logic factorization. Safronov et al. (2020) extend the BT formulation to plan in “belief state”, supporting non-deterministic actions and facilitating planning with the objective to reduce uncertainty.

2.6.3.2 Graphical Robot Programming Languages

Task models represent robot programs by explicitly modeling the hierarchical structure of tasks and providing data structures for task and motion planners. While they model programs at a higher level of abstraction that primitive-level representations such as MPs, they are typically intended to be used by algorithms such as planners, rather than human users. Graphical robot programming languages are robot program representations designed to enable humans to compose, edit and understand complex robot programs.

Platform-specific representations In industrial robotics, manufacturer-specific graphical program representations complement textual representations (see Section 2.6.1) and have supplanted textual programs as the default representation for common applications such as spot welding or palettizing . All major robot manufacturers provide graphical representations of programs as a hierarchical, tree-like structure, which programmers can interact with via GUIs on teach pendants attached to the robot controller or through desktop applications (Krot and Kutia, 2019; Heimann and Guhl, 2020). Unlike task models, graphical robot programming languages do not explicitly model task semantics such as goals or constraints, but are "syntactic sugar" (Landin, 1964) for the manufacturer's textual programming language, providing programmers with a higher-level view on an underlying textual robot program. Universal Robots (UR) provides the PolyScope graphical program representation (Universal Robots, 2018a), which permits robot programmers to compose and modify URScript programs by drag-and-drop of graphical function blocks in a tree structure on the teach pendant. Via a dedicated marketplace, UR programmers can download third-party predefined application-specific function blocks for e.g. machine tending or surface finishing. Fanuc's CRX suite of collaborative robots features a similar graphical program representation, which permits programmers to intuitively compose robot programs on the robot's teach pendant that directly correspond to textual KAREL and TP programs (FANUC America Corporation, 2023). While industrial graphical programming languages do not explicitly model task semantics and do not afford automatic task planning, they have been shown to enable inexperienced programmers to solve complex manipulation tasks, chiefly due to their hiding of syntactic complexity behind graphical abstractions (Ajaykumar et al., 2021).

Cross-platform representations The increasing proliferation of industrial robot manufacturers and the heterogeneity of the various sensors and actors used in complex manufacturing applications have given rise to cross-platform graphical program representations which are independent from individual hardware manufacturers. Cross-platform graphical robot programs explicitly model task semantics to varying degrees, and vary in the ways in which they relate to lower-level representations closer to the robot hardware.

The ArtiMinds Robot Task Model (ARTM) (Jäkel and Dirschl, 2016) is an industrial robot program representation based on strategy graphs (Jäkel, 2013). Tasks are modeled as a hierarchical graph of parameterized *templates*. Templates are pre-parameterized generalized manipulation strategies that represent common skills, such as linear or collision-free motion, grasping, insertion or search, but also higher-level tasks such as peg-in-hole insertion or aligning two objects. Templates as well as hierarchy and control flow structures such as if conditions or while loops can be composed by drag-and-drop and parameterized via an interactive wizard system. The ARTM is a hybrid program representation in that it combines three representations in one: A parameterized graph of visual, modular, semantically well-defined templates is internally represented by a strategy graph; the strategy graph, in turn, parameterizes a motion planner, which generates low-level robot trajectories and, depending on the task, associated runtime and goal constraints, e.g. for a force controller. The planned paths, goal and runtime constraints are then translated into executable robot programs in manufacturer-specific programming languages. Due to its cross-platform compatibility and the availability of a wide range of motion templates, the ARTM is used as a source program representation in the experiments in Chapters 3 and 4.

Similar cross-platform graphical robot programming languages and frameworks have been proposed (Krot and Kutia, 2019; Heimann and Guhl, 2020). The Intrinsic Flowstate (White, 2023) and drag&bot (Naumann, 2017) programming platforms, for example, represent robot programs in a manner similar to the ARTM, but directly control the robot via low-level joint commands, rather than generating robot code. All cross-platform graphical robot programming languages are explicit program representations with known task (and program) semantics, making them particularly suitable for industrial applications. As such, they are fundamentally explainable, and enable the auditing of robot behavior by human experts or automated systems.

2.6.4 Deep Neural Networks

Since the invention of backpropagation provided researchers with an efficient, general-purpose learning algorithm (Rumelhart et al., 1986), neural networks have been used as a highly flexible representation for learning, planning and optimization across all domains of science and engineering. Per the universal approximation theorem, multilayer feedforward neural networks with at least one hidden layer and nonlinear activations are capable of approximating any function at any degree of accuracy, given a sufficient amount of hidden units (Hornik et al., 1989). The availability and rapidly decreasing cost of computing hardware have enabled the development of deep neural architectures with very large numbers of hidden layers. Deep neural networks have enabled comparatively simple learning and optimization algorithms to solve highly complex problems purely from data, enabling the automated solutions for protein folding (Jumper et al., 2021), synthesis of high-resolution video (Blattmann et al., 2023) or multilingual machine translation (Fan et al., 2021), problems which had been considered very challenging, if not intractable, by classical AI methods. The appeal of neural networks lies in the fact that they are universal, implicit representations: Given sufficient training data, neural networks can be trained to represent nearly anything, without requiring a priori specifications of the structure of the representation. Suitable training algorithms can elicit latent representations exhibiting desired characteristics for planning (Lynch et al., 2019), classification (Yeh et al., 2017), or algebra (Lee et al., 2019); by virtue of representing information implicitly in latent space, neural networks delegate the search for an efficient representation to solve a given problem to the learning algorithm and training data. Neural networks support the solution of problems for which no efficient representation is known, e.g. because the input and output spaces are extremely high-dimensional, as in protein folding (Jumper et al., 2021). In robotics, deep neural networks have a long tradition of representing robot behavior (Horne et al., 1990; Pierson and Gashler, 2017; Kroemer et al., 2021). Neural robot program representations differ in the modeled input-output relationships as well as in the modeled level of abstraction, such as low-level motions or high-level tasks.

2.6.4.1 Deep Neural Policies

The most common neural robot program representation are what may be termed *neural policies*: Deep neural networks which predict the next action a robot should take, given an observation of the current world state. Robot behavior is often modelled as a partially observable Markov decision process (POMDP), where the robot executes an action at each time step, yielding a state- and action-dependent reward, followed by a transition to a new state (Lauri et al., 2023). Robot actions are typically incremental $\mathcal{C}$-space motions (joint state increments), Cartesian end-effector positions or joint torques (Bahl et al., 2020). Neural policies are often trained via RL (Kober et al., 2013; Levine et al., 2016), but can also be trained via imitation learning (Zheng et al., 2022) or other methods. For the purpose of this work, the learning algorithm is of lesser importance than how the learned policies represent robot behavior, and at which level.

End-to-end sensorimotor policies translate raw sensory input about the world and robot state into low-level joint positions, joint torques or Cartesian end-effector positions. Levine et al. (2016) train deep visuomotor policies for general-purpose manipulation using RL, which map raw images to joint torques using one single deep CNN. Hansen et al. (2022) present a deep visuotactile policy, which combines input from tactile sensors, 2D images of the environment as well as proprioceptive information on the robot state to produce Cartesian end-effector pose and is trained via RL to solve manipulation tasks. Diffusion policies (Chi et al., 2023) represent robot visuomotor policies as a conditional denoising diffusion process. Chi et al. (2023) train a neural network to represent the action distribution as a gradient field by predicting noise added in the training process, similar to the training of diffusion-based image generation networks (Rombach et al., 2022).

End-to-end sensorimotor policies implicitly represent robot motion or primitive robot skills at a level similar to MPs. To tractably solve long-horizon, complex tasks using end-to-end learning, hierarchical neural policy architectures have been proposed. Hierarchical architectures exploit the hierarchical nature of tasks in a manner similar to task models. At each hierarchy level, however, tasks and skills are represented implicitly in the latent space of neural networks (Pateria et al., 2021). Frans et al. (2018) propose a hierarchical RL approach, in which one *master policy* is trained to select a learned sub-policy suitable to the given task. Both master and sub-policies are trained together in an end-to-end manner. Gupta et al. (2020) propose a similar two-level neural representation, but employ a two-stage training process in which both levels are jointly trained first by imitation learning on human demonstrations of “semantically meaningful behaviors” not necessarily related to the final task, and then finetuned by RL on concrete long-horizon manipulation tasks. The DISH (Ha et al., 2021) architecture for hierarchical policy learning proposes a different approach: Instead of training a high-level policy to select among several low-level policies, a high-level neural policy generates task-relevant high-level parameters, which are then received by a single low-level policy to generate robot control signals. Unlike for task models, task parameters in DISH are not fixed and carry explicit semantic meaning, but carry implicit semantics which arise through the training process.

The general approach of prescribing a hierarchical architecture for robot behavior, but leveraging learning to manifest the concrete representation, is echoed in the literature on robot meta learning, which is concerned with “learning to learn” new tasks or task variants efficiently (Hospedales et al., 2021). Modular architectures such as Meta Networks (Munkhdalai and Yu, 2017) propose dedicated neural architectures for meta learning, in which one neural network (the *meta learner*) learns abstract knowledge across tasks and parameterizes another network (the *base learner*) to quickly learn new tasks. In Meta Networks, the representation of task knowledge, while implicit, is still modular: The meta learner represents aspects of task knowledge useful for learning new tasks, while the base learner represents the policy to execute the task. Model-agnostic meta learning methods such as MAML (Finn et al., 2017a) or Reptile (Nichol et al., 2018) facilitate meta learning for arbitrary model architectures by inducing representations suitable for meta learning via the training algorithm. Model-agnostic meta learning illustrates the representational flexibility of neural networks to simultaneously represent cross-task shared knowledge as well as task-specific knowledge in a single model, without the need for a priori representation design.

Most neural policies do not explicitly encode task objectives or constraints; rather, goals and constraints are implicitly represented, and are reflected in the reward function used at training time. This has the considerable disadvantage of requiring retraining when the task objectives or constraints change. Retraining upon changing task objectives may be avoided by conditioning the policy on an explicitly represented input of the task goal (Kaelbling, 1993). Gupta et al. (2020) and Groth et al. (2021) condition deep visuomotor policies on an image of the intended state of the environment after task completion. Lynch and Sermanet (2021) query the policy with a natural-language description of the task goal, while Myers et al. (2023) combine image and natural-language goal descriptions by learning an embedding of the language description that is aligned with a goal image. The explicit representation of task constraints typically require the use of algorithmic priors (Jonschkowski et al., 2018), the integration of differentiably implemented algorithms into the neural architecture, to force network outputs to the constraint manifold (Qureshi et al., 2020; Ni and Qureshi, 2024).

2.6.4.2 Multimodal Transformer Architectures

The widespread, popular adoption of ChatGPT (OpenAI, 2023) and similar large language model (LLM)-based AI assistants has given credence to the hypothesis that token-based sequence-to-sequence neural architectures can serve as general-purpose representations for tasks in a wide range of domains, including robotics (Zeng et al., 2023). The Transformer architecture (Vaswani et al., 2017) underlying all current state-of-the-art LLMs predicts the next token in a sequence by leveraging an attention mechanism, which computes the relative importance of tokens in a fixed-size context window around the current token for the current prediction. While the Transformer and related architectures quickly achieved state-of-the-art results for many applications such as time-series forecasting (Su et al., 2023) or machine translation (Vaswani et al., 2017), the availability of web-scale training datasets and increases in performance and availability of computational resources have permitted Transformer architectures to become the de facto default representation for general-purpose AI systems capable of superhuman performance on a variety of unrelated or only partially related tasks (Zhong et al., 2024). In robotics, multimodal Transformer architectures have seen increasing adoption due to their ability to represent policies with multimodal state or action spaces in one single network. Reed et al. (2022) propose Gato, one of the first Transformer-based multimodal general-purpose architectures. Gato first transforms the input sequence of tokens (integer-valued tensors) into token embeddings via a modality-specific embedding function. The token embeddings are then processed by a Transformer network to predict the next token in the sequence. Gato supports images, discrete actions, continuous actions, proprioceptive inputs such as joint torques and natural-language text as input and output modalities. Instead of employing modality- or task-specific submodules, Gato is jointly trained end-to-end on large-scale datasets spanning a wide range of tasks and modalities such as simulated robot control, Atari games, and vision-language tasks such as image captioning. While Gato's performance remained behind that of task- or modality-specific systems, it demonstrated the ability of Transformer architectures to learn useful cross-modality representations across tasks.

In the domain of robotics, Zitkovich et al. (2023) propose RT-2, a multimodal large-scale Transformer network for robot learning. RT-2 is a vision-language-action (VLA) model that takes token embeddings of natural language, images and robot actions as inputs and predicts the next action the robot should take, such as an incremental change in end-effector position and orientation. RT-2 can be used as a goal-conditioned neural policy by prompting it with an image of the current state of the environment and a natural-language description of the task goal to produce an action sequence (robot motion) to solve the task. RT-2 is co-finetuned on both web-scale datasets as well as a dedicated robotics dataset. Its performance gives credence to the hypothesis that pretraining and co-finetuning on multimodal data for tasks not associated with robot control, such as image captioning or visual question answering, benefits robot control, in that general-purpose representations of objects, naive physics, natural-language semantics and other concepts are learned, which aid in the inference of policies for e.g. manipulation tasks. Zitkovich et al. (2023) identify such “emergent capabilities” for symbol understanding, reasoning and human recognition.

The strong generalization capabilities and flexibility of RT-2 suggest its use as a *foundation model* in downstream AI systems, either as an architectural component in a modular neural architecture or by finetuning on more specialized tasks. Hu et al. (2023) provide an overview of foundation models in robotics. They highlight that dedicated robotic (VLA) foundation models propose an end-to-end approach to robot learning, acting effectively as multimodal neural robot control policies that implicitly learn perception, task understanding, planning and control in one large distributed representation. A range of dedicated VLA foundation models for robotics have been proposed (Bonatti et al., 2022; Bousmalis et al., 2023; Stone et al., 2023). Like deep neural policies, multimodal foundation models are implicit robot program representations. As such, their chief advantage is their ability to manifest multimodal, latent representations useful across a wide variety of tasks as a function of their training data and learning algorithm. The success of large-scale Transformer architectures has given credence to the *scaling hypothesis*: That “breakthrough progress [in AI] eventually arrives by [...] scaling computation by search and learning” (Sutton, 2019), evidenced by empirical research on neural scaling laws (Kaplan et al., 2020). While large-scale, implicit, distributed representations enable end-to-end robot task learning, they carry inherent interpretability challenges (see Section 2.7 for a detailed discussion). Moreover, reliance on scaling datasets and compute to achieve a given level of performance is challenging in specialized domains for which there is little training data, or in which the collection of training data requires physical interaction with humans or fragile workpieces (Hu et al., 2023). For these applications, the integration of foundation models with other representations into hybrid systems promises to increase task-specific performance without requiring additional training data.

2.6.5 Hybrid Representations

The discussed robot program representations each have their respective advantages and shortcomings, particularly with respect to ease of use by human programmers, interpretability, expressivity and support for data-driven learning and optimization. Consequently, a large and heterogeneous family of hybrid program representations combines explicit, implicit, symbolic and subsymbolic representations to optimally resolve these trade-offs. NRPs are a hybrid representation in that they are doubly neurosymbolic: The shadow program DCG is composed of modular deep neural networks as well as differentiable planners, forming a program graph via mechanisms of symbolic composition; moreover, NRPs are a dual representation, establishing a one-to-one structural and semantic equivalence between a skill-based source program (such as a task model) and its corresponding neurosymbolic shadow. In the literature on robot learning, program synthesis and optimization, a wide range of hybrid representations have been proposed, which combine representations in a variety of ways.

CRAM The Cognitive Robot Abstract Machine (CRAM) is a cognitive architecture which endows robots with mechanisms of cognition for perception, planning, action and metacognition (Beetz et al., 2010; Beetz et al., 2023). CRAM is tightly integrated with the KnowRob KR&R engine (Tenorth and Beetz, 2013; Beetz et al., 2018). The CRAM Plan Language (CPL) is a LISP-based domain-specific language (DSL) for specifying *plans* to solve tasks at varying levels of abstraction. CRAM plans are hierarchies of *designators*, symbolic representations for robot actions, objects, locations or low-level robot motions. CRAM plans can be *underspecified*, i.e. contain ungrounded designators, which can be resolved at planning time or dynamically at runtime by grounding from perception (Beetz et al., 2015; Mania et al., 2024), reasoning over knowledge bases (Beßler et al., 2021), prior experience (Beetz et al., 2018) or simulations (Haidu et al., 2018; Mania et al., 2021). CRAM plans are explicit and symbolic program representations in that the task structure and task parameters are directly reflected in the structure and parameters of CRAM plans. Due to the grounding of underspecified plans at runtime and the use of external knowledge bases, perception and simulations for symbol grounding, CRAM integrates both symbolic and subsymbolic robot programming. The KnowRob framework permits the definition of hybrid reasoning routines combining symbolic reasoning in Prolog with subsymbolic, external reasoners (such as grasp planners or object detectors) (Beßler, 2022). MetaWizard, the interactive AI assistant proposed in Chapter 4, leverages CRAM plans as an intermediate representation for the synthesis of NRPs, and proposes semi-symbolic reasoners to ground NRP parameters in human VR demonstrations, real-world percepts and simulations (Alt et al., 2024c).

Neural textual program synthesis The dual nature of NRPs representationally separates learning and program optimization from execution. Several recent LLM-based robot programming approaches are founded on a similar paradigm. Liang et al. (2023) propose Code as Policies (CaP), an approach for robot program synthesis using LLMs. CaP uses an LLM trained for general-purpose code generation (OpenAI Codex, Chen et al. (2021)) to synthesize textual robot programs given natural-language descriptions of the task. The LLM synthesizes Python code which uses and parameterizes function calls to a robot API providing perception and manipulation primitives, such as object detection, point-to-point motions or grasps. Instead of finetuning the LLM on the semantics of the robot API, the authors provide descriptions as comments and use descriptive function names, which are provided along with a natural-language description of the environment as part of the prompt. Liang et al. (2023) demonstrate that CaP can be used recursively, i.e. that the LLM can generate code containing calls to functions which, in turn, use the LLM to generate their own implementation. The textual, explicit robot program representation is used for human-AI interaction (goal conditioning via prompting) and program execution, but also as an intermediate representation in a recursive chain of reasoning. Reliance on an explicit, textual program representation with well-defined semantics permits the use of LLMs for program synthesis without retraining, as the LLM has been trained to perform high-level reasoning over general-purpose Python code, and all robot-specific or even environment-specific aspects, such as grounding of objects via visual perception, is hidden behind symbolic, textual abstractions. The paradigm of combining explicit, textual program representations with large neural networks to generate and manipulate them has been demonstrated in the context of simulation generation (Wang et al., 2023b), tabletop manipulation (Liang et al., 2023) or high-level household assistance (Vemprala et al., 2023). Instead of generating executable Python code, other approaches have generated explicit textual representations to serve as inputs for classical task or motion planners (Wake et al., 2023; Joublin et al., 2023; Rana et al., 2023).

Informed machine learning NRPs leverage differentiable motion planners such as DGPMP2-ND (Alt et al., 2025) and differentiable Cartesian trajectory generators to provide the neural networks with useful priors and reduce the learning problem from long-horizon trajectory prediction to a sequence-to-sequence translation. The field of *informed machine learning* is founded on the principle of intertwining implicit and explicit representations in order to leverage existing, known algorithmic solutions to (sub-)problems, such as planning algorithms, to improve the efficiency or safety of learning systems (von Rueden et al., 2021). Informed machine learning has been proposed in the context of representing of physical dynamic systems. Physics-informed neural networks (Cuomo et al., 2022), Neural ODEs (Chen et al., 2018), Neural Flows (Biloš et al., 2021) or neural Kalman filters (Revach et al., 2022) integrate *algorithmic priors* by tightly integrating neural network architectures with differentiable implementations of known system dynamics (Jonschkowski et al., 2018). In robot programming, Neural Dynamic Policies (NDPs, Bahl et al. (2020)) combine a deep neural network with a differentiable implementation of forward integration of the DMP differential equations. The resulting DCG can be trained end-to-end by supervised or reinforcement learning, outperforming purely neural architectures on a range of dynamic tasks such as opening a faucet, pushing or throwing. At the skill level, CNMPs (Seker et al., 2019) and related hybrid movement primitives (see Section 2.6.2) constitute similar hybrid representations.

2.7 Discussion

NRPs are a neurosymbolic robot program representation that combines the advantages of implicit and explicit representations to bridge the representational gap between learning and control. It is a dual program representation by virtue of maintaining an equivalence relation between a skill-based, explicitly represented source program, such as a task model or a textual robot program, and a neurosymbolic shadow program. NRPs are the first robot program representation to propose such a dual structure. By virtue of this duality, NRPs realize the requirements and desiderata laid out in Section 2.2.1.

2.7.1 Learning, Planning and Optimization

2.7.1.1 Learning

NRPs afford learning by gradient descent over the shadow program DCG: Shadow skills can be learned jointly or in isolation to represent real-world robot and world dynamics. NRPs leverage the learning capacity of DNNs to learn complex non-linear system dynamics, such as force-sensitive interactions with the environment during tactile manipulation. At the same time, the shadow program DCG combines DNNs with explicit representations of robot behavior, such as semantically meaningful shadow program parameters $\boldsymbol{x}^{\bar{P}}$, robot states $\boldsymbol{\theta}$ and trajectories $\boldsymbol{\theta}$ as well as differentiable priors in the form of Cartesian and $\mathcal{C}$-space motion planners. NRPs are a doubly neurosymbolic, hybrid program representation: It associates a symbolic source program with a neurosymbolic, differentiable surrogate. The structural and semantic equivalence properties of source and shadow programs facilitate shadow program learning in two ways. First, in a manner similar to hierarchical neural architectures (Pateria et al., 2021) such as DISH (Ha et al., 2021) or Relay Policies (Gupta et al., 2020), the hierarchical structure of the shadow program DCG decomposes the task-level learning problem into several skill-level subproblems, which can be learned in isolation or pre-initialized from a library of learned shadow skills. The learning problem is further simplified by the use of differentiable priors as an instance of informed machine learning (von Rueden et al., 2021), which reduce shadow skill learning to learning the residual between the prior and ground-truth trajectories, a considerably easier learning problem than generatively bootstrapping the posterior trajectory directly (He et al., 2016).

Crucially, NRPs learn predictive models of robot and world dynamics. This is a marked difference between NRPs and hierarchical neural policies: Neural policies are trained to represent the *optimal* behavior to solve a given task, whereas NRPs learn a *predictive* model, conditional on task or skill inputs and the robot state. While neural policies solve the problem of behavior optimization at training time by e.g. imitation learning, NRPs solve the comparatively easier problem of learning a forward model and relegate the optimization problem to dedicated optimization algorithms (see Chapter 3). This has three decided advantages. First, the optimization criterion is not required to be known at training time, and NRPs are trained to minimize a general-purpose trajectory prediction error (see Equation 2.1). Consequently, no retraining is required when the task objective changes, and explicit goal-conditioning is avoided, simplifying the training procedure. This property is crucial for service robotics applications, in which tasks goals typically change at every invocation of the program, and industrial applications, where e.g. small-batch production requires increasing flexibility with respect to frequently changing task objectives. Second, NRPs can be trained purely from unlabeled observation data of past task executions, with small perturbations to the input parameters and initial states. This manner of training data collection resembles Learning from Play (Lynch et al., 2019) and exhibits the same advantages. Crucially, it can be performed completely autonomously without human supervision, and avoids the shortcomings of RL-based approaches such as goal misgeneralization (Shah et al., 2022) and random exploration, which is infeasible in constrained industrial settings (Sünderhauf et al., 2018; Brosset et al., 2019). Third, the fact that NRPs are world models enables them to act as foundation models for downstream tasks, such as parameter optimization or task and motion planning. Echoing related work on robotics foundation models (Zitkovich et al., 2023), Kienle et al. (2024) propose a neural shadow skill architecture for NRPs based on multimodal, visuotactile Transformer architecture, which combines vision and trajectory modalities and can be pretrained for a wide range of robotic manipulation tasks.

One limitation of a dual program representation founded on structural and semantic equivalence is that the (in principle) unlimited expressivity of the shadow program representation can only translate to flexible robot behavior within the representational constraints of the source program. Consider a NRP with a URScript source program. As the source program is used for execution on the robot, the robot's range of possible behaviors will be constrained by the URScript API; downstream optimizers or planning algorithms can only optimize URScript function parameters or plan URScript programs. From a safety perspective, this is a requirement in many industrial applications; from a capabilities perspective, this can be a severe limitation. The flexibility of NRPs to support *arbitrary* skill-based representations as source programs permits the user to choose a source program representation with a sufficient level of expressivity for the intended application.

2.7.1.2 Planning and Optimization

Their dual structure permits NRPs to afford first-order optimization of program parameters for source programs which are not differentiable (Alt et al., 2021; Alt et al., 2022b; Alt et al., 2025) (see Chapter 3). As such, NRPs are the first robot program representation to realize first-order optimization via a differentiable surrogate, similar to the use of differentiable surrogates in other engineering domains (Vandegar, 2020). Section 3.4 provides a detailed analysis of first-order optimization of NRPs. In addition, structural equivalence to an explicit, modular program representation such as a task model permits the use of symbolic task planners for the source program, which exploit prior knowledge and the hierarchical task structure for efficient planning (Alt et al., 2023; Alt et al., 2024c) (see Chapter 4). This echoes the advantage of modular program representations such as HTNs, task models or TAMP operators, which support efficient search-based task planning due to their hierarchical structure with explicit semantics. Section 4.4 provides a detailed discussion of program synthesis with NRPs.

2.7.2 Hardware and Task Agnosticity

NRPs are agnostic with respect to the source program representation, provided it is parameterized and explicit. A corollary of this representational agnosticity is that NRPs can be used to learn and optimize robot programs for a wide variety of tasks and hardware platforms, provided they can be represented by the source program. Source program representations such as ARTM task graphs (Jäkel and Dirschl, 2016) and other task models, sequences of movement primitives or other explicit representations are generally suited. Likewise, due to the reliance on DNNs as universal function approximators in the shadow program DCG, NRPs can be trained to represent arbitrary world and robot dynamics. The representational power of NRPs is limited largely by the input and output modalities of the shadow skills. The architecture proposed in Section 2.4.1.4 accepts arbitrary real-valued skill input tensors and start states in state space $\mathcal{S}$ and outputs posterior trajectories in trajectory space $\mathcal{S}^M$. This effectively restricts it to only represent tasks whose semantics can be expressed in terms of Cartesian end-effector poses, $\mathcal{C}$-space configurations and end-effector wrenches (see Section 2.4.1.1). This state space covers a wide range of complex manipulation tasks in service and industry robotics (see Chapter 3); however, the poses of objects in the environment or the behavior of other agents are not explicitly represented. In principle, the input and state spaces can be extended to arbitrary real-valued spaces without loss of generality. Kienle et al. (2024) extend the input space of NRPs by a tokenized representation of images of the environment to represent visuotactile tasks which require force and vision-adaptive behavior.

By similar logic, NRPs are as hardware-agnostic as the chosen source program representation. For cross-platform graphical programming languages, task models or similar hardware-agnostic representations, NRPs can be trained to represent the kinematics and dynamics of arbitrary robot hardware. NRPs have been trained and tested chiefly on six-axis industrial robot arms (see Chapter 3), though they are not limited to 6-DoF serial kinematics.

2.7.3 Human Interpretability

One considerable advantage of explicit program representations is that they allow human programmers or formal provers to audit robot programs which may have been generated or optimized by AI systems (Espiau et al., 1996; Webster et al., 2016; Luckcuck et al., 2019). Implicit representations, such as DNNs, distribute the representation of concepts across neurons and layers (Hinton, 1989), making the interpretation of network behavior highly challenging (Chang et al., 2024). A corollary of the distributed nature of representation in neural networks is *polysemanticity*, the ability of individual neurons to simultaneously represent semantically unrelated concepts by “superposition” of sparse features (Elhage et al., 2022). Polysemanticity has been observed in networks of all sizes and architectures (Scherlis et al., 2023) and poses significant challenges to mechanistic interpretability, as post-hoc disentanglement of features, e.g. via sparse autoencoders, is lossy and computationally expensive (Huben et al., 2023). Hospedales et al. (2021) note that while e.g. meta-learning symbolic, human-interpretable activations is possible, this comes at the cost of eschewing end-to-end differentiability and the use of first-order optimizers or loss of accuracy. Deep neural policies, which learn implicit representations of task goals and constraints, pose an additional challenge to interpretability, as the learned objectives or constraints may not reflect the true task objectives or constraints, e.g. due to reward misspecification (Pan et al., 2021), reward hacking (Skalse et al., 2024), goal misgeneralization (Shah et al., 2022) and related phenomena (Pitis, 2023). In a manner similar to the concept of “global surrogates” in the XAI literature (Burkart and Huber, 2021), NRPs leverage a surrogate model to resolve the capability-safety tradeoff. However, where global surrogates in XAI learn an explicit representation to be functionally equivalent to an implicit, neural representation, NRPs start from an explicit, explainable representation and train a differentiable, implicit surrogate for learning and optimization. By using the explicit, explainable source program representation for user interaction and execution on the robot, NRPs ensure that the ultimate behavior of the robot is explainable to the human programmer. Their dual nature enables NRPs to leverage the representational flexibility and differentiability of implicit neural program representations for learning and first-order optimization without requiring them to be explainable. The “safety tax” of NRPs lies in the limited expressivity of explicit program representations. The neural components of shadow skills can, in principle, learn to represent arbitrary behavior; this would, however, violate the semantic equivalence property, negating the core benefit of NRPs – facilitating first-order optimization over explicitly represented robot programs that could otherwise not be optimized. In practice, well-designed task models offer sufficient flexibility to express tasks of very high complexity. The factor delaying or preventing the use of robots for complex tasks rather lies in the difficulty of identifying good or optimal parameters for robot skills, while requiring the explainability of skill-based robot programs.

2.7.4 Human Interaction

The duality between explicit source and implicit shadow program representations not only ensures the explainability of robot behavior, but also facilitates the creation and modification of NRPs by human programmers. The modularity and exposure of semantically well-defined task-space parameters makes robot programming languages, task models and hybrid representations such as CRAM plans particularly suited for use in interactive programming paradigms such as textual or task-based programming. Even under data-driven programming regimes such as Programming by Demonstration (PbD), explicit representations facilitate the post-hoc editing of robot programs by human experts. End-to-end neural approaches lack such mechanisms for post-hoc modification, and interactive learning paradigms such as active or imitation learning facilitate human interaction during training, but their lack of interpretability prevents targeted post-hoc editing of the learned policies. NRPs balance data-driven learning and first-order optimization with the ability of humans to edit the explicitly represented source programs and their parameters before or after optimization. Use of source program representations designed for human programming, such as textual programming languages, permits programmers to interact with robot programs via established interfaces and abstractions, such as teach pendants, GUI applications or textual editors, while simultaneously permitting purely data-driven learning and optimization by human demonstration (Alt et al., 2021; Alt et al., 2025), lifelong self-learning (Alt et al., 2022b), interactive program synthesis in VR (Alt et al., 2023) or natural-language interaction (Alt et al., 2024c).

By giving a central role to explicit program representations, NRPs re-establish AI-enabled programming as bidirectional, interactive communication between human and machine. However, Chen and Huang (2023)'s criticism of LLM-based architectures applies to NRPs as well: The bidirectional communication between programmer and programmed system is not lossless. While the use of explicit program representations for execution and human-machine interaction permit human editing and understanding of the generated program, that does not necessarily imply correctness of the program. Chen and Huang (2023) propose *factual* and *physical* error classes. Factual errors are e.g. erroneous model outputs due to hallucinations, such as predicted trajectories which do not approximate the physical reality, or do not approximate it to sufficient fidelity, e.g. due to overfitting on insufficiently diverse training data. Physical errors include e.g. collisions with the environment at runtime. NRPs are susceptible to both error types, and any downstream use of NRPs, such as for first-order optimization or program synthesis, will be affected by such model errors. The modular architecture of NRPs, however, helps contain the scope of model errors at the skill or hierarchy level, and the semantic equivalence between the source program and the shadow program DCG enables straightforward testing against such errors – the quality of e.g. a learned shadow skill is exactly proportional to the degree to which it approximates its corresponding source skill.

2.7.5 Additional Desiderata

Beside simplifying the learning problem and facilitating human interpretability and editability, the modular architecture of NRPs ensures that they satisfy the additional desiderata for neurosymbolic program representations introduced in Section 2.2.1. NRPs facilitate symbolic composition, i.e. the construction of complex programs from primitive, semantically well-defined components (Chaudhuri et al., 2021); moreover, they facilitate hierarchical abstraction, permitting to hide complex skill sequences behind higher-level subtask abstractions, and afford learning at all levels of hierarchy. Symbolic composition and hierarchical abstraction in NRPs is not complete, however. The length of skill or subtask sequences and the depth of hierarchies is limited by the available computational resources, such as graphics processing unit (GPU) memory. Moreover, loop structures with many iterations can cause vanishing gradients, which may require additional regularization (Wu et al., 2021).

From an application perspective, one of the most important corollaries of the dual nature of NRPs is their interoperability with existing programming systems, workflows and execution environments. The realities of industrial robot programming, in particular, mandate a program representation that seamlessly integrates into existing robot programming ecosystems. The agnosticity of NRPs with respect to the source program representation permits the creation of NRPs for existing robot programs in a wide range of commonly-used textual, task model-based or hybrid source program representations by training the shadow DCG on existing robot experience data. In industrial robotics, this permits the use of NRPs alongside traditional methods of robot programming, reducing the overhead of adoption (Alt et al., 2024a).

2.8 Conclusion

NRPs are a novel neurosymbolic robot program representation that combine a skill-based robot program for execution and human editing with a learnable, differentiable shadow DCG for first-order optimization. NRPs are a dual representation, enforcing a bidirectional correspondence between source and shadow representations. A wide range of existing skill-based robot program representations can be converted to NRPs, comprising most industrial robot programming languages and a range of state-of-the-art robot task models. NRPs support sequential and hierarchical composition of primitive skills to complex programs, as well as control flow structures for iteration and conditional branching. By integrating neural networks, differentiable programming and collision-free motion planning into one unified DCG, NRPs act as differentiable models of robot behavior. As a data structure, NRPs afford the design of symbolic and neural program synthesis and program optimization algorithms, which are the subject of Chapters 3 and 4. NRPs have evolved in the context of several publications (Alt et al., 2021; Alt et al., 2022b; Alt et al., 2025; Kienle et al., 2024). Their ability to represent complex force-dynamic manipulation tasks is demonstrated in the context of several real-world industrial and service robotics applications, notably Experiments 3.1.3, 3.2.3 and 3.3.2. By bridging the representational divide between the symbolic, explicit robot program representations designed and optimized for human programmers, and the subsymbolic, implicit program representations favored by first-order learning and optimization approaches, NRPs support neurosymbolic robot programming workflows that satisfy the needs of both human programmers and AI-based programming methods.

CHAPTER 3

First-Order Robot Program Parameter Optimization

The identification of suitable program parameters to achieve high-level task objectives is one of the core challenges of robot programming. Under the manual programming paradigm, robot programmers specify both the structure of the robot program as well as its parameterization, to specify robot behavior which achieves the task objectives. Textual program representations typically require fine-grained parameterization at the motion level, such as the specification of velocities, accelerations, blending radii, force controller parameters, time delays etc. A comparatively simple robot program for peg-in-hole insertion, for example, may have tens to hundreds of subsymbolic, continuous-valued program parameters, controlling aspects such as the force and torque setpoints for a proportional-integral-derivative (PID) controller to push the peg into the hole, or parameters defining approach and depart poses as well as intermediate waypoints. Identifying parameterizations which not only solve a task, but also optimize for nonfunctional task objectives such as cycle time or robustness, is a challenging high-dimensional multicriterial optimization problem. In the current industrial state of the art, robot program parameterization is performed by human experts in a costly, time-consuming process of trial and error. Automating this process via AI-based, data-driven methods has the potential of freeing resources and allowing robot programmers to use their knowledge more effectively.

As an example for an industrial robot program parameterization problem, consider the `servoj` function of the URScript programming language, one of its core functions for position-controlled motion (Universal Robots, 2018b). `servoj` accepts six parameters: A target joint configuration `q`, velocity `vel`, acceleration `acc`, time duration `t` during which the robot is controlled, a `lookahead_time` which smooths the resulting trajectory, and a `gain` parameter indicating the degree of fidelity to which the reference trajectory is to be followed. Even for a simple robot program causing a robotic manipulator to follow a reference trajectory, parameterizing `servoj` is highly challenging: First, the states `q` on the reference trajectory must

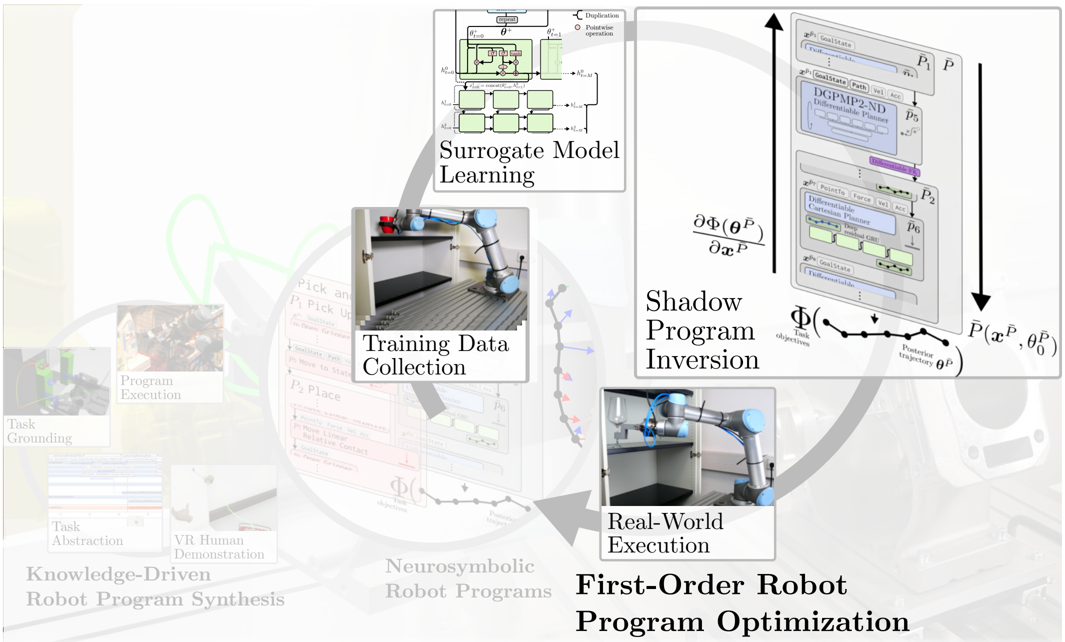

Duplication
Pointwise operation
Surrogate Model Learning
Training Data Collection
$\frac{\partial \Phi(\theta^{\bar{P}})}{\partial x^{\bar{P}}}$
DGPMP2-ND Differentiable Planner
Differentiable Cartesian Planner
Shadow Program Inversion
$\Phi($
Task objectives
Posterior trajectory $\theta^{\bar{P}}$ $)$
$\bar{P}(x^{\bar{P}}, \theta_0^{\bar{P}})$
Real-World Execution
Task Grounding
Program Execution
Task Abstraction
VR Human Demonstration
Knowledge-Driven Robot Program Synthesis
Neurosymbolic Robot Programs
First-Order Robot Program Optimization

Figure 3.1: Shadow Program Inversion (SPI) is an optimization algorithm for Neurosymbolic Robot Programs (NRPs). By gradient descent over the shadow program, program parameters and motion trajectories are optimized with respect to user-provided task objectives, subject to motion-level constraints.

be determined, e.g. by a motion planner. Appropriate values of `vel` and `acc` must be specified to determine the motion dynamics, often under motion-level task constraints such as acceleration limits to prevent spillage of liquids or minimize energy consumption. The parameters `t`, `lookahead_time` and `gain` jointly determine the overall smoothness of the trajectory as well as the path accuracy. Their mutual influence and exact mathematical meaning are not transparent to the user and poorly documented, requiring several iterations of trial and error even for experienced programmers.

In the context of task-based programming, parameter optimization is an equally important and challenging problem. Task-based program representations abstract away most motion-level and hardware-specific details in favor of task-level parameterizations. One central challenge is the need for joint optimization of task- and motion-level parameters. Consider the example of a service robot placing a cup from a table into a shelf (see Figure 3.18). Using the ARTM graphical programming language (Schmidt-Rohr et al., 2013), this task can be solved using a sequence of the predefined subtasks `Grasp`, `Move to State` and `Ungrasp`. `Grasp` and `Ungrasp` are in turn composed of primitives for approaching and departing the cup, as well as primitives for closing and opening the gripper. Program parameterization requires specifying the approach and depart poses, the initial gripper opening for grasping, the pose of the cup before grasping, the object-relative grasp point, the target position of the cup for ungrasping, as well as velocities and accelerations for all primitives. Crucially, `Move to State` and the approach and depart primitives plan collision-free motions, given a 3D representation of the environment. The choice of approach, depart and grasp poses can affect the feasibility of collision-free motion, and will greatly affect its efficiency, e.g. in terms of energy consumption.

This chapter introduces SPI, a general-purpose parameter optimizer for parameterized robot programs. SPI performs first-order model-based optimization over NRPs to compute optimal parameterizations with respect to a wide range of task objectives, such as cycle time, robustness, or proximity to a human demonstration. Section 3.1 introduces SPI, contextualizes it in the field of program parameter optimization and describes experiments evaluating SPI in both industrial and household settings. Section 3.2 proposes a method for lifelong learning and continuous program re-optimization in the face of real-world stochastic noise processes. This lifelong variant of SPI is evaluated on industrial tactile peg-in-hole applications subject to nonstationary drift and shift processes. Section 3.3 casts parameter optimization as joint task- and motion-level constrained optimization problem and presents an extended version of SPI capable of simultaneously optimizing task parameters and motion trajectories, as well as a comprehensive evaluation on industrial quality assurance and household pick-and-place tasks.

3.1 Optimization of Robot Program Parameters via Inversion of Differentiable Shadow Programs

Section 2.3.2 defines a robot program P as a discrete-time continuous-valued stochastic process $P := \{\Theta_t\}$ with time index $t \in \{1, 2, ..., M_P\}$ and state space $\mathcal{S}$. A *trajectory* $\boldsymbol{\theta}^P$ is a sample of $\{\Theta_t\}$, containing M_P states $\theta_t^P, t \in \{1, 2, ..., M_P\}$. The law of $\{\Theta_t\}$ is parameterized by the program inputs $\boldsymbol{x}^P$ and conditional on the initial state θ_0^P as well as a random variable H representing the environment.

The parameter optimization problem concerns finding the optimal parameterization $\boldsymbol{x}^{P,*}$ to minimize a cost function Φ reflecting the task objectives:

$$\boldsymbol{x}^{P,*} = \arg\min_{\boldsymbol{x}^P} \Phi(P) \tag{3.1}$$

◁ *Parameter optimization problem*

Consider the industrial peg-in-hole task introduced in Section 1.1.1.2 and the robot program shown in Figure 2.4. Optimization of the `ExtentsX` and `ExtentsY` parameters of the `Spiral Search Relative` skill, as well as the `PointTo` parameter of the preceding `Approach` skill, can ensure that the `Spiral Search Relative` is executed in close proximity to the hole, and that the shape of the spiral is aligned with the spatial distribution of hole positions to, on average, reliably find the hole, without spending excessive time on search in low-probability areas. Even for this comparatively simple use case, the parameter optimization problem expressed in Equation 3.1 is ill-posed. First, the probabilistic view on P is mathematically underspecified: How exactly $\boldsymbol{x}^P$ parameterizes the law of $\{\Theta_t\}$, or how that probability measure could reasonably be represented mathematically, cannot be specified for the general case, as robot programs vary widely from application to application. Moreover, a true solution to Equation 3.1 optimizes $\boldsymbol{x}^P$ for executions of P in real-world environments; however, the real world is not a mathematical object over which an optimization algorithm can be defined.

SPI proposes to solve the optimization problem via a well-defined, learned model of the robot program's real-world execution dynamics (Alt et al., 2021). SPI leverages the NRP program representation (see Chapter 2) to associate P with a DCG $\bar{P}$, which contains neural components that can be trained to approximate P for a given range of parameters and start states in a given environment. Unlike P, $\bar{P}$ is a computable function of the program inputs $\boldsymbol{x}^{\bar{P}}$; moreover, $\bar{P}$ is differentiable, permitting the use of first-order optimizers over robot behavior. Due to structural equivalence, the optimized shadow program parameters $\boldsymbol{x}^{\bar{P},*}$ can be transferred back to the original model.

SPI was first introduced in Alt et al. (2021). This section expands on this publication and presents SPI in greater detail.

3.1.1 Overview

Unlike for the source program P, the parameter optimization problem for the shadow program $\bar{P}$ is well-defined:

Shadow program ▷ parameter optimization problem

$$\boldsymbol{x}^{\bar{P},*} = \arg\min_{\boldsymbol{x}^{\bar{P}}} \Phi(\bar{P}(\boldsymbol{x}^{\bar{P}}, \theta_0^{\bar{P}})) \tag{3.2}$$

$\bar{P}$ is a DCG, and as such a differentiable function of its leaf nodes $\boldsymbol{x}^{\bar{P}}$ and $\theta_0^{\bar{P}}$. $\bar{P}$ can be evaluated by a computer without requiring real-world execution on a robot, and reflects real-world robot behavior by virtue of being trained on robot data. The environment H is implicitly represented in the learned weights of the shadow program's neural networks.

As $\bar{P}$ is a function, the ideal solution to the parameter optimization problem would be to invert $\Phi(\bar{P})$ to yield the inputs and start states that achieve some desired value ϕ of Φ:

$$\boldsymbol{x}^{\bar{P},*}, \theta_0^{\bar{P}} = (\Phi(\bar{P}(\boldsymbol{x}^{\bar{P}}, \theta_0^{\bar{P}})))^{-1} = \bar{P}^{-1}(\Phi^{-1}(\phi)) \tag{3.3}$$

Generally, the shadow program $\bar{P}$ will not be invertible, as it is generally not injective: Several parameterizations may produce the same robot behavior. Moreover, the DCG of $\bar{P}$ will contain operations that do not have a unique inverse, such as the square function or FK. The differentiability of $\bar{P}$, however, permits iterative inversion by gradient descent in the input space: For each iteration i, a forward pass through the shadow program DCG is performed, yielding the posterior trajectory $\boldsymbol{\theta}^{\bar{P},i}$. $\Phi(\boldsymbol{\theta}^{\bar{P},i})$ is backpropagated through the DCG by automatic differentiation (Paszke et al., 2017), permitting the computation of the gradient $\frac{\partial \Phi(\bar{P}(\boldsymbol{x}^{\bar{P},i}, \theta_0^{\bar{P}}))}{\partial \boldsymbol{x}^{\bar{P},i}}$ of

```
def optimize(p: DCG, x_init: Tensor, theta_0: Tensor, gamma: float, phi: [Tensor] -> Tensor, num_iterations: float) -> Tensor:
    x = x_init
    optim = Adam(parameters=[x], lr=gamma)
    for i in range(num_iterations):
        optim.zero_grad()
        posterior_trajectory = forward(p, x, theta_0)
        loss = phi(posterior_trajectory)
        loss.backward()
        optim.step()
    return x

```

Listing 3.1: SPI iteratively optimizes the input parameters `x` of a DCG `p` with respect to task objectives `phi` by gradient descent.

the task objective Φ with respect to the current program inputs $\boldsymbol{x}^{\bar{P},i}$. The inputs are then updated via first-order optimization, such as stochastic gradient descent (SGD):

$$\boldsymbol{x}^{\bar{P},i+1} = \boldsymbol{x}^{\bar{P},i} - \gamma \frac{\partial \Phi(\bar{P}(\boldsymbol{x}^{\bar{P},i}, \theta_0^{\bar{P}}))}{\partial \boldsymbol{x}^{\bar{P},i}}, \tag{3.4}$$

◁ *Iterative parameter optimization*

where γ is the update rate. In practice, an adaptive first-order optimizer such as Adam (Kingma and Ba, 2015) results in faster and more stable optimization. Listing 3.1 provides a high-level summary of SPI.

SPI exposes two high-level hyperparameters: The update rate γ and the number of iterations N_{iter}. Note that only the program inputs $\boldsymbol{x}^{\bar{P}}$ are optimized, while the weights of the neural networks in $\bar{P}$ remain frozen and dropout is disabled. Like the shadow program DCG itself, SPI is implemented in PyTorch (Paszke et al., 2019) and makes efficient use of GPU acceleration for both forward and backward passes.

3.1.2 Differentiable Task Objectives

SPI permits the optimization of robot program parameters with respect to a wide range of task objectives Φ, provided Φ is a differentiable function of the posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$ generated by $\bar{P}$. Task objectives may be simple boolean conditions such as task success, real-valued indicators such as cycle time, more complex criteria such as similarity to a human demonstration, or combinations thereof. This section introduces SPI through the lens of first-order iterative program inversion, solving a multicriterial optimization problem. Section 3.3 re-casts program parameter optimization as a constrained optimization problem and introduces additional task- and motion-level constraints such as smoothness or collision-freeness.

3.1.2.1 Process Metrics

Program parameter optimization in industrial robotics is frequently motivated by a need for industrial production processes to meet non-functional requirements such as remaining below a maximally permissible time limit (the *cycle time*) to perform a task, or performing a task with limited joint effort. Other applications, such as the production of expensive, labor-intensive products, have strict functional requirements with respect to task success rates, limiting the amount of rejects, or with respect to production tolerances such as limits on the forces and torques exerted on cables or connectors.

Cycle time As the posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$ has a fixed temporal sampling interval, the number of points on $\boldsymbol{\theta}^{\bar{P}}$ serves as a proxy for the cycle time. However, counting the number of items in a sequence is not a differentiable operation. SPI realizes a differentiable proxy for the cycle time by summation of EOS tokens along the posterior trajectory:

Cycle time ▷ objective

$$\Phi_{\text{cycle}}(\boldsymbol{\theta}^{\bar{P}}) = \sum_{n=1}^{|\boldsymbol{\theta}^{\bar{P}}|} \Big(1 - \sigma\big((\boldsymbol{\theta}^{\bar{P}}_{n,\text{EOS}} - 0.5) * T\big)\Big) \tag{3.5}$$

The Sigmoid function σ over the EOS tokens of the posterior trajectory, re-centered around 0 and spread to the edges of the domain of σ by multiplication with a large constant T (here 100), returns a value very close to one if the motion has been completed before timestep n, and a value close to zero otherwise. Φ_{cycle} effectively “counts” the number of zero-valued EOS tokens on $\boldsymbol{\theta}^{\bar{P}}$ in a differentiable way.

Task success By a similar principle, the task success tokens of the posterior trajectory can be exploited to realize a differentiable measure of task success. As Φ is a task *cost* to be minimized, the probability of task failure is given by

Task success ▷ objective

$$\Phi_{\text{fail}}(\boldsymbol{\theta}^{\bar{P}}) = \max\Big(0, \min\big(\frac{1}{|\boldsymbol{\theta}^{\bar{P}}|} \sum_{n=1}^{|\boldsymbol{\theta}^{\bar{P}}|} \boldsymbol{\theta}^{\bar{P}}_{n,\text{succ}}, 1\big)\Big). \tag{3.6}$$

Note that Φ_{fail} does not return the true probability of task failure, as $\bar{P}$ is not a probabilistic model. Rather, it is proportional to the average of the task success tokens on $\boldsymbol{\theta}^{\bar{P}}$. This formulation implicitly assumes that the program P successfully solved a task if all subprograms P_i were successful. For cases in which only a subset of subprograms is relevant for task success, Φ_{fail} can be computed over the corresponding sub-trajectories – due to backpropagation, only program parameters which influence those subtrajectories, both directly and indirectly, will be optimized.

Path length For motions defined in Cartesian space, the path length is a useful proxy for the overall joint effort. SPI defines a Cartesian path length measure

$$\Phi_{\text{path}}(\boldsymbol{\theta}^{\bar{P}}) = \frac{1}{|\boldsymbol{\theta}^{\bar{P}}|} \sum_{n=1}^{|\boldsymbol{\theta}^{\bar{P}}|-1} \Big(\|\boldsymbol{\theta}^{\bar{P}}_{n+1,\text{pos}} - \boldsymbol{\theta}^{\bar{P}}_{n,\text{pos}}\|_2 + d_q(\boldsymbol{\theta}^{\bar{P}}_{n+1,\text{ori}}, \boldsymbol{\theta}^{\bar{P}}_{n,\text{ori}}) \Big), \tag{3.7}$$

◁ *Path length objective*

where $\|\cdot\|_2$ is the L2 norm, $\boldsymbol{\theta}^{\bar{P}}_{n,\text{pos}}$ and $\boldsymbol{\theta}^{\bar{P}}_{n,\text{ori}}$ represent the Cartesian end-effector position and orientation components of $\boldsymbol{\theta}^{\bar{P}}$, respectively, and d_q is the quaternion distance

$$d_q(q_1, q_2) = \cos^{-1}(2\langle q_1, q_2\rangle^2 - 1), \tag{3.8}$$

where $\langle q_1, q_2\rangle$ denotes the inner product of quaternions q_1 and q_2.

Force and torque objectives Many industrial applications, such as peg-in-hole assembly or surface finishing, involve force-controlled interaction with the environment. Service robotics applications also often require the handling of sensitive, breakable objects such as plates or glasses. For such applications, the task objective

$$\Phi_{\text{max\_wrench}}(\boldsymbol{\theta}^{\bar{P}}) = \max \boldsymbol{\theta}^{\bar{P}}_{:,\text{wrench},d} \tag{3.9}$$

◁ *Wrench limit objective*

computes the maximal force or torque along the dth dimension of the 6-dimensional wrench vectors on the posterior trajectory. Alternatively, the task objective

$$\Phi_{\text{contact}}(\boldsymbol{\theta}^{\bar{P}}) = \text{MSE}(\boldsymbol{\theta}^{\bar{P}}_{M^{\bar{P}},\text{wrench}}, \omega_0) \tag{3.10}$$

◁ *Contact wrench objective*

penalizes wrenches at the end of the motion which deviate from a wrench setpoint ω_0. This task objective is particularly useful for the optimization of force-controlled contact motions, such as when placing THT electronics components on a PCB (see Experiment 3.1.3.1).

3.1.2.2 Multicriterial Optimization

Most real-world applications require a combination of possibly contradicting task objectives. Consider the force-controlled electronics assembly task introduced in Section 1.1.1.2. Both task success and cycle time are relevant optimization targets; they are, however, mutually exclusive, in that slower, more fine-grained search over a larger search area increases the probability of success at the cost of additional search time. At the task level, SPI supports multicriterial optimization by weighted addition of multiple task objectives to an aggregated task cost

$$\Phi(\boldsymbol{\theta}^{\bar{P}}) = \sum_{i=1}^{N^{\Phi}} w_i \Phi_i(\boldsymbol{\theta}^{\bar{P}}), \tag{3.11}$$

◁ *Multicriterial task objective*

where N^{Φ} is the number of task objectives. In addition to task-level objectives, Section 3.3 introduces motion constraints, which further contrast with task objectives, e.g. by mandating collision-freeness.

3.1.3 Experiments

3.1.3.1 Optimization of Force-Controlled Contact Skills

To evaluate SPI in the context of industrial manipulation, the optimization of force-controlled contact motions is considered. Force-controlled contact is an important building block of many industrial robot tasks, such as assembly or quality control. Moreover, force-controlled contact motions often constitute crucial bottlenecks in industrial robot processes, as they must be comparatively slow to avoid build-up of excessive forces. The manual parameterization of contact motions is challenging because the actual contact force is determined by a spring-mass-damper system composed of the robot and the contact surface, whose damping and spring characteristics are a priori unknown. The following experiments test the hypothesis that SPI can optimize the parameters of force-controlled contact motions to avoid exceeding force limits, while minimizing total cycle time.

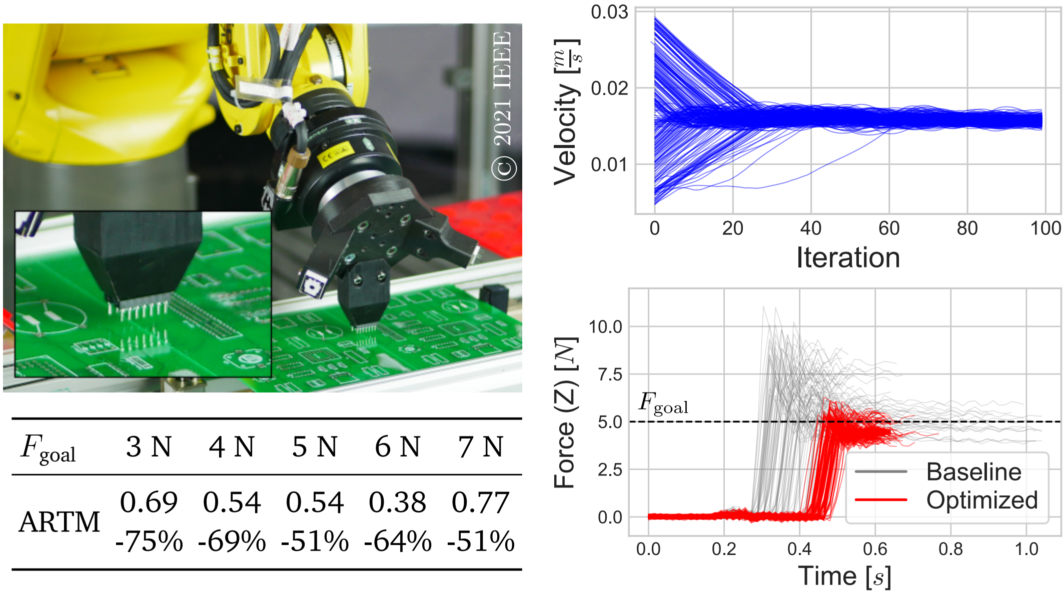
Data-driven optimization of contact forces In an initial set of experiments, SPI is used to optimize the motion direction, velocity and acceleration of a `Move Linear Relative Contact` ARTM skill. `Move Linear Relative Contact` moves the robot along a given direction vector `PointTo` with a velocity `Vel` and acceleration `Acc` until a force setpoint F_{goal} is reached. The actual contact force F_{contact} will generally exceed F_{goal}, as the robot will begin decelerating only after F_{goal} has been registered, moving further into the surface. F_{goal} merely imposes a lower bound on the maximum force, while `Vel` and `Acc` determine the true force upon contact (see Figure 3.2 (bottom right, gray)).

A shadow model is pretrained on 50,000 simulated trajectories and finetuned on 500 real-world executions with randomly sampled values of `Vel` and `Acc`. Simulated data was generated by combining a linear acceleration model of Cartesian end-effector motion with a spring-damper system, whose parameters roughly fit observed ground-truth data. The task objective consisted of a linear combination of the cycle time Φ_{cycle} (Equation 3.5) and the contact force objective Φ_{contact} (Equation 3.10) with force setpoint F_{goal}. Optimization was performed for values of F_{goal} of 3, 4, 5, 6 and 7 N, collecting 250 optimized parameterizations for each goal force. `Vel` and `Acc` were initialized randomly. In total, 1,250 optimized programs were executed on a FANUC LR Mate 200iD/7L manipulator and FS-15iA force-torque sensor[1] to measure the resulting ground-truth trajectories.

The results are shown in Figure 3.2. The optimized parameterizations result in maximum contact forces very close to the desired F_{goal}, with average remaining deviations of 0.60 N from the force setpoint, an average improvement by 62 % over the initial parameterizations. Note that optimization for different target forces does not require retraining of the model. Figure 3.2 (top right) illustrates the convergence behavior of SPI for a target force of 5 N. SPI converges on an

---

[1] FANUC, Oshino-mura, Japan

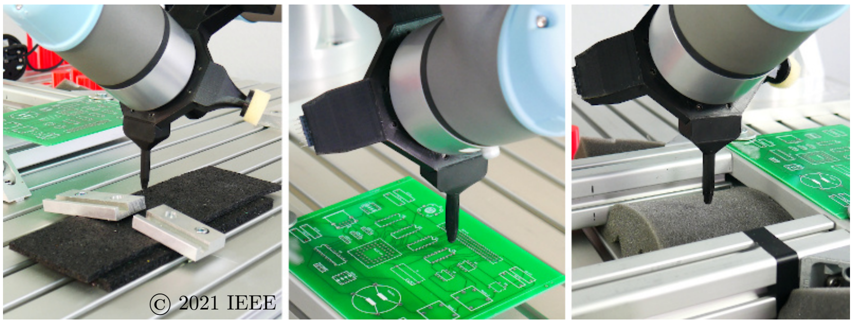


F_{goal}	3 N	4 N	5 N	6 N	7 N
ARTM	0.69 -75%	0.54 -69%	0.54 -51%	0.38 -64%	0.77 -51%

Figure 3.2: Optimization of force-controlled contact with SPI (Alt et al., 2021). Bottom left: Mean deviation from F_{goal} over 250 optimizations and improvement over the initial parameterization. Right: Convergence behavior (top) of SPI for the velocity parameter and resulting force trajectories (bottom) for $F_{\text{goal}} = 5\ N$ for a linear motion ARTM skill.

optimal velocity in under 40 iterations regardless of the initial parameterization. Figure 3.2 (bottom right) plots the force trajectories resulting from executing the 250 optimized parameterizations against 250 baseline trajectories in which the F_{goal} parameter was set to 5 N, but velocities and accelerations were chosen at random. Note that the resulting contact forces range between 3 and 10 N, while the optimized parameterizations result in contact forces centered around 5 N with considerably reduced variance.

Generalization to different source program representations In a second series of experiments, SPI is used to parameterize low-level primitives with respect to three different surfaces (PCB, rubber and foam) with damping characteristics ranging from near-linear (foam) to highly nonlinear (rubber). The target pose, velocity, and acceleration parameters of a `movel` URScript primitive (Universal Robots, 2018b) as well as the temporal scaling parameter τ and the target pose of a linear discrete DMP (Ijspeert et al., 2013) are optimized respectively to achieve the target force F_{goal}. For each combination of surface and skill type, a shadow program is pretrained on 5,000 simulated trajectories and finetuned on 500 real-world executions. For each combination of primitive, surface, and goal force, 100 optimizations are performed


© 2021 IEEE

	Rubber			PCB			Foam		
F_{goal}	5 N	10 N	20 N	1 N	5 N	8 N	1 N	1.5 N	2 N
URScript	1.43	1.63	2.76	0.16	0.68	0.95	0.24	0.15	0.16
	-75%	-84%	-84%	-92%	-71%	-80%	-36%	-51%	-74%
DMP	0.56	0.65	2.55	0.14	0.18	0.26	0.16	0.21	0.17
	-96%	-94%	-85%	-90%	-93%	-95%	-60%	-54%	-78%

Table 3.1: Optimization of contact motions for different skill frameworks and surfaces (Alt et al., 2021): Mean deviation from F_{goal} over 100 optimizations from random initial parameters (top, in N) and improvement of this error over the initial parameterization (bottom).

from randomly initialized parameters. A total of 1800 trajectories are collected on a UR5e collaborative manipulator.[2]

The results are shown in Table 3.1. The optimized parameterizations result in deviations below 0.25 N for most surfaces. Note that neither `movel` nor the DMP are force-controlled, and the achieved precision in reaching the target force is exclusively due to the optimization of program parameters by SPI.

3.1.3.2 Optimization of Force-Controlled Spiral Search

Like contact motions, search is a fundamental component of many industrial robotics tasks. Most production processes are subject to process noise, which may arise due to imprecise positioning of workpieces, sensor or actor inaccuracies, wear and tear, and a wide range of other sources. Search skills use real-time sensor data, such as tactile information from a force-torque sensor, to find workpiece features at runtime. A force-controlled spiral search skill, for example, conducts a spiral motion while exerting constant force against a surface to find a hole. When a sufficiently large deviation in the end-effector pose along the pushing direction is detected, the skill execution stops and the search is deemed successful. Search skills trade off search time for increased robustness. Like contact motions, they constitute bottlenecks in industrial robot processes, as they can incur considerable cycle time penalties,

---

[2] Universal Robots A/S, Odense, Denmark

particularly if the robustness requirements are very high. The manual optimization of search skills is challenging, as the underlying noise distribution is unknown to the human programmer and often at submillimeter scale. The following experiment tests the hypothesis that SPI can automatically optimize force-controlled spiral search skills to optimally resolve the trade-off between cycle time and robustness.

Experiment setup An electronics assembly scenario is considered, in which a robot is tasked to place THT components onto a PCB. Due to inaccurate positioning systems, the position of the holes on the PCB is subject to stochastic variation. Moreover, the pins of the THT components may be slightly bent within the manufacturer's tolerances. A robot program consisting of a linear approach motion (a `Move Linear` ARTM skill) followed by a `Spiral Search Relative` skill is used to reliably place each component. `Move Linear` accepts a target pose `PointTo` as well as a velocity and acceleration, and performs a motion linear in Cartesian space to `PointTo`. `Spiral Search Relative` executes a force-controlled spiral search motion starting from the current end-effector pose. It accepts parameters parameters `MinForce`, `MaxForce`, `MinDepth`, `MaxDepth`, `ExtentsX`, `ExtentsY`, `PathIncrement` as well as velocity `Vel` and acceleration `Acc`. `ExtentsX` and `ExtentsY` determine the size of the spiral along its principal axes, in mm; `PathIncrement` determines the distance between spiral arms, and therefore indirectly the number of windings of the spiral. During search, a force controller ensures that a force between `MinForce` and `MaxForce` is exerted against the surface. If the deviation along the pushing direction lies between `MinDepth` and `MaxDepth`, the search terminates and is considered successful.

A shadow program is pretrained on 50,000 simulated trajectories and finetuned on 2,500 real-world executions. The real-world experiment setup is shown in Figure 3.5. A FANUC LR Mate 200iD/7L industrial manipulator and FS-15iA force-torque sensor[3] are used. Two test datasets of 250 samples are collected, respectively. The first dataset ($\mathcal{D}_{\text{random}}$) contains input parameters are randomly initialized from a range deemed reasonable by a robot programming expert; the second dataset ($\mathcal{D}_{\text{expert}}$) contains an "oracle" parameterization manually finetuned by a human programming expert. For both real-world training and test data collection, the variance in the position of the hole is simulated by offsetting the start pose of the robot by random 2D offsets sampled from a unimodal Gaussian distribution. Program parameters are optimized with respect to cycle time Φ_{cycle} (Equation 3.5), task success Φ_{fail} (Equation 3.6), and path length Φ_{path} (Equation 3.7) individually, as well as the combined objectives $\Phi_{\text{fail}} + \Phi_{\text{path}}$ and $\Phi_{\text{fail}} + \Phi_{\text{cycle}}$. $\Phi_{\text{path}} + \Phi_{\text{cycle}}$ was not considered, as omitting Φ_{fail} led to degenerate results that were not suitable for any application – the objective of a search is, after all, to successfully find the searched feature.

---

[3] FANUC, Oshino-mura, Japan

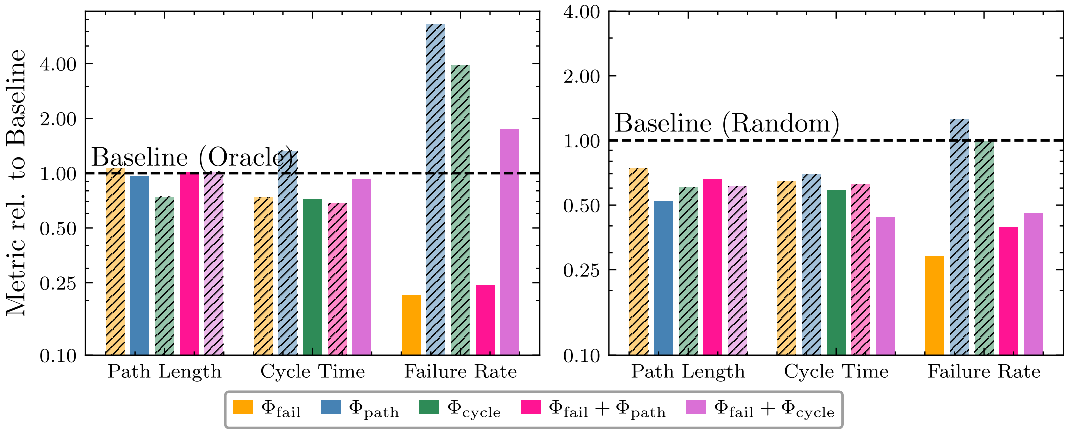


Figure 3.3: Empirical process metrics for different SPI task objectives Φ, relative to oracle (human expert, left) and random (right) baselines. Hatched bars indicate metrics that are not represented by, and in some cases run counter to, the task objective Φ (Alt et al., 2021).

Results The results are shown in Figure 3.3. For each task objective, the mean path lengths, cycle times and failure rates were computed across the collected real-world executions. Compared to the “oracle” parameterization by a human expert, SPI achieved considerable improvement in cycle time and failure rates, with improvements by 31% and 79% respectively. With respect to path length, gains were more moderate (26% reduction). As expected, optimization with respect to task objectives comprising Φ_{fail} effectively reduced failure rates, while other task objectives, most notably the exclusive optimization with respect to path length or cycle time, were detrimental to robustness and led to increases in failure rates by factors 6 and 4, respectively. This illustrates the inherent conflict between the robustness objective Φ_{fail} and the efficiency objectives Φ_{path} and Φ_{cycle} – robustness requires slower, more fine-grained search in a larger search area. In Figure 3.3, metrics that do not represent the task objective, such as the path length for Φ_{fail}, or the cycle time for Φ_{path}, are hatched. The reduction in path length realized by optimizing Φ_{cycle} illustrates that cycle time and path length correlate, as shorter spirals also terminate sooner. The failure of SPI to jointly optimize $\Phi_{\text{fail}} + \Phi_{\text{cycle}}$ indicates that optimization of mutually counteracting task objectives may not result in programs that are superior with respect to both metrics. Compared to the random baseline, SPI achieves considerable improvements for all task objectives with respect to the corresponding metrics, with reductions by 48 % in path length, 71 % in failure rate and 56 % in cycle time.

Figure 3.4 illustrates the evolution of spiral search trajectories over the course of 250 SPI iterations for the five considered task objectives. Optimization with respect to Φ_{fail} yields trajectories that cover almost the complete hole distribution, while task objectives Φ_{cycle} and Φ_{path} produce shorter spirals, as expected. They do not cover the hole distribution, as evidenced by high failure rates (see Figure 3.3).

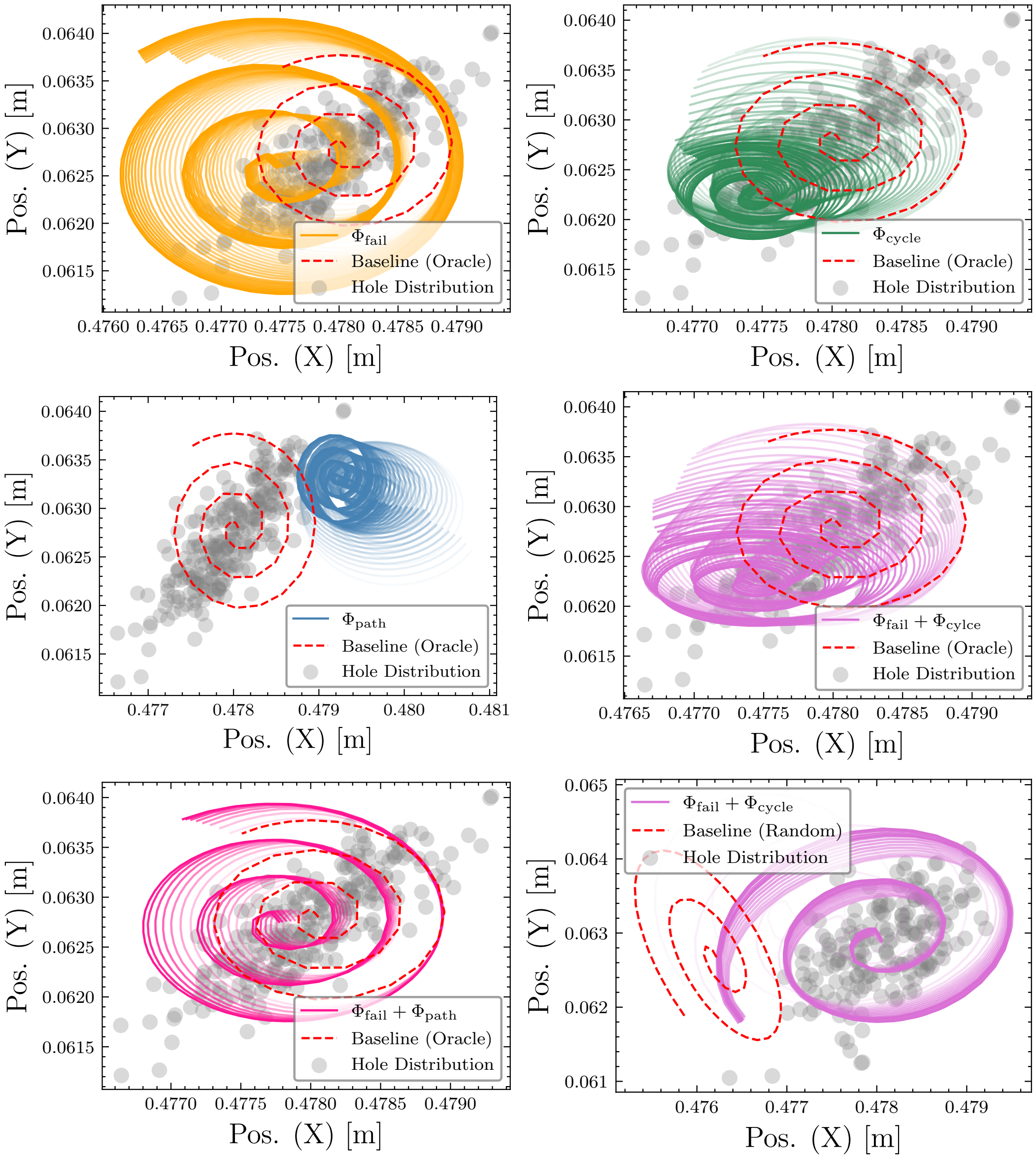


Figure 3.4: Spiral search motions optimized by SPI for different task objectives Φ (Alt et al., 2021). Higher opacity indicates later SPI iterations.

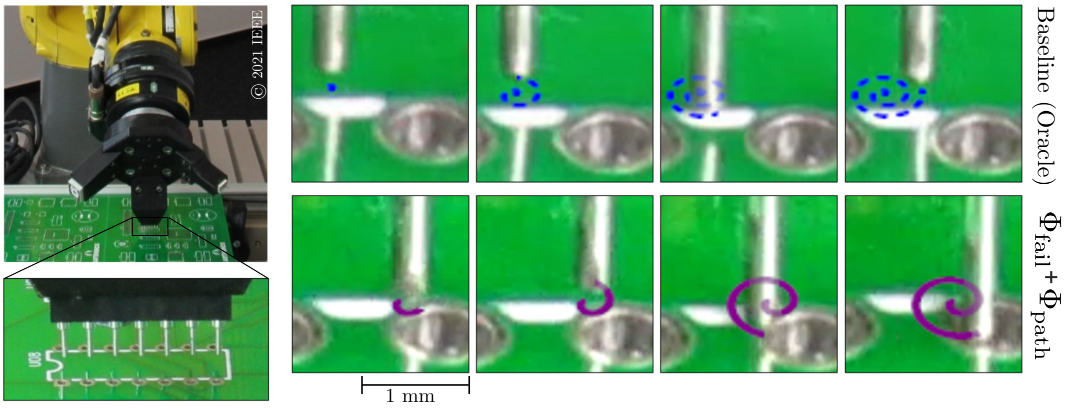


Figure 3.5: Exemplary baseline (top) and optimized (bottom) spiral trajectories. In line with the task objective (minimization of failure rate and path length), the optimized spiral is shorter, and begins closer to the mean position of the hole (Alt et al., 2021).

This is, however, intended behavior: Without including task success in the task objectives, SPI cannot be expected to produce search motions that find the sought feature. When included in the task objective, Φ_{fail} acts as a regularizer, ensuring that the functional requirement of finding the hole is respected.

3.1.3.3 Household Fetch-and-Place With VR Human Demonstrations

Particularly in the domain of service robotics, inferring program parameters from human demonstrations of a task avoids manual programming and parameter tweaking, and permits intuitive human-robot interaction and teaching. SPI supports optimization of robot programs with respect to human demonstrations both at the task as well as the motion level. This section introduces a dedicated task objective Φ_{demo}, which causes SPI to adapt program parameters to cause the posterior trajectory to approximate a given demonstration trajectory. Section 3.3.2 treats a human demonstration as a motion-level constraint on the DGPMP2-ND differentiable motion planner.

Given a demonstrated trajectory $\check{\boldsymbol{\theta}}$, a task-level demonstration cost can be defined as the sum of the pointwise Euclidean and quaternion distances between positions and orientations on $\boldsymbol{\theta}^{\bar{P}}$ and $\check{\boldsymbol{\theta}}$. If $\check{\boldsymbol{\theta}}$ contains a gripper state, e.g. when the demonstration was recorded in VR and includes the distance between thumb and index finger, Φ_{demo} may also include the pointwise distances between the demonstrated and predicted gripper states:

Human ▷ demonstration objective

$$\begin{aligned}\Phi_{\text{demo}}(\boldsymbol{\theta}^{\bar{P}},\check{\boldsymbol{\theta}}) = \sum_{i=1}^{|\boldsymbol{\theta}^{\bar{P}}|} \Big(\|\boldsymbol{\theta}^{\bar{P}}_{i,\text{pos}} - \check{\boldsymbol{\theta}}_{i,\text{pos}}\|_2 + d_q(\boldsymbol{\theta}^{\bar{P}}_{i,\text{ori}}, \check{\boldsymbol{\theta}}_{i,\text{ori}}) \\ + d_g(\boldsymbol{\theta}^{\bar{P}}_{i,\text{gripper}}, \check{\boldsymbol{\theta}}_{i,\text{gripper}})\Big), \end{aligned} \tag{3.12}$$

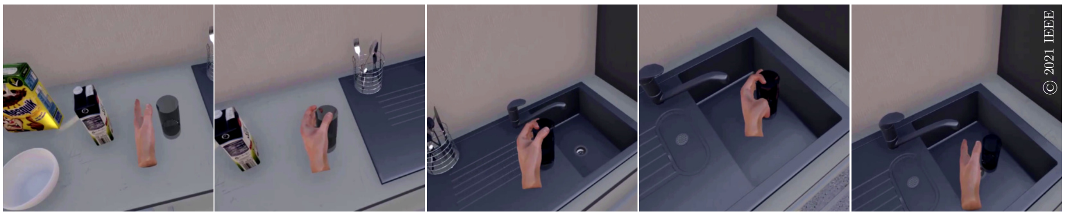


Figure 3.6: VR human demonstration for a household pick-and-place task.

where d_q denotes the quaternion distance (Equation 3.8) and $\boldsymbol{\theta}_{i,\text{gripper}}$ the gripper state at the ith point on $\boldsymbol{\theta}$. The distance function d_g is dependent on the gripper state. For the parallel gripper used in this experiment, $\boldsymbol{\theta}_{i,\text{gripper}}$ is a real number denoting the Cartesian gripper opening in mm and d_g is the mean absolute error (MAE). Note that this implementation of Φ_{demo} requires $\boldsymbol{\theta}^{\bar{P}}$ and $\check{\boldsymbol{\theta}}$ to have the same number of points. As the neural networks in $\bar{P}$ have been trained to predict trajectories of fixed length $M^{\bar{P}}$, padding $\check{\boldsymbol{\theta}}$ to length $M^{\bar{P}}$ avoids the need for retraining.

Experiment setup This experiment tests the hypothesis that SPI can automatically infer parameters for a multi-skill robot program given a single VR demonstration of the task. A household assistance task is considered, in which a glass is picked up and deposited in a sink. The source program P is an ARTM program consisting of of a linear approach motion (a `Move Linear` ARTM skill), a motion `Open Gripper` to open the gripper, a sequence of three `Move Linear` transfer motions, a `Close Gripper` skill, and a `Move Linear` depart motion. Besides the `PointTo`, `Vel` and `Acc` parameters of the `Move Linear` motions, `Open Gripper` and `Close Gripper` each have parameters `GoalState`, the target gripper configuration, and `Vel`, the gripper velocity, for a total of 19 symbolic parameters represented as a subsymbolic parameter vector $\boldsymbol{x}^P$ of length 49. For the parallel gripper used in this experiment, `GoalState` and `Vel` are scalars, corresponding to the gripper opening in mm and $\frac{\text{mm}}{\text{s}}$. Four VR human demonstrations were recorded in the Unreal VR environment of the KnowRob framework (Haidu and Beetz, 2021), in which a human picks up a cup and deposits it in a sink (see Figure 3.6).

The state space $\mathcal{S}$ consists of Cartesian poses as well as a one-dimensional gripper state, denoting the gripper opening in mm. Both the posterior trajectories $\boldsymbol{\theta}^{\bar{P}}$ predicted by $\bar{P}$ and the demonstrations $\check{\boldsymbol{\theta}}$ are trajectories in $\mathcal{S}$. The “gripper opening” in $\check{\boldsymbol{\theta}}$ is computed from the distance between the thumb and index fingertips of the hand of the VR avatar.

As learning is not required for this task, the shadow program $\bar{P}$ for P does not contain any neural adapters. Parameter optimization is performed purely via SPI given a single human demonstration. The task objective

Multicriterial demonstration objective ▷

$$\begin{aligned}\Phi(\boldsymbol{\theta}^{\bar{P}}, \check{\boldsymbol{\theta}}) = {} & w_{\text{pos}}\Phi_{\text{pos}}(\boldsymbol{\theta}^{\bar{P}}, \check{\boldsymbol{\theta}}) \\ & + w_{\text{ori}}\Phi_{\text{ori}}(\boldsymbol{\theta}^{\bar{P}}, \check{\boldsymbol{\theta}}) \\ & + w_{\text{gripper}}\Phi_{\text{gripper}}(\boldsymbol{\theta}^{\bar{P}}, \check{\boldsymbol{\theta}}) \\ & + w_{\text{goal}}\Phi_{\text{goal}}(\boldsymbol{\theta}^{\bar{P}}, \check{\boldsymbol{\theta}}),\end{aligned} \tag{3.13}$$

where Φ_{pos} is the MSE between the position of the VR avatar's hand and the predicted end-effector position, Φ_{gripper} is the MAE between the demonstrated hand openings and predicted Cartesian gripper openings, and Φ_{goal} is the MAE between the demonstrated and predicted and-effector positions at the end of the trajectory. Φ_{ori} is the mean quaternion distance (see Equation 3.8) between the demonstrated and predicted end-effector orientations.

This is a challenging multidimensional optimization problem, which is further complicated by the difference in dynamics between the human demonstration and the predicted trajectory. SPI must first optimize the velocity and acceleration parameters of all skills before criteria such as grasp pose accuracy can be meaningfully optimized. To make the optimization more tractable, the demonstrations were split into three segments corresponding to pick, transfer and place actions. Splitting of trajectories was performed based on force-dynamic events, such as contact of the hand with the cup (see Section 4.1.2.4). ARTM subprograms for pick, transfer and place were optimized separately. The program parameters are initialized randomly. The optimized robot programs are executed on a UR5 collaborative manipulator[4] and a Robotiq 2FG-85 parallel gripper.[5]

Results The task could be solved successfully for each of the four demonstrations, indicating that SPI can optimize robot program parameters with respect to human demonstrations. Initial, demonstrated and optimized trajectories for one of the demonstrations are shown in Figure 3.7. SPI generated parameterizations resulting in a robot trajectory that approximates the human demonstration, subject to prior constraints imposed by the semantics of the robot skills in the source program. The linearity constraint of the `Move Linear` transfer motions causes the most visible deviations from the demonstrated trajectory, which is not subject to any constraints. Likewise, the gripper can only be opened or closed during the `Open Gripper` skills in the source program, making it impossible for SPI to further reduce the demonstration error.

Figure 3.8 plots the demonstrated and optimized trajectories over the course of iterative optimization. Figure 3.9 shows the real-world execution of the optimized source program. As SPI optimizes the program parameters of NRPs, human

---

[4] Universal Robots A/S, Odense, Denmark

[5] Robotiq Inc., Lévis, Canada

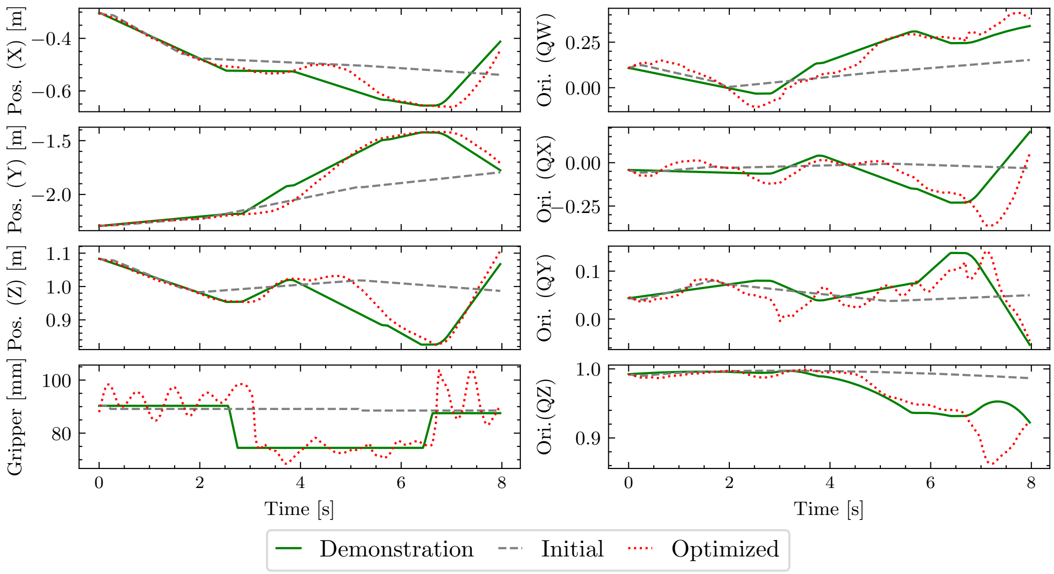


Figure 3.7: Optimization of pick-and-place program parameters with respect to a human VR demonstration. Subject to the (e.g. linearity) constraints imposed by the robot skills, the robot trajectory resulting from execution of the optimized parameterization (green) approximates the demonstration (red).

programmers can make post-hoc adjustments to the robot program via their accustomed user interfaces and programming workflows. In this case, it was demonstrated that the optimized source program could be trivially adapted to different pick-up poses by manually overriding the `PointTo` parameter of the approach motion for grasping. The adapted program was validated on three different pick-up poses on the real-world robot setup.

While this experiment demonstrates the ability of SPI to optimize robot program parameters with respect to human demonstrations, this experiment makes several simplifying assumptions that will be relaxed in further experiments. First, it assumes faithful approximation of the demonstrated trajectory as a proxy for task success. Chapter 4 introduces a program synthesis system for NRPs that parses human demonstrations at a semantic level, and generates robot programs that perform the tasks the user intended, rather than the motions the user demonstrated. Second, it does not consider collision-freeness during optimization, implicitly requiring the VR environment to be identical to the real-world execution environment. Section 3.3 introduces an extended version of SPI that explicitly considers the optimization of robot program parameters subject to collision constraints.

3.1.4 Related Work

SPI is a model-based first-order optimizer for robot program parameters. As such, a discussion of its merits and limitation requires contextualization within the family of

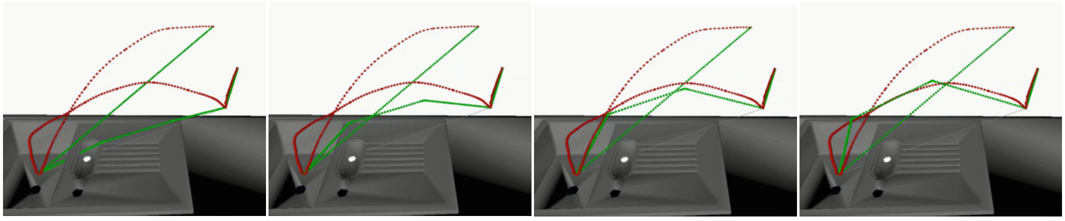


Figure 3.8: Posterior trajectory (green) during optimization of pick-and-place program parameters with respect to a human VR demonstration (red).

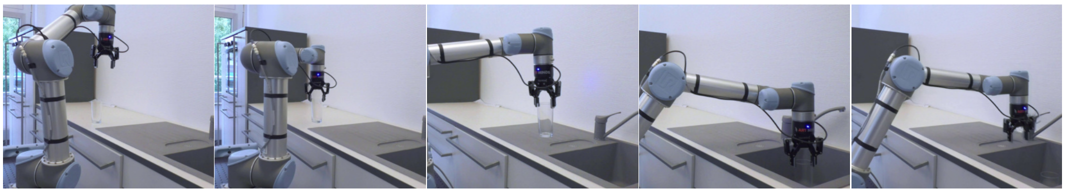


Figure 3.9: Real-world execution of a fetch-and-place task optimized with respect to a VR human demonstration.

first-order model-based optimization approaches, as well as within the application area of program parameter optimization.

3.1.4.1 First-Order Model-Based Optimization

The idea of optimizing a system's parameters by first-order optimization over a differentiable forward model of the system has been proposed as a solution to challenging optimization problems in a wide range of applications. First-order optimization algorithms leverage gradient information to direct the optimizer towards local optima of the objective function in an iterative manner (Lan, 2020). Examples for general-purpose first-order optimization algorithms include the Gauss-Newton (Magreñán and Argyros, 2018), Levenberg-Marquardt (LM) (Levenberg, 1944) and Frank-Wolfe algorithms (Frank and Wolfe, 1956), which are all realizations of this pattern in different forms. Beck (2017) provides an overview of gradient-based optimization.

Optimization over differentiable surrogates In engineering disciplines, first-order optimization is generally performed over some differentiable model of a real-world system. The use of such *differentiable surrogates* enables the use of gradient-based optimizers for systems which are not differentiable themselves. By virtue of their ability to approximate complex real-world phenomena to a high level of fidelity in a purely data-driven manner (Hornik, 1991), neural networks are particularly suitable general-purpose representations for first-order optimization. Neural surrogates have been used in conjunction with a wide range of first-order optimization methods in application domains such as astronomy (Himes et al., 2020), aeronautics (Liao et al., 2021) or robotics (Behl et al., 2020) to enable first-order optimization over hard-to-model real-world systems. Holeňa et al. (2010) provide an overview over surrogate-based optimization in materials science, where neural surrogates are used to avoid the need for real-world measurements, which may be prohibitively costly. (Lai, 2022) employs a surrogate CNN to avoid the need for computationally expensive molecular dynamics simulations to estimate antibody viscosities in a clinical context. In robotics, neural surrogates have been used primarily in modeling real-world phenomena that are particularly difficult to simulate, such as the deformation behavior of soft objects (Zhong et al., 2006; Wu et al., 2023b). SPI applies neural surrogates to robot program parameter optimization by leveraging a (partly) neural surrogate – the shadow program – to optimize robot program parameters with respect to a function of the expected real-world robot behavior.

Neural Network Iterative Inversion At its core, SPI solves an *inverse problem*: Computing the set of parameters $\boldsymbol{x}^*$ which cause optimal robot behavior with respect to task objectives Φ (see Equation 3.2). It solves this problem by approximating the robot program P with a surrogate DCG $\bar{P}$, and performing gradient descent in the input space of $\bar{P}$. Methodologically, SPI is a generalization of Neural Network Iterative Inversion (NNII) (Hoskins et al., 1992), which proposes to solve challenging inverse problems by learning a neural network to represent the function to be inverted, freezing its weights after training, and "inverting" the network by iterative search in the input space (Jensen et al., 1999). Williams, Linden and Kinderman (WLK) inversion is a straightforward formulation of NNII that performs inversion by backpropagation and SGD. Since the original formulation by Williams (1986) and Linden and Kindermann (1989), several variants of NNII have been proposed. WLK inversion only finds a single local solution to the inverse problem, which may not correspond to the true global inverse; alternative, multi-element inversion techniques employ e.g. evolutionary algorithms to find the global inverse by simultaneous, distributed search (Jensen et al., 1999). Wong and Kolter (2017) propose to substitute SGD by interpreting the neural network as a set of non-convex constraints, casting the inversion problem as constrained optimization and solving it via the Alternating Direction Method of Multipliers (Boyd et al., 2011). They report higher precision and faster convergence compared to SGD-based inversion. Ardizzone et al. (2018) propose Invertible Neural Networks, a neural network architecture which is invertible by virtue of enforcing a bijection between inputs and outputs, which permits the tractable approximation of the true posterior of the inverse. NNII has been applied to domains relevant to robotics such as optimal control (Hoskins et al., 1992), but most saliently to the computation of IK for learned robot geometries. Martin and Millán (1997) train a neural network to approximate the forward kinematics of a robotic manipulator, and use NNII in conjunction with RL to implement a $\mathcal{C}$-space controller. Raible et al. (2023b) train a DNN to approximate the kinematics of an industrial robot manipulator subject to real-world geometric errors caused by deformation under load as well as wear and tear, and use NNII to compute compensatory joint offsets. SPI is a generalization of WLK inversion to complex DCGs, combining gradient-based network inversion with differentiable programming to invert a differentiable function represented by multiple chained neural networks and other differentiable modules. To my knowledge, SPI is the first application of gradient-based model inversion to robot program parameter optimization.

3.1.4.2 Robot Program Parameter Optimization

The ubiquity of parameterized robot program parameterizations (see Section 2.6) motivated the development of algorithms for the automatic optimization of program parameters. The proposed approaches can be divided into gradient-free and gradient-based approaches, which are detailed below.

Gradient-free optimizers Due to the fact that robot programs effect changes in the physical world, a majority of robot program parameter optimizers require repeated executions of the robot program on real-world or simulated environments to iteratively approximate an optimal parameterization. As gradient information is typically not available, these approaches are generally gradient-free. Parameter optimizers based on evolutionary algorithms have been proposed for path planning (Liang et al., 2017), legged locomotion (Chernova and Veloso, 2004; Urieli et al., 2011; Kulk and Welsh, 2011), contact-rich manipulation (Marvel et al., 2009), or optimization of program parameters in domains beyond robotics (Wu et al., 2015; Sohn et al., 2016). In evolutionary approaches, an individual typically represents a candidate parameterization, changes to parameterizations are introduced at each generation subject to a given set of mutation and reproduction rules, and candidates are selected at each generation to maximize a fitness function encoding the task objectives. Like RL-based robot learning approaches, the convergence properties of evolutionary algorithms depend to a large extent on the fitness function. Moreover, the evaluation of the fitness function requires the execution of the robot program, which may incur significant overhead, particularly in real-world environments. Vollmer and Hemion (2018) propose to avoid the need for explicit specification of the fitness function by directly incorporating human feedback on candidate program executions. In a similar spirit, Racca et al. (2020) propose an active learning approach, in which the robot executes candidate parameterizations, receives human feedback, and computes the next candidate parameterization to maximize the expected information gained from the next trial. To further reduce the amount of required program executions, Berkenkamp et al. (2023) propose an approach based on Bayesian optimization, which has been empirically shown to be highly sample-efficient (Srinivas et al., 2010; Bull, 2011) without requiring gradient information, while still permitting global optimization (Mockus, 1989). Bayesian optimization has been used for parameter optimization in the context of legged locomotion

(Lizotte et al., 2007; Calandra et al., 2014), manipulation (Akrour et al., 2017) and feedback control (Marco-Valle, 2020).

First-order optimizers First-order optimization promises faster convergence, as gradient information is exploited to direct search (Loshchilov and Hutter, 2022). A large family of parameter optimization approaches folds parameter optimization into learning, and directly learn optimal parameters for a given task. The majority of MP learning approaches follow this principle (see Section 2.6.2). Alternative approaches train an auxiliary model such as a neural network, e.g. via RL or LfD, to directly output optimal program parameters. Zhou et al. (2020) train a Mixture Density Network from human demonstrations to predict optimal parameters for movement primitives. Kumar et al. (2024) learn a *parameter policy*, which generates optimal skill parameters for a given robot state, and is trained by exploration in an RL-like manner.

A different family of approaches avoid learning optimal solutions to the parameter optimization problem, and instead learn a suitable forward model of real-world robot actions that affords downstream optimization. One considerable advantage of such approaches is that the task objectives are not required to be known at training time, avoiding the need for expensive retraining when the task objectives change. Methodologically, several approaches have leveraged differentiable programming, model inversion or combinations of both in order to avoid repeated executions of candidate parameterizations. Differentiable programming has been used in conjunction with differentiable physics simulators to optimize the parameters of low-level motion controllers (Toussaint et al., 2018; Degrave et al., 2019; Hu et al., 2019a; Hu et al., 2019b; Qiao et al., 2020; Jatavallabhula et al., 2023). A methodologically related line of work exploits differentiable programming together with learned, typically neural, models to optimize the parameters of robot kinematics or dynamics models (Haug, 2021; Meier et al., 2022), particularly for manipulation problems in which real-world robot kinematics and dynamics differ from prior models due to a priori unknown phenomena such as wear and tear or payload-induced deformations (Raible et al., 2023b).

SPI performs first-order optimization over a learned forward model of a robot program. It is, to my knowledge, the first approach to optimize robot program parameters via gradient-based optimization over a DCG that combines differentiable kinematics and planners with learned neural modules.

3.1.5 Discussion

SPI solves a complex multicriterial optimization problem using a deceptively simple algorithm. The algorithmic simplicity is achieved by relegating the lion's share of the complexity to the NRP program representation: NRPs combine differentiable planners and neural networks in a modular fashion to represent real-world, long-horizon, sensor-adaptive robot programs as DCGs, permitting the use of standard automatic differentiation tools and first-order optimizers to perform parameter optimization in a straightforward manner. SPI is the first model- and gradient-based optimizer for robot program parameters that uses a partially learned, differentiable program surrogate. Like other model-based optimizers, it avoids repeated execution of candidate parameterizations at runtime, as the objective function is evaluated on a learned forward model of the expected real-world program execution. As a corollary, the task objectives are not required to be known at training time, and re-optimizing program parameters upon changing task objectives is performed in a zero-shot manner over the same learned forward model. An additional consequence of performing optimization over a learned surrogate, rather than e.g. a simulator, is that the gap between the model used for optimization and reality will be comparatively small; SPI optimizes program parameters with respect to a function of the posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$, the expected real-world behavior of the robot, as learned from real-world data. Like other first-order optimizers, SPI uses gradient information to steer the search, allowing for fast convergence even in high-dimensional input spaces.

SPI *jointly* optimizes all robot program parameters $\boldsymbol{x}^{\bar{P}}$, or arbitrary subsets of $\boldsymbol{x}^{\bar{P}}$. At each iteration, SPI performs a forward pass through the entire shadow program $\bar{P}$, and the task objectives Φ are evaluated for the complete trajectory. Long-horizon manipulation tasks typically have inter-skill dependencies, e.g. when the start state of an insertion skill depends on the last state of a preceding search skill, or when the way in which an object is grasped determines the feasibility of downstream manipulations. SPI respects such inter-skill dependencies by design: If the parameterization $\boldsymbol{x}^{\bar{p}_i}$ of an upstream skill $\bar{p}_i$ influences the resulting trajectory of some downstream skill $\bar{p}_j$, this will be reflected in the gradients $\frac{\partial \Phi(\boldsymbol{\theta}^{\bar{p}_j})}{\partial \boldsymbol{x}^{\bar{p}_i}}$, and the inputs of $\bar{p}_i$ will be adjusted to optimize e.g. the success probability of $\bar{p}_j$. In Experiment 3.1.3.2, the parameters of an approach motion are optimized to maximize the success probability of a downstream search. Joint cross-skill parameter optimization naturally arises as a consequence of the core operating principle of SPI – iterative optimization over a forward model of a robot program.

In addition to the mentioned advantages, SPI has two additional properties that make it particularly suitable for application in industrial and service robotics, respectively. First, like the underlying NRP program representation, SPI is agnostic with respect to the source program representation and can be used to optimize most skill-based robot program representations (see Experiment 3.1.3.1). This makes SPI particularly suitable for industrial applications, where a wide variety of manufacturer-specific programming languages are used. Second, SPI is agnostic with respect to the task objective, provided the task objective can be expressed as a differentiable function of the robot trajectory. The task objectives introduced in Section 3.1.2 are merely examples. In future work, the use of neural networks as task objectives will be investigated. Neural objective functions can facilitate the learning of complex task objectives in a manner similar to inverse RL (Adams et al., 2022).

Like most first-order optimizers, SPI performs *local optimization*, which may result in convergence on a local minimum of Φ. Premature convergence is a limitation common to all local optimizers, and can be addressed by a variety of methods. SPI uses the Adam optimizer, which employs learning rate decay with momentum to avoid premature convergence (Kingma and Ba, 2015). Loshchilov and Hutter (2022) proposes to combine momentum-based learning rate decay with "warm restarts", periodically resetting the learning rate to a higher value. Prakash et al. (2024) propose to combine warm restarts with Bayesian optimization for efficient global first-order optimization. As SPI does not place a restriction on the optimization algorithm itself, Adam can be replaced by a global optimizer without loss of generality.

A more fundamental limitation of SPI lies in the fact that as a model-based optimizer, the quality of learned forward model imposes upper bound on the quality of the optimization results. Unlike model-free optimizers, which rely on repeated executions of the robot program to evaluate the objective function during optimization, SPI relies on the surrogate model's faithful representation of real-world robot dynamics. In application domains with limited data availability, such as high-mix, low-volume manufacturing, or in service robotics domains with highly dynamic environments, the shadow program may not represent robot-environment dynamics with sufficient accuracy, causing SPI to optimize program parameters with respect to an inaccurate model of reality. This limitation can be overcome by employing state-of-the-art data-efficient machine learning models that combine data efficiency with high learning capacity. Recent work has explored replacing the GRU-based model architecture of NRPs with multimodal Transformer networks, adding a vision modality to enable faithful prediction of robot trajectories in dynamic environments (Kienle et al., 2024). Section 3.2 introduces a lifelong learning approach, which ensures that the shadow program is continuously updated to reflect current ground-truth robot-environment dynamics. The astute reader may note that SPI implicitly assumes that the environment is stationary, i.e. that the environment at optimization is drawn from the same distribution as the environment at training time. The lifelong learning mechanism introduced in Section 3.2 allows to relax this assumption to a degree, and permits the optimization of robot programs in the face of nonstationary environments.

3.2 Lifelong Learning and Optimization in Stochastic Environments

One of the foundational narratives of the field of robotics is the promise to automate repetitive and redundant tasks. Research and development in industrial robotics has advanced to the point that a wide variety of tasks can be automated with robots; in some industries, such as automotive manufacturing, some tasks such as welding or chassis assembly have been nearly completely automated. Other industries and tasks, however, have not achieved comparable rates of robotic automation. While the benefits of robots – uninterrupted around-the-clock production, increased robustness and standardized quality, and decreased need for increasingly rare, highly specialized workers – apply to equal measure for tasks such as electronics assembly, which, in the automotive industry, remains a largely manual process. Consider the wiring of a car center console, which requires connecting user controls – buttons, dials, or touchscreens – to the car's mechatronic and infotainment busses. This requires plugging tens of connectors attached to a wire harness into the respective sockets on the back of the center console. Currently, this task is typically performed by humans, despite the availability of multiple robots on the same assembly line. The central reason for the comparable lack of automation for this particular task, and similar tasks like it, is the presence of *stochastic process variances*. Section 3.1 motivates the use of search strategies to compensate process variances via tactile sensing, and demonstrates that SPI can leverage process data to optimize search parameters to minimize cycle time and maximize the probability of task success (see Experiment 3.1.3.2). However, SPI implicitly assumes that the process noise distribution at training time is *stationary*, i.e. that the noise distribution remains unchanged over time – and that the noise distribution does not change between training and evaluation of the NRP. For many complex real-world production tasks, this assumption does not hold. In the case of center console assembly, for example, both the center console and the cable connectors are positioned via external effectors, typically conveyor belts and feeder systems. The positioning of the center console, may be subject to drifts or sudden shifts between executions; moreover, components are often sourced from different suppliers, causing e.g. the geometry of connectors to vary suddenly between successive batches, causing nonstationary variances in the pose of the connector in the gripper. In high-precision applications, nonstationary process variances may be due to temperature variations, causing submillimeter deformations in the links and changing the viscosity of lubricants in the joints, resulting in nonstationary, constantly changing deviations from modeled kinematics and dynamics. Over long time horizons, wear and tear or mechanical defects may have similar effects, degrading a robot's absolute positioning accuracy over time (Raible et al., 2023b). To deploy SPI in real-world environments, the problems of efficiently training NRPs to reflect stochasticity in the environment, as well as keeping NRPs up-to-date with nonstationary noise processes, must be solved.

The study of SPI in nonstationary, stochastic environments presented in this section was first presented in Alt et al. (2022b). This section provides a revised and more detailed account.

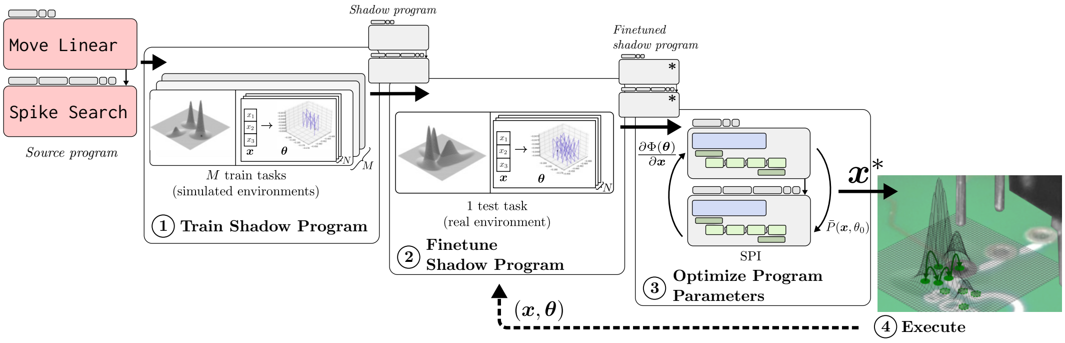


Figure 3.10: Closed-loop SPI for lifelong robot program parameter optimization in nonstationary environments (Alt et al., 2022b).

3.2.1 Data-Efficient NRP Learning in Stochastic Environments

3.2.1.1 Stochastic Environments

Chapter 2 models robot trajectories as stochastic processes $\{\Theta_t\}$, that associate a probability distribution over robot states θ with each timestep t. The law of $\{\Theta_t\}$ is parameterized by $\boldsymbol{x}^P$ and is conditional on θ_0^P as well as the current environment H. Chapter 2 left H unspecified. For the purpose of this Section, H is defined to be a *probability distribution over task-relevant features*. For the task of center console wiring, for example, H may be a probability distribution over the Cartesian workspace poses of sockets on the center console, and the pose of the grasped connector relative to the TCP. In general terms, H denotes the probability distribution from which the environment is sampled. In the context of the applications considered in the present section, in which robot programs are executed repeatedly over time, H may be thought of as drawn from a stochastic process $\{H_k\}$, which associates an environment distribution with every program execution k. In the context of center console assembly, the probability distribution over the socket poses varies over time according to $\{H_k\}$: For the $k_0{}^{\text{th}}$ program execution, the socket poses are distributed according to H_{k_0}. No assumptions about the law of $\{H_k\}$ are made – in particular, $\{H_k\}$ is not assumed to be Gaussian.

◁ *Environment distribution*

3.2.1.2 Sequential Transfer Learning of NRPs in Stochastic Environments

Transfer learning studies how AI systems can leverage information learned in the context of some task or domain to improve their performance in other tasks or domains, or solve unseen tasks altogether (Zhuang et al., 2021). *Sequential transfer learning* approaches pretrain machine learning (ML) models on a large source

◁ *Sequential transfer learning*

dataset[6] collected over a wide range of tasks, and finetune it at runtime on a much smaller dataset representing the task at hand (Ruder, 2019).

Alt et al. (2022b) propose an extension to SPI that leverages sequential transfer learning to keep model parameters up to date with changing environment distributions, permitting continuous re-optimization of robot program parameters over the lifetime of a production cell. By recasting the NRP learning phase of SPI as a transfer learning problem, the shadow program is pretrained offline on a large dataset representing a range of environment distributions, and finetuned at runtime on a much smaller dataset containing samples of the current environment. To that end, the definition of a *task* as “an ordered ensemble of [skills that] depicts a concrete representation of steps in a workflow to solve a specific goal” (Pantano et al., 2022) put forth in Chapter 2 is extended to be contingent on the environment. For the purpose of this Section, then, a task $\mathcal{T}$ associates robot behavior with a stochastic environment $\{H_k\}$. Formally, a task $\mathcal{T}$ is an *indexed conditional stochastic process* with primary process $\{H_k\}$, describing the environment at the kth program execution, and secondary process $\{\Theta_{k,t}\}$ describing the corresponding robot trajectory over t timesteps. $\{\Theta_{k,t}\}$ is governed by some latent probability distribution

$$\Theta_{k,t}|H_k \sim P_\Theta(\cdot|H_k), \tag{3.14}$$

where $P_\Theta(\cdot|H_k)$ denotes the conditional trajectory distribution $\Theta_{k,t}$ for environment H_k. In this framework, the NRP learning problem consists of learning this latent distribution.

For a given task $\mathcal{T}_0$, the corresponding *task dataset*

$$\mathcal{D}_{\mathcal{T}_0} = \{(\boldsymbol{x}_0^P, \theta_{0,0}^P, \boldsymbol{\theta}_0^P), ..., (\boldsymbol{x}_N^P, \theta_{0,N}^P, \boldsymbol{\theta}_N^P)\}$$

contains N sampled program executions for different program parameters and start states, with trajectories $\boldsymbol{\theta}_n^P$ realizations of $\mathcal{T}_0$; i.e., all trajectories were executed in environments whose task-relevant features are distributed according to probability distributions generated by the same stochastic process $\{H_k\}^{\mathcal{T}_0}$. The real-world datasets used for training NRPs in Chapter 2 and Section 3.1 are task datasets for a single task $\mathcal{T}_0$; the NRPs for the experiments in Section 3.1.3 were pretrained in simulation on a different task $\mathcal{T}_1$, which is identical to $\mathcal{T}_0$ except in the ways the simulated environment differs from reality. Sequential transfer learning proposes to pretrain ML models on a large variety of tasks, to facilitate efficient finetuning on a given task at hand (Ruder, 2019). To that end, the shadow program $\bar{P}$ is pretrained on a large *source dataset* $\mathcal{D}_S = \bigcup_{m=0}^{M} \mathcal{D}_{\mathcal{T}_m}$ containing executions of P in a variety of environments. $\mathcal{D}_S$ may contain executions of P on different robots, with different workpiece geometries, or, as it is a common occurrence in industrial

Source dataset ▷

---

[6] The terms *source dataset* and *source task* are unrelated to the concept of a *source program*, the symbolic part of the dual NRP representation. In line with established terminology in transfer learning, a source task forms part of the set of tasks on which a ML model is pretrained (the *source domain*), and a source dataset contains data representing several source tasks (Zhuang et al., 2021).

robotics, for different configurations of the hardware in a robot workcell. $\bar{P}$ is then finetuned on the *target dataset* $\mathcal{D}_T = \mathcal{D}_{\mathcal{T}_{\text{curr}}}$ containing only data sampled from the current task $\mathcal{T}_{\text{curr}}$. In the case of assembly-line industrial production processes, for example, $\mathcal{D}_T$ contains data from the last N production cycles. Depending on the quality of the pretraining dataset, the amount of training data in $\mathcal{D}_T$ can be very small – the experiments in Section 3.2.3 finetune on 128 samples of $\mathcal{T}_{\text{curr}}$. Moreover, if $\mathcal{D}_S$ contains pretraining examples with a sufficiently diverse range of program inputs, $\mathcal{D}_T$ may contain only program executions with one single set of program parameters, while still generalizing sufficiently well to new parameterizations during execution. Echoing the literature on sequential finetuning, lower learning rates during finetuning improve convergence and permit the model to retain priors learned during pretraining (Mosbach et al., 2020). In Experiment 3.2.3.1, the proposed sequential transfer learning scheme is evaluated on a real-world search-based peg-in-hole application in the presence of stationary process noise.

◁ *Target dataset*

3.2.2 Lifelong NRP Learning and Optimization in Nonstationary Environments

SPI as introduced in Section 3.1 implicitly assumes $\{H_k\}$ to be stationary, i.e. that environments at successive program executions are drawn from the same probability distribution. In the case of center console assembly, this means that the distribution underlying the spatial noise in the pose of the sockets is stationary – and that the offline optimization of e.g. a `Spiral Search Relative` skill, which adapts search parameters to optimally fit the process noise distribution, yields skill parameters which remain optimal for subsequent executions of the program. The realities of non-stationary sources of process noise, such as drifts due to thermal expansion or shifts due to supplier changes, imply that program parameters optimized at time k may be outdated at the next program execution $k + 1$.

To keep the shadow model up-to-date with nonstationary process noise, the proposed sequential transfer learning scheme introduced in Alt et al. (2022b) and described above can be modified to form a closed-loop, lifelong learning system. Figure 3.10 provides an overview of the proposed system. After pretraining on a large source dataset $\mathcal{D}_S$, program parameters are optimized via SPI and the source program P is executed on the robot. The resulting trajectory is recorded and added to the task dataset dataset $\mathcal{D}_T$. After $\mathcal{D}_T$ has reached a sufficient size (e.g. 128 samples in Experiment 3.2.3.3), the NRP is finetuned on $\mathcal{D}_T$, parameters are re-optimized and the resulting trajectory is added to $\mathcal{D}_T$. This cycle of finetuning and optimization is repeated for every subsequent iteration. As the environment is not assumed to be stationary, $\mathcal{D}_T$ is implemented as a fixed-size ring buffer, and outdated samples are eventually removed from the finetuning dataset. To prevent catastrophic forgetting of the priors learned during pretraining (Kirkpatrick et al., 2017), the original, pretrained model is finetuned from scratch at each iteration.

3.2.3 Experiments

To empirically test the hypothesis that SPI with a sequential transfer learning scheme enables data-efficient optimization of robot program parameters in stochastic environments, a series of experiments is conducted. The experiments consider mechanical and electrical assembly tasks in simulated and real environments with stationary and nonstationary process noise.

3.2.3.1 Force-Controlled Spiral Search Optimization in Stationary Environments

A first experiment considers an industrial search-based insertion task (see Figure 3.11). SPI with sequential transfer learning is used to optimize `Spiral Search Relative` and approach motion parameters to minimize failure rate and cycle time on a real-world experiment setup (Alt et al., 2022b). The results are compared to a zero-order Non-dominated Sorting Genetic Algorithm (NSGA-II) baseline (Deb et al., 2002).

Experiment setup The task is shown in Figure 3.11 (right). A mechanical assembly task is considered, which requires inserting a steel cylinder into a hole with tolerances on the order of 1/100 mm. This process is representative for a class of manufacturing processes with tolerances exceed the robot's repeat accuracy, requiring a combination of force-sensitive search and zero-moment insertion. Under real-world conditions, the receptacle is subject to additional positioning errors, e.g. due to an imprecise conveyor belt. With sequential transfer learning, SPI promises to compensate this process noise by adapting the parameters of the search skill via first-order optimization over a forward model of the robot program, finetuned on a small sample of real-world program executions.

The source program for valve assembly is shown in Figure 3.11 (left). The ARTM source program representation is used (Schmidt-Rohr et al., 2013). The program consists of an approach motion (a `Move Linear` skill), followed by a spiral search motion starting at the current robot state (`Spiral Search Relative`) and an `Insert Moment` skill for zero-moment insertion of the cylinder. If `Spiral Search Relative` fails to find the hole, the program terminates and no insertion is performed. For a detailed description of `Spiral Search Relative` and its parameters, see Section 3.1.3.2. The task consists of optimizing program parameters to maximize the likelihood of successful insertion while minimizing cycle time under stationary process noise. To that end, the approach pose (the `PointTo` parameter of the approach motion) as well as the spiral extents (`ExtentsX` and `ExtentsY`), width between spiral arms (`PathIncrement`) and motion dynamics (`Vel` and `Acc`) are optimized. Note that the approach pose also determines the orientation of the spiral, i.e. the primary axes of the ellipse covered by the search.

In the context of this experiment, the environment distribution of task-relevant features $\{H_k\}$ represents the position in the XY-plane of the robot's workspace. In

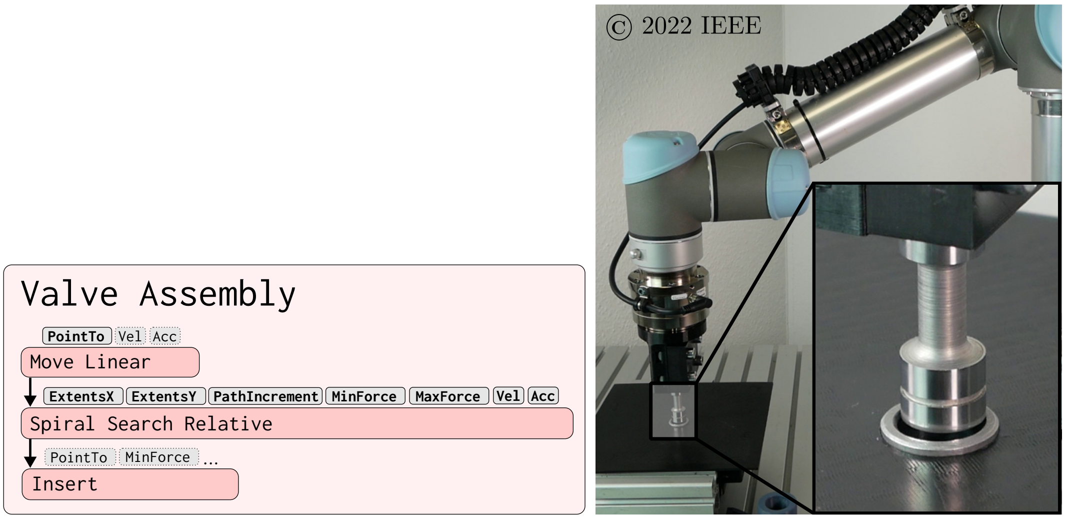


Figure 3.11: Robot program (left) and hardware setup (right) for search-based valve assembly (Alt et al., 2022b). Optimized robot program parameters are highlighted in bold font.

this experiment, $\{H_k\}$ is assumed stationary: For every program execution, the hole position is sampled from the same distribution H. To elicit a variety of possible noise distributions on the same hardware setup, process noise is modeled as a bivariate Gaussian mixture with one to six components, and is artificially added as an offset to robot motions.

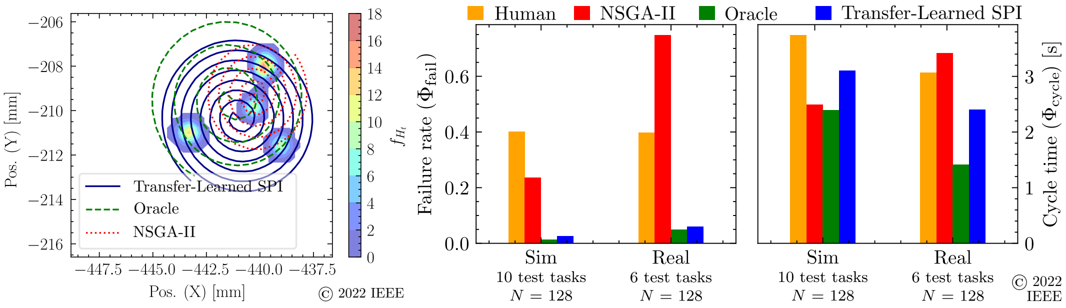
The shadow program is trained on a source dataset $\mathcal{D}_S$ consisting of a total of 128.000 program executions collected in 1000 simulated environments (training tasks), each with a different hole distribution. The simulation consists of a simple, scripted kinematics and dynamics model with naive Newtonian physics, including damping and friction. The shadow program is then finetuned on a target dataset $\mathcal{D}_T$ containing 128 program executions in the environment (test task) at hand. Unlike in Experiments 3.1.3.1 and 3.1.3.2, inputs are not required to be sampled from a region covering the optimization domain; rather, diversity in program inputs is only required in the source dataset, and all program executions in the task dataset are sampled with the same set of parameters. Parameter optimization is performed to minimize a linear combination of failure rate Φ_{fail} and cycle time Φ_{cycle} (see Equations 3.6 and 3.5). The experiment is repeated for 10 different simulated and 6 real-world test tasks on a UR5 manipulator[7] with an ATI Axia80 force-torque sensor.[8]

Baselines SPI is compared against a human expert parameterization, an application-specific heuristic and the zero-order NSGA-II optimizer.

---

[7] Universal Robots A/S, Odense, Denmark

[8] ATI Industrial Automation Inc., Apex, USA



(a) (b)

Figure 3.12: Optimization of spiral search parameters in the presence of stationary process noise (Alt et al., 2022b). (a) Exemplary spiral paths for NSGA-II, a handcrafted “oracle” heuristic and SPI for a multimodal Gaussian hole distribution. (b) Failure rates and cycle times for SPI as well as human expert, NSGA-II and oracle baselines on 10 and 6 simulated and real-world noise distributions (128 test executions per distribution).

1. **Human expert**: A fixed set of parameters considered suitable by a robot programming expert for the given application (peg-in-hole insertion with very low positional tolerances). No data-driven, empirical parameter optimization is performed.

2. **Oracle**: A task-specific heuristic that estimates optimal values for several program parameters using information about the ground-truth valve body poses. The orientation of the approach target pose (`PointTo` of `Move Linear`) is determined via Principal Component Analysis over the ground-truth valve body positions, aligning the principal axis of the spiral motion with the principal axis of the hole distribution. The spiral extents `ExtentsX` and `ExtentsY` are set to the widths of the distribution along the principal axes.

3. **NSGA-II** with hyperparameters $\mu = \lambda = 25$ (Deb et al., 2002). Suitable hyperparameters were determined by hyperparameter search in a simulated environment.

Results The results are shown in Figure 3.12. With respect to task success, SPI significantly outperforms both the human expert and NSGA-II baselines to result in near-zero failure rates ($\Phi_{\text{fail}} = 0.06$ v. 0.40 and 0.74 respectively on real-world test tasks; see Figure 3.12b, left). SPI is competitive with the “oracle” heuristic ($\Phi_{\text{fail}} = 0.05$) while avoiding need for ground-truth information about the hole distribution, which is not available in real-world applications, and without requiring task-specific adaptations. With respect to cycle time, SPI outperforms the human expert baseline and NSGA-II on real-world tasks ($\Phi_{\text{cycle}} = 2.41$ s v. 3.07 s and 3.42 s, respectively), but is outperformed by NSGA-II on simulated tasks.

Discussion The results highlight the benefits of data-driven approaches to parameter optimization. They replicate the results of Experiment 3.1.3.2 with only 128 training examples of the test task, and without requiring diverse inputs in the task dataset. As the training scheme is the only algorithmic difference between experiments 3.1.3.2 and 3.2.3.1, the results indicate considerable gains in data efficiency due to sequential transfer learning. The possibility to finetune shadow models on small task datasets permits pretrained shadow models to act as general-purpose foundation models for downstream tasks, enabling data-efficient generalization to task variants (here, different environment distributions). Data-efficient fine-tuning enables real-world use of SPI in commercial industrial applications, as it avoids costly stoppages of production for data collection. The poor performance of NSGA-II on real-world tasks may reflect the difficulty of ensuring reliable convergence of genetic algorithms, particularly when only few evaluations of the fitness function are possible. In this experiment, NSGA-II was allowed 250 program executions, 122 more than the 128 training examples provided to SPI. The inconsistent performance of NSGA-II motivates further experiments comparing the relative performance of gradient-based and gradient-free parameter optimizers.

3.2.3.2 Benchmarking Sequential Transfer Learning and Meta Learning

Besides sequential transfer learning, a wide range of alternative approaches for task generalization have been proposed. Meta learning is concerned with “learning to learn” by leveraging prior knowledge to adapt quickly to new tasks (Hospedales et al., 2021). Like sequential transfer learning, meta learning decomposes the learning problem into two stages; unlike sequential transfer learning, however, in which pretraining and finetuning solve the same learning problem on different datasets, meta learning approaches first train a learning system to effectively learn new tasks on a large, diverse *meta-training set*, and then use that learning system to learn the task at hand on a much smaller *meta-test set* containing only data for one specific task. Model-agnostic meta learning methods, which can be applied to learning systems without requiring modification of model architectures, can be applied directly to the learning stage of SPI (Finn et al., 2017a). As in the sequential transfer learning formulation proposed in Section 3.2, a task corresponds to goal-oriented robot behavior subject to a stochastic environment-generating process $\{H_k\}$. Interpreting the source dataset $\mathcal{D}_S$ as the meta-train dataset and the task dataset $\mathcal{D}_T$ as the meta-test dataset, the sequential transfer learning problem can be recast as a meta learning problem, in which the meta-training phase replaces the pretraining phase, and the weights of $\bar{P}$ are adjusted by meta-training over $\mathcal{D}_S$ to maximize the effectiveness of learning the current task from $\mathcal{D}_T$.

From the fact that meta learning approaches train learning systems to efficiently learn new tasks, then, follows the hypothesis that meta learning improves the data efficiency or performance of sequential transfer learning in the context of learning NRPs for peg-in-hole insertion tasks under uncertainty. To that end, the proposed sequential transfer learning scheme is compared with three model-agnostic meta learning approaches:

1. **Model-Agnostic Meta Learning (MAML):** MAML formulates meta learning as double-loop gradient descent, in which an inner loop minimizes a task objective $\mathcal{L}_{\text{task}}$, and an outer loop optimizes a meta-objective $\mathcal{L}_{\text{meta}}$ that represents the ability of the system to learn new tasks (Finn and Levine, 2017). At each training iteration, a task $\mathcal{T}$ is sampled from $\mathcal{D}_S$ and M steps of SGD with task-learning rate α are performed to update the weights $\phi^{\bar{P}}$ of $\bar{P}$ for the task-learning objective $\mathcal{L}_{\text{task}}$ (here the regular NRP training loss, see Equation 2.1). This task-learning stage results in a set of task-adapted weights $\tilde{\phi}^{\bar{P}}$. The meta-objective is then evaluated for the task-adapted model, and one step of gradient descent is performed to update the original parameters $\phi_{\bar{P}}$ of the model $\bar{P}$, before any task adaptation, with meta-learning rate β. The meta-learning objective $\mathcal{L}_{\text{meta}}$ is the sum of all N task-level losses. Note that MAML is a second-order learning algorithm: In the outer (meta-)learning loop, weights $\phi_{\bar{P}}$ are updated by gradient descent over the computational graph representing, in turn, M iterations of inner-loop gradient-based task learning. I refer to Finn and Levine (2017) for a detailed description of MAML.

2. **First-Order MAML (FOMAML):** MAML requires automatic differentiation of a nested loop, which is computationally expensive for large values of M (the number of task-learning iterations). FOMAML has been found to speed up MAML by roughly 33% on image classification tasks (Finn and Levine, 2017) FOMAML avoids computing the second-order gradient by avoiding differentiation through the inner-loop gradient update; rather, the gradient used for outer-loop (meta-)learning is simply the gradient of the meta-objective for the network parameter values after the last task-learning step. Nichol et al. (2018) and Finn and Levine (2017) describe FOMAML in greater detail.

3. **Reptile**: Reptile is a first-order model-agnostic meta-learning algorithm (Nichol et al., 2018). As in MAML and FOMAML, the task-learning step consists of M steps of SGD on the task-learning objective $\mathcal{L}_{\text{task}}$, yielding task-adapted weights $\tilde{\phi}^{\bar{P}}$; the meta-learning step updates the weights $\phi^{\bar{P}}$ of $\bar{P}$ along the direction $\tilde{\phi}^{\bar{P}} - \phi^{\bar{P}}$ with meta-learning rate β. Reptile has been found to perform similarly to FOMAML on several meta-learning benchmarks. I refer to Nichol et al. (2018) for further details.

Experiment setup The valve assembly task and source program illustrated in Figure 3.11 is considered. Given the source and task datasets $\mathcal{D}_S$ and $\mathcal{D}_T$ defined on page 115, NRP shadow programs are trained using sequential transfer learning

Learning algorithm	Traj. precision	Success acc.
Sequential transfer learning ($N = 128$)	**0.80**	**0.72**
MAML ($N = 5$)	0.95	0.64
FOMAML ($N = 5$)	0.96	0.62
FOMAML ($N = 128$)	0.99	0.58
Reptile ($N = 128$)	1.35	0.56

Table 3.2: Comparison of sequential transfer learning with meta learning alternatives for NRP training (Alt et al., 2022b).

as well as MAML, FOMAML and Reptile. For all meta-learning approaches, hyperparameters are set to $M = 5$, $\alpha = 0.01$, and $\beta = 0.001$. Models are trained for 50 epochs. As second-order MAML proved too memory intensive to train on the available hardware (a single NVIDIA 1080 Ti GPU), the inner-loop batch size N (the number of training examples per task) is reduced to 5 for MAML. For comparison, FOMAML is evaluated with both $N = 5$ and $N = 128$. The performance of the resulting models is evaluated on 10 simulated test tasks (stationary hole-distribution-generating processes, $\{H_k\}$) with 128 finetuning and 128 test samples each. All trained shadow programs are evaluated on two metrics: *Trajectory precision*, the MSE between predicted and ground-truth end-effector poses; and *success accuracy*, the proportion of correctly classified task success labels.

Results The results are shown in Table 3.2. Sequential transfer learning outperforms all meta learning alternatives, both with respect to the predicted end-effector positions as well as success labels. The comparison with MAML permits to draw few conclusions, as MAML only sees 5 examples during its inner-loop task adaptation phase, compared to the 128 finetuning examples for sequential transfer learning. However, increasing the inner-loop batch size to 128 did not improve the performance of FOMAML, indicating that the inner-loop batch size is not a performance bottleneck for FOMAML.

Discussion While meta learning has been considered highly promising for few-shot learning tasks (Hospedales et al., 2021; Wang et al., 2020; Sun et al., 2019), where the number of training examples per task is very small, this promise failed to manifest better training outcomes in the context of NRP learning in stochastic environments. Table 3.2 gives credence to the hypothesis that sequential transfer learning is a suitable alternative to meta learning in the medium-data regime, where few-shot learning is not required and it is possible to collect a small finetuning dataset. These findings are in line with recent empirical comparisons of meta learning and sequential transfer learning (Kolesnikov et al., 2020; Shysheya et al., 2022; Mandi et al., 2022; Patacchiola et al., 2023). In compute-constrained environments, sequential transfer learning is considerably more computationally

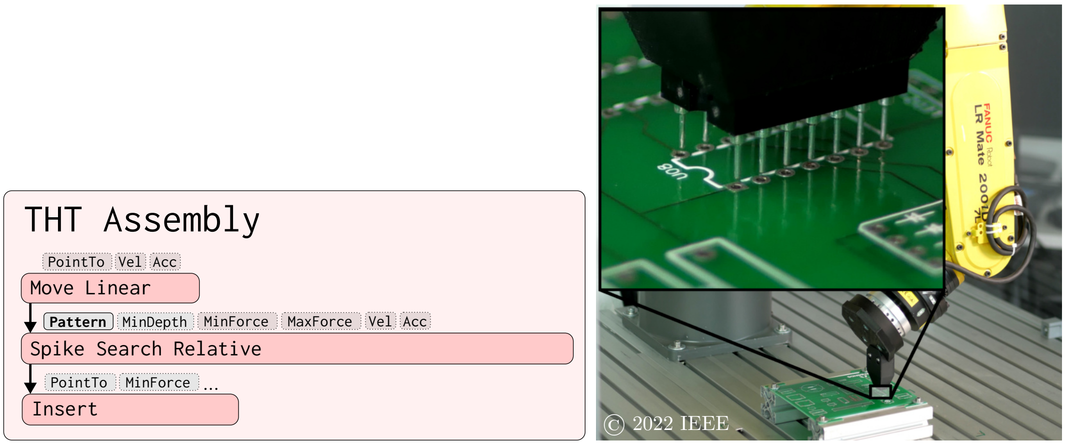


Figure 3.13: Robot program (left) and hardware setup (right) for THT assembly (Alt et al., 2022b).

efficient than meta learning, avoiding the additional accumulation of gradients during task adaptation. Additional experiments are required to provide empirical measurements of the performance of meta learning in the absence of such resource constraints, e.g. with larger inner-loop batch sizes, a larger, more diverse source dataset or empirical hyperparameter search.

3.2.3.3 Force-Controlled Probe Search Optimization in Nonstationary Environments

To determine to what extent the results of the above experiment generalize to a different task, more complex program structures and nonstationary environments, a PCB assembly task is considered (Alt et al., 2022b). The robot is tasked to insert a 14-pin socket into the corresponding holes on a PCB (see Figure 3.13, right). Reflecting phenomena commonly occurring in real-world production lines, the positioning of the PCB in the XY-plane is subject to nonstationary process noise, e.g. due to drift in imprecise positioning systems such as conveyor belts, wear and tear or sudden supplier changes.

Experiment setup The task is shown in Figure 3.13 (right). To compensate process noise and ensure robust insertion, a probe search strategy is used, in which the robot repeatedly touches the surface with the workpiece and a specified contact force until the workpiece drops in the hole. In the domain of THT assembly, probe search is the preferred strategy for placing components as it avoids lateral movement over the surface, preventing damage to the conducting surface of the PCB and avoiding bending the pins of the workpiece.

The robot program for search-based THT assembly is shown in Figure 3.13 (left). The ARTM source program representation is used (Schmidt-Rohr et al., 2013). The program consists of a linear approach motion (a `Move Linear` skill) followed by a probe search beginning at the current end-effector pose of the robot (`Spike Search Relative`) and insertion with zero side forces (`Insert`). For the purpose of this experiment, the `Points` parameter of `Spike Search Relative` is optimized, which defines the search pattern as a sequence of poses, relative to the start of the search. `Points` is initialized as a 4x4 grid in the XY plane, combined into a 32-dimensional parameter vector. The robot moving successively from the top-right to the bottom-left point of the grid. An optimal search pattern would reflect characteristics of the underlying hole distribution – notably, the first probes should be made in regions with high probability density, with later probes exploring lower-probability regions.

The noise of the hole position is modeled as bivariate Gaussian mixture with up to 6 modes. Unlike in 3.2.3.1, process noise is not assumed stationary. Instead, three different nonstationary noise processes $\{H_k\}$ are considered:

- **Drift:** At each timestep, the modes of $\{H_k\}$ are translated by a constant offset. This models process noise induced by wear and tear or thermal expansion.
- **Shift:** At each timestep, the modes of $\{H_k\}$ are translated by a uniformly random offset with probability $P_{\text{shift}} = 0.05$. This models sudden changes to the production process, such as slightly varying workpiece geometries between batches, or slight reconfigurations of the robot workcell during planned downtime.
- **Brownian motion:** At each timestep, the modes of $\{H_k\}$ are translated by an offset sampled from a bivariate Gaussian distribution. This models "chaotic", fast-changing drift-like effects.

SPI is trained on a source dataset $\mathcal{D}_S$ containing 128.000 program executions in 1000 different environments (training tasks). For each training task, a new hole-distribution-generating process $\{H_k\}$ is instantiated, and an environment distribution H is sampled before each program execution. Each task is repeated 100 times ($t = 1, 2, ..., 100$). For the real-world executions, the sampled hole offsets are artificially added as an offset to robot motions, permitting simulation of complex process noise on a static hardware setup. As in Experiment 3.2.3.1, diversity in program inputs is only required in the source dataset, and all program executions in the task dataset are sampled with the same set of parameters. As nonstationary noise processes are considered, the lifelong learning and continuous re-optimization scheme introduced in Section 3.2.2 is used. The task dataset $\mathcal{D}_T$ is implemented as a ring buffer of size 128, test executions are immediately added to the ring buffer and the original, pretrained model is finetuned at each iteration on the updated task dataset.

The experiment is repeated for 10 different simulated and 6 real-world test tasks on a Fanuc LR Mate 200iD/7L manipulator with an FS-15iA force-torque sensor.

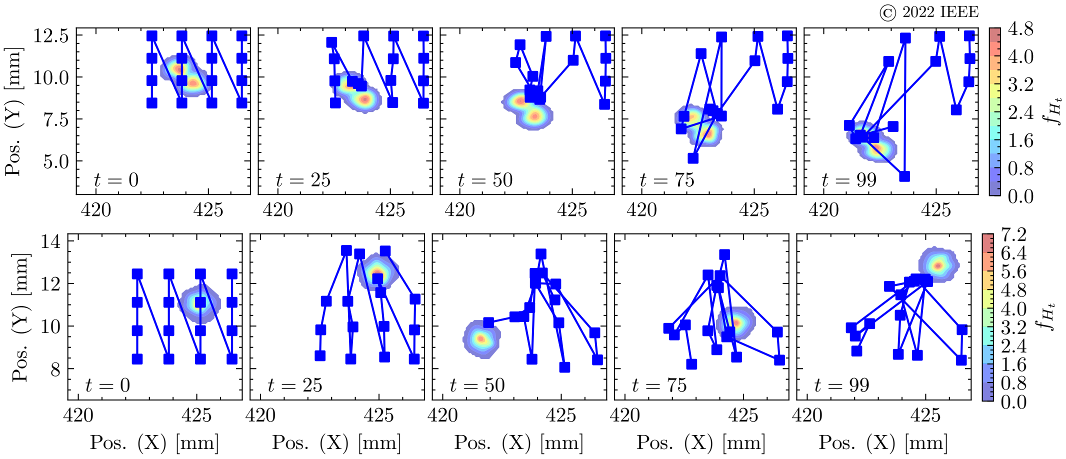
Baselines SPI with sequential transfer learning is compared against the following baselines (Alt et al., 2022b):

1. **Grid:** A fixed 4x4 grid covering the complete search region.
2. **Heuristic:** A task-specific heuristic, that fits a 16-mode GMM to those probe points in the finetuning dataset for which search was successful, and sets the optimized probe points to the resulting modes.
3. **NSGA-II** with hyperparameters $\mu = \lambda = 100$ for simulated tasks, and $\mu = \lambda = 30$ for real-world tasks (Deb et al., 2002).

Regularizers Optimization is performed to minimize failure rate Φ_{fail} (see Equation 3.6). When optimizing the search pattern, SPI inherently faces an exploration – exploitation trade-off: Parameters are optimized to fit the learned, implicit characteristics of the environment distribution, at the expense of those areas of the workspace that have seen few to no samples in the training data. With small task datasets, this can cause optimization results to overly emphasize workspace regions frequently present in the dataset, while ignoring regions of lower probability density. This is particularly salient for shift processes, in which the occurrence of a shift and the environment distribution after the shift cannot be extrapolated from the task dataset before the shift occurs. For this reason, two regularizers are investigated:

1. **L1 distance to initial grid:** $\Xi_{\text{init}}(\boldsymbol{x}^{\bar{p},j}) = \sum_{i=0} |\boldsymbol{x}_i^{\bar{p},j} - \boldsymbol{x}_i^{\bar{p},0}|$ penalizes candidate parameterizations $\boldsymbol{x}^{\bar{p},j}$ at SPI iteration j proportional to their L1 distance to the initial parameterization $\boldsymbol{x}^{\bar{p},0}$. In the context of search pattern optimization, this regularizer encourages the optimizer to keep optimized patterns close to the initial grid. Ξ_{init} is a general-purpose regularizer that can be applied to any application use case.
2. **Euclidean cross-distance between points:** $\Xi_{\text{cdist}} = \frac{1}{d_{\text{cross}}(\boldsymbol{x}^{\bar{p},j}, \boldsymbol{x}^{\bar{p},0})}$ penalizes candidate search patterns for which the sum of the Euclidean distances between all point pairs (d_{cross}) is small. In the context of probe search, it encourages the optimizer to distribute probe points and avoid clusters of probes in close proximity. Unlike Ξ_{init}, Ξ_{cdist} is specific to probe search, as it assumes $\boldsymbol{x}^{\bar{p}}$ to represent 2D Cartesian positions.

Optimization is performed with respect to $\Phi_{\text{fail}}(\boldsymbol{\theta}^{\bar{p},j})$ as well as the regularized objectives $\Phi_{\text{fail}}(\boldsymbol{\theta}^{\bar{p},j}) + \Xi_{\text{init}}(\boldsymbol{x}^{\bar{p},j})$ and $\Phi_{\text{fail}}(\boldsymbol{\theta}^{\bar{p},j}) + \Xi_{\text{cdist}}(\boldsymbol{x}^{\bar{p},j})$. Note that the regularized objectives not only score the current predicted trajectory $\boldsymbol{\theta}^{\bar{p},j}$ at the jth SPI iteration, but also the current candidate parameters $\boldsymbol{x}^{\bar{p},j}$.



© 2022 IEEE

Figure 3.14: Optimization of probe search patterns for nonstationary noise processes, here linear drift (top) and Brownian motion (bottom) (Alt et al., 2022b). The search pattern is optimized via SPI with lifelong learning to minimize the likelihood of task failure, subject to the Ξ_{init} regularizer.

Results Figure 3.14 shows exemplary search pattern evolutions produced by SPI with Ξ_{init} regularization for linear drift and Brownian motion noise processes. SPI continuously adapts the search pattern to reflect the underlying noise distribution, even in the presence of relatively fast-changing nonstationary processes.

The quantitative results are summarized in Table 3.3. Without optimization, probe search failed in more than 60% of search attempts across noise types. Without regularization, SPI reduces failure rates by 33%, 11% and 21% for drift, Brownian and shift noise processes, respectively. With Ξ_{init} regularization, improvements are lower in the presence of drift and Brownian noise (18% and 9%), but higher for shift processes (32%), indicating that the more conservative optimizer behavior induced by Ξ_{init} regularization may reflect in improved robustness only in the presence of a priori unpredictable process noise such as sudden shifts, which can only be compensated for after the fact; drift and Brownian motion processes have low-frequency stochastic properties which can be learned during finetuning and which remain valid for the duration of the process. SPI with Ξ_{cdist} regularization yields best results, reducing failure rates by 53%, 27% and 43%, respectively. This gives credence to the hypothesis that Ξ_{cdist} regularization strikes a balance between allowing the optimizer to produce search patterns which deviate from the original grid, while avoiding overfitting on the task dataset by tightly clustering probe points on the (implicitly learned) modes of the hole distribution.

Discussion The results give credence to the hypothesis that continuous, lifelong finetuning enables SPI to update program parameters with respect to nonstationary stochastic environments. The proposed continuous finetuning scheme updates the weights of the shadow program $\bar{P}$ based on a very low number of samples (here,

	Drift	Brownian	Shift
No optimization	0.679	0.679	0.600
SPI (Φ_{fail})	0.458	0.605	0.472
SPI ($\Phi_{\text{fail}} + \Xi_{\text{init}}$)	0.556	0.616	0.406
SPI ($\Phi_{\text{fail}} + \Xi_{\text{cdist}}$)	**0.318**	**0.494**	**0.339**

Table 3.3: Failure rates for different regularizers and stochastic processes (Alt et al., 2022b). SPI with the Ξ_{cdist} regularizer reduces failures by up to 53% over 100 timesteps.

128), while avoiding catastrophic overfitting. As the nonstationary environment changes at every iteration k, the model must generalize at every iteration, as the task dataset will lag behind the ground-truth process $\{H_k\}$ by one timestep. It is particularly notable that SPI adapts search pattern despite never being trained on an explicit representation of the environment distribution: The relationship between search pattern and task success is an implicit function of the environment distribution, and is implicitly represented in the learned weights of $\bar{P}$. Learning this relationship and inverting it via NNII gives rise to search patterns which reflect this latent representation. The ability of SPI to update program parameters with nonstationary noise processes enables its continuous use on long-running production lines, in which the underlying environment distributions will invariably change over time.

3.2.4 Related Work

SPI with sequential transfer learning addresses the fundamental problem of efficiently optimizing robot program parameters for noisy environments. More specifically, it addresses the situation in which the environment at optimization time follows a different distribution than the environment at (pre-)training time. Such *distributional shift* is a profound challenge for ML systems beyond robotics (Bansak et al., 2024). In the context of model-based optimization, the challenge is to ensure that the model accurately represents the environment distribution at optimization time. For neural ML models, the transfer learning, meta learning and zero-shot learning literature has brought forth a range of algorithmic and representational solutions to addressing distributional shift. This Section focuses on model-agnostic approaches that can be applied to existing model architectures such as the stacked recurrent architecture introduced in Section 2.4.1.4, without requiring changes to the architecture.

3.2.4.1 Transfer Learning

Transfer learning is a family of ML approaches designed to relax the assumption that training and test data share the same distribution. Transfer learning approaches center around learning priors from a large-scale source dataset in order to simplify, or make tractable, learning or optimization for downstream tasks following a previously unseen distribution (Abnar et al., 2021). Pan and Yang (2010) as well as Zhuang et al. (2021) provide comprehensive overviews of transfer learning.

Full finetuning Among the neural transfer learning approaches, sequential transfer learning, the method proposed and described in Section 3.2, is a *model-agnostic* technique for transfer learning which makes no assumptions about model architecture (Alt et al., 2022b; Ruder, 2019). This transfer learning approach finetunes all network parameters after initializing them by pretraining on a large source dataset. It has been used with considerable success in computer vision, where pretraining networks on the ImageNet dataset (Deng et al., 2009) has become a common method to bootstrap networks for downstream tasks in which little training data is available (Agrawal et al., 2014; Kolesnikov et al., 2020; Ridnik et al., 2021). The same paradigm has been successfully applied in other domains such as autonomous driving (Strudel et al., 2021; Zhang et al., 2022; Yuan et al., 2023) or robot-assisted surgery (Ross et al., 2018; Bodenstedt et al., 2020; Mateen et al., 2024).

Fixed feature extraction The most common alternative to finetuning the full set of model weights is to train or finetune only the last layers of a network on the target dataset, using a pretrained model as a *fixed feature extractor* (Krizhevsky et al., 2012; Kornblith et al., 2019). This strategy has been employed with particular success in larger network architectures such as Transformers (Vaswani et al., 2017), for which full finetuning may be prohibitively expensive. In Transformer-based encoder-decoder architectures, such as those underpinning robot VLA models, pretrained encoders are often used as-is, and only decoder networks are finetuned (Kienle et al., 2024; Ghosh et al., 2024; Driess et al., 2023).

Empirical findings In their seminal work, Yosinski et al. (2014) empirically investigate several finetuning strategies. They find that sequential transfer learning generally improves network performance on the target task. Their findings also suggest that transfer may be negatively impacted by “splitting” a pretrained network to adapt the last layers, as some layers may have “fragilely co-adapted”, spreading feature extraction across layers (Yosinski et al., 2014). Similarly, Kornblith et al. (2019) find strong empirical evidence that pretraining on large-scale source datasets considerably improves performance on downstream tasks under a transfer learning regime, echoing the results of Experiments 3.2.3.1 and 3.2.3.3. He et al. (2019b) find that, in the context of ImageNet pretraining, sequential transfer learning considerably speeds up convergence on downstream tasks, but does not impact final task performance. The degree to which their results transfer to the present context is limited, however, by the fact that they finetune on COCO (Lin et al., 2014), itself a large-scale dataset, unlike the very small task datasets considered here. In a large-scale meta study, Abnar et al. (2021) similarly find that while sequential transfer learning generally improves downstream performance, too high accuracy on the source dataset may reduce performance on downstream tasks due to saturation effects. In the present context, pretrained shadow programs typically do not perform well on their own, as the models are not conditioned on the environment distribution, but rather learn the distribution from the training data. The high diversity of environment distributions in the source datasets prevents models from overfitting on individual distributions during pretraining; rather, they learn general motion characteristics that widely apply across environment distributions, and precise performance for a given environment distribution is acquired during finetuning. The results of Experiments 3.2.3.1 and 3.2.3.3 give credence to this interpretation.

3.2.4.2 Meta Learning

Meta learning, or “learning-to-learn”, is concerned with enabling AI systems to learn effective learning algorithms from data (Thrun and Pratt, 1998). In the context of model-based optimization in the presence of distributional shifts, meta learning approaches promise to facilitate efficient fine-tuning on few examples of the distribution at hand by explicitly training AI systems to learn efficiently.

Model-agnostic meta learning A wide variety of model-agnostic meta learning algorithms have been proposed, which can be applied to near-arbitrary neural network architectures. Finn and Levine (2017) introduced MAML, a method that optimizes network parameters so that they can be quickly fine-tuned with a few gradient steps on new tasks. MAML approaches meta learning by double-loop gradient descent: In an inner loop, individual tasks are learned via SGD, while in an outer loop, SGD is performed over the inner-loop learning procedure, making MAML a second-order learning algorithm. In response to the considerable computational overhead of double-loop SGD, Finn and Levine (2017) propose FOMAML, a computationally efficient variant that approximates the meta-gradient by ignoring second-order derivatives, thereby reducing the computational overhead while still achieving competitive performance in rapid adaptation scenarios. Reptile (Nichol et al., 2018) simplifies the meta-learning process by performing multiple steps of SGD and then updating model parameters towards the final parameters obtained after these steps. Probabilistic MAML (PMAML), introduced by Finn et al. (2018), incorporates uncertainty estimation into the MAML framework, allowing for more robust adaptation in uncertain and dynamic environments commonly encountered in robotics. Instead of learning a set of model weights that can quickly be finetuned to fit new tasks, Andrychowicz et al. (2016) propose learning an optimizer itself, which can then be used to adapt models to new tasks with a similar structure than those in the source dataset.

Meta learning in robotics In robotics, model-agnostic meta learning methods have been applied in the context of learning locomotion policies (Kaushik et al., 2020; Gurumurthy et al., 2020), fault diagnosis of industrial robots (Liu et al., 2023) or learning from demonstrations (Hu et al., 2022). While meta learning approaches can be used to solve transfer learning problems, meta learning approaches have been widely used for few-shot learning, with task datasets containing only a very low, typically single-digit number of training examples (Finn et al., 2017b; Yu et al., 2018; Ghadirzadeh et al., 2021).

Empirical findings Experiment 3.2.3.2 applies and evaluates model-agnostic meta learning approaches for efficient learning of forward models of robot programs in stochastic environments. The results indicate that sequential transfer learning may be more effective for transfer learning problems with medium-sized, rather than few-shot, task datasets. This finding is in line with empirical findings in computer vision (Kolesnikov et al., 2020; Shysheya et al., 2022) as well as robot policy learning (Mandi et al., 2022; Patacchiola et al., 2023) indicating that “multi-task pretraining with finetuning on new tasks performs equally as well, or better, than meta-pretraining with meta test-time adaptation” (Mandi et al., 2022).

3.2.4.3 Zero-Shot Transfer

Recent advances in large Transformer-based architectures have given rise to an alternative approach to robot learning in the presence of stochastic environments, which eschews finetuning or meta learning in favor of zero-shot transfer without any examples of the task distribution.

VLA models VLA models such as RT-2 (Zitkovich et al., 2023), RoboFlamingo (Li et al., 2023b), Octo (Ghosh et al., 2024) or OpenVLA (Kim et al., 2024) combine token embeddings of vision, language and robot actions as input and output modalities and process them via a set of modality-specific and cross-modality Transformer-based encoder and decoder modules. The modality-specific modules are typically state-of-the-art vision or language models such as ViT (Dosovitskiy et al., 2020), Llama 2 (Touvron et al., 2023), DINOv2 (Oquab et al., 2023) or PaLM-E (Driess et al., 2023) trained on web-scale datasets. VLA models are often used as *foundation models* for downstream tasks, either as-is or as part of a larger neural architecture, typically without finetuning (Li et al., 2024). This is possible due to the astonishing zero-shot transfer abilities of large-scale Transformer-based architectures, which can generate highly specific predictions given a description of the target task as part of the context or prompt (Kojima et al., 2022). Zero-shot transfer via multimodal foundation models has been used in the context of robotics in applications ranging from navigation (Huang et al., 2023a; Shah et al., 2023) to task planning (Singh et al., 2023; Rana et al., 2023).

Finetuning foundation models If the target domain is highly specific and relatively far removed from the domains covered in the source dataset, foundation models may still be finetuned on domain- or task-specific data (Mosbach et al., 2020; Xu et al., 2023; Alt et al., 2024b), though this incurs considerable computational overhead and can be avoided in favor of Retrieval-Augmented Generation (RAG) or and similar prompt-based strategies in many applications (Chen et al., 2024). Kienle et al. (2024) pursue a hybrid strategy, finetuning a pretrained multimodal Transformer on robot- and application-specific data, but zero-shot generalize at runtime to variants of the finetuning task (e.g. differently colored cables for a cable insertion task). They evaluate their work in the context of SPI on search-based insertion tasks similar to those considered in Experiments 3.2.3.1 and 3.2.3.3 and constitute a direct evolution of the work introduced by Alt et al. (2022b) and presented in Section 3.2. However, large-scale pretrained Transformers have been shown to generate erroneous outputs, or “hallucinations”, particularly on zero-shot transfer tasks outside of their training distribution (Gunjal et al., 2024). Ensuring precision and reliable transfer in VLA models is a crucial direction of ongoing research (Xu et al., 2023).

3.2.5 Discussion

The proposed sequential transfer learning scheme addresses a the crucial issue of learning forward models of robot behavior in stochastically varying environments, with a focus on both stationary and nonstationary noise processes. Experiments in simulated and real-world industrial search-based peg-in-hole applications demonstrate the data efficiency of sequential transfer learning, allowing for the finetuning of forward models for previously unseen environment distributions on as little as 128 finetuning examples (see Experiment 3.2.3.1). Lifelong finetuning on a streaming dataset permits SPI to continuously re-optimize robot program parameters in the presence of nonstationary process noise (see Experiment 3.2.3.3). The improved data efficiency of sequential transfer learning is in line with empirical findings from the literature (Yosinski et al., 2014; Kornblith et al., 2019; Abnar et al., 2021), and allows for robot program parameter optimization in real-world industrial applications, in which only a limited amount of finetuning examples may be available. The pretraining and finetuning schedule integrates well into the development process of industrial robot cells, in which serial production is preceded by a ramp-up phase. During ramp-up, the assembly line is tested at reduced production volume, and small amounts of finetuning data can be collected (Alt et al., 2024a). During serial production, process data can be continuously recorded for lifelong finetuning and program re-optimization. The fact that sequential transfer learning alleviates the need for diverse program inputs in the task dataset further facilitates deployment during ramp-up, as making small changes to parameters during data collection may not be feasible for all production processes.

Experiment 3.2.3.3 demonstrates that continuous, lifelong finetuning enables SPI to continuously (re-)optimize program parameters with respect to nonstationary noise processes. In real-world production lines, this enables SPI to compensate low-frequency, long-horizon noise processes such as wear and tear (Raible et al., 2023b) as well as high-frequency, short-horizon disruptions such as supplier changes (Alt et al., 2022b). A ring-buffer size of 128 finetuning examples in the task dataset and finetuning the original model at each iteration appear to avoid both catastrophic forgetting and overfitting for the four-layer stacked GRU architecture of SPI shadow skills (see Section 2.4.1.4).

Zero-shot transfer to new environments or task variants is a promising direction of future work. The shadow program architecture presented in Chapter 2 does not accept a representation of the environment or task as an input; all information about environment or task is implicitly represented in the weights of the neural networks. Zero-shot transfer requires the model to accept some representation of the current task as inputs. Kienle et al. (2024) extend the NRP shadow program architecture by a tokenized image of the current environment, which is passed to the shadow program along with the program inputs and robot state. This avoids the need for continuous finetuning, provided all changes in the environment are visible in the image, and are drawn from a distribution close to those represented in the training data. The successive integration of multimodal foundation models, combined with additional pretraining on high-quality, domain-specific data and a rich representation of the current environment and task as part of the model's inputs, may permit shadow programs to zero-shot generalize across increasingly dynamic, stochastic environments.

3.3 Joint Optimization of Task Parameters and Motion Trajectories

For most manipulation tasks, task success requires not only achievement of high-level task objectives, but also the respect of a set of low-level constraints on the robot motions performed. Consider the robotic shopping assistant outlined in Chapter 1: When tasked to pick up an object on a shelf and placing it into a shopping basket, task success cannot be defined exclusively by whether the target object is in the shopping basket after the robot has performed the task, but must include additional criteria such as whether the robot collided with other objects, human customers or itself during task execution. For other applications such as industrial surface treatment tasks, smoothness of motion or minimization of higher-order dynamics may be crucial to task success. To optimize for task success in the presence of motion-level constraints, task parameters and motion trajectories must be jointly optimized.

Experiment 3.1.3.3 first demonstrated the need for respecting complex motion-level constraints during parameter optimization. In the experiment, a robot is tasked to transfer a cup into a sink. Robot program parameters are optimized via SPI with respect to an objective function derived from a VR human demonstration. As the source program consisted of grasp and put-down subprograms connected by

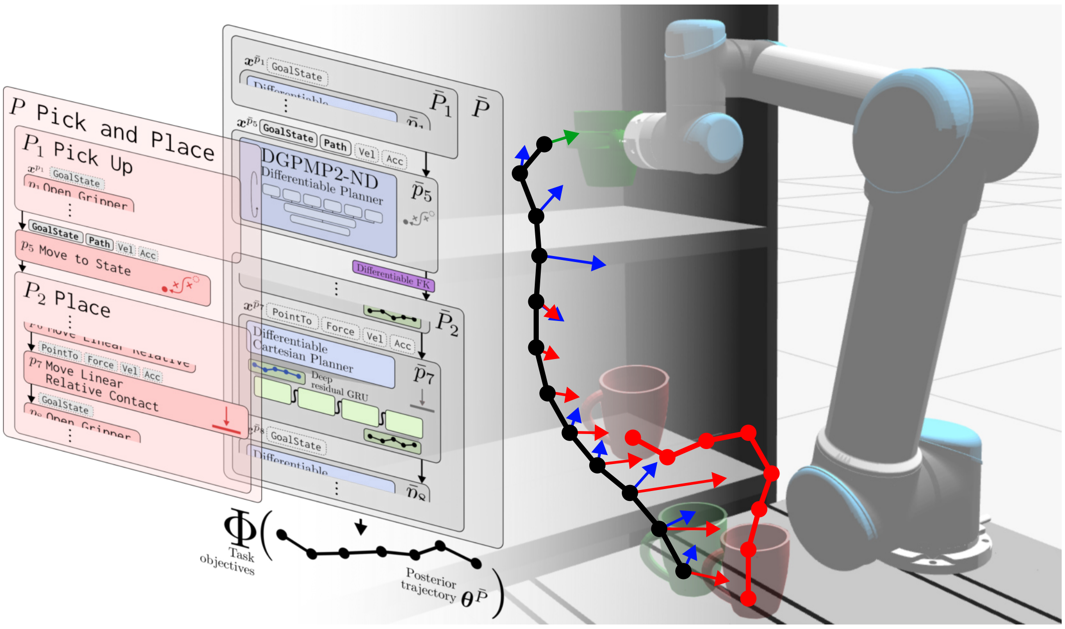


Figure 3.15: Shadow Program Inversion with Differentiable Planning (SPI-DP) permits the joint optimization of robot program parameters and motion trajectories by nested gradient-based optimization over a learned shadow program ($\bar{P}$, grey) to optimize both task- and motion-level objectives such as collision avoidance (blue), target pose accuracy (green), or proximity to a human demonstration (red).

a sequence of linear Cartesian transfer motions, the resulting motion trajectories were constrained to be linear in Cartesian space by design, and the optimization objective – proximity to a human demonstration – implicitly ensured that the target poses of the linear transfer motions were collision-free. There was no algorithmic guarantee, however, of collision-freeness of the overall motion, and the environment at optimization time was implicitly required to be identical to that in which the human demonstration was recorded. To provide such guarantees, and to support arbitrary environments at optimization time, it does not suffice for SPI to be a parameter optimizer; SPI must also be a motion planner.

3.3.1 First-Order Parameter and Trajectory Optimization with Differentiable Motion Planning

This section introduces SPI-DP, a first-order robot program optimizer capable of jointly optimizing robot program parameters and motion trajectories. It expands on work first published in Alt et al. (2025).

3.3.1.1 Optimizing Planned Motion Skills

SPI is a model-based robot program parameter optimizer. As such, it optimizes the parameters of parameterized robot skills – skills p for which the resulting robot behavior is a function of a set of parameters $\boldsymbol{x}^p$. The optimized parameters $\boldsymbol{x}^{*,\bar{p}}$ computed by SPI over shadow skill $\bar{p}$ are transferred back to p, and p is executed on the real robot with the optimized parameters to produce optimal behavior. If the resulting real-world robot trajectory $\boldsymbol{\theta}^p$ is completely specified by $\boldsymbol{x}^p$, as is the case for e.g. DMP skills, there is neither need nor opportunity for trajectory optimization. There is, however, a large class of skills, which may be termed *planned motion skills*, for which trajectory optimization is both possible and, in many applications, required. The `Move to Point` ARTM skill is an illustrative example. It accepts four parameters: A Cartesian target pose `PointTo`, velocity `Vel` and acceleration `Acc`, and an optional `Path` parameter representing a path planned by an external motion planner. If `Path` is not given, the ArtiMinds RPS will attempt to plan a collision-free path to `PointTo`. Similarly, the `Path Force` ARTM skill accepts a mandatory `Path` parameter containing a user-specified reference trajectory for a force controller as well as several controller parameters. `Move to Point` and `Path Force` represent a range of skills accepting a reference trajectory as an input parameter. Planned motion skills also include skills for which the reference trajectory is provided by kinesthetic teaching, a widespread class of skills in industrial robotics (Ajaykumar et al., 2021; Heimann and Guhl, 2020; Krot and Kutia, 2019). To optimize such skills in a data-driven way, the reference trajectory must, in many cases, be optimized along with the remainder of the skill (and often program) parameters. Consider a robot program to grasp an object; the grasp pose must be jointly optimized with the motion trajectory to approach the target object, to ensure that it is reachable and the approach is collision-free.

3.3.1.2 Shadow Program Inversion with Differentiable Planning

Section 2.5 introduces DGPMP2-ND, a differentiable motion planner based on first-order trajectory optimization with respect to motion-level objectives, or constraints, such as smoothness, collision-freeness, motion-level proximity to a human demonstration, adherence to joint limits, or precision at reaching Cartesian target poses. DGPMP2-ND is integrated into the shadow program DCG, acting as a differentiable prior which gives shadow programs the ability to bootstrap complex trajectories under Cartesian or $\mathcal{C}$-space constraints (see Figure 3.16). In the context of SPI, DGPMP2-ND enables the optimization of motion trajectories for planned motion skills, jointly with any other parameters of the containing program (see Figure 3.15).

Integration of DGPMP2-ND makes differentiable motion planning under collision, smoothness and other constraints part of the shadow program forward pass. Joint program parameter and motion trajectory optimization can then be performed by double-loop gradient descent over the shadow program DCG. The

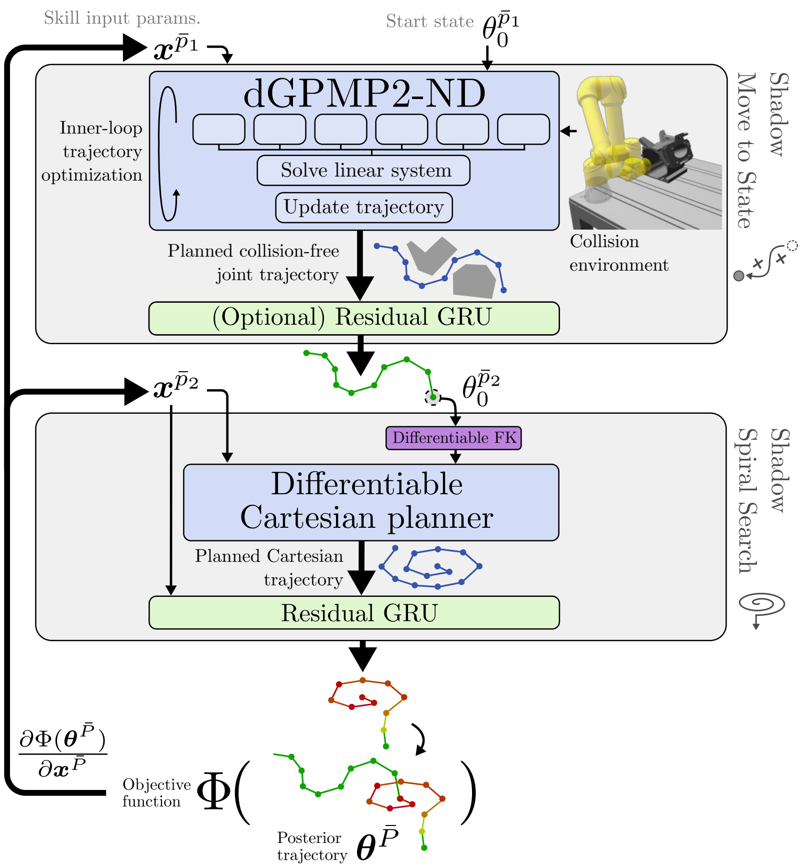

Skill input params.
$x^{\bar{p}_1}$
Start state $\theta_0^{\bar{p}_1}$
dGPMP2-ND
Inner-loop trajectory optimization
Solve linear system
Update trajectory
Shadow Move to State
Collision environment
Planned collision-free joint trajectory
(Optional) Residual GRU
$x^{\bar{p}_2}$
$\theta_0^{\bar{p}_2}$
Differentiable FK
Differentiable Cartesian planner
Shadow Spiral Search
Planned Cartesian trajectory
Residual GRU
$\frac{\partial \Phi(\theta^{\bar{P}})}{\partial x^{\bar{P}}}$
Objective function
$\Phi($ $)$
Posterior trajectory $\theta^{\bar{P}}$

Figure 3.16: SPI-DP performs joint parameter and trajectory optimization by double-loop gradient descent over the shadow program DCG.

SPI-DP ▷ resulting second-order optimizer is called SPI-DP. It is illustrated in Figure 3.16 at the example of a planned motion skill (`Move to Point`) followed by a spiral search. From a birds-eye view, SPI-DP is vanilla SPI: The shadow program DCG $\bar{P}$ is forward-evaluated to predict posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$; the task objective Φ is computed over $\boldsymbol{\theta}^{\bar{P}}$; gradients $\frac{\partial \Phi(\boldsymbol{\theta}^{\bar{P}})}{\partial \boldsymbol{x}^{\bar{P}}}$ are backpropagated and inputs $\boldsymbol{x}^{\bar{P}}$ are incrementally updated. However, if $\bar{P}$ contains a free-space motion skill, parts of $\boldsymbol{\theta}^{\bar{P}}$ will be a function of first-order planning with DGPMP2-ND, and the computation of gradients $\frac{\partial \Phi(\boldsymbol{\theta}^{\bar{P}})}{\partial \boldsymbol{x}^{\bar{P}}}$ requires differentiation through the gradient-based update of DGPMP2-ND (see Section 2.5.2.6). As DGPMP2-ND is differentiable by design and implemented in PyTorch, $\frac{\partial \Phi(\boldsymbol{\theta}^{\bar{P}})}{\partial \boldsymbol{x}^{\bar{P}}}$ can be computed end-to-end via automatic differentiation.

3.3.1.3 Hyperparameters and Best Practices

SPI-DP exposes several additional hyperparameters to configure the nested double-loop optimization algorithm. In particular, the outer- and inner-loop learning rates α and β must be balanced: Low values of α improve the stability of SPI, but require high values of β to prevent exploding runtimes; high values of α cause SPI to converge more quicky, but larger changes to programs parameters per SPI iteration in turn require more inner-loop replanning by DGPMP2-ND. Across all experiments, initializing the DGPMP2-ND update rate β with relatively high values $(0.25 \leq \beta \leq 1)$ and performing as few DGPMP2-ND iterations as possible yielded best results, as it allowed for more SPI iterations at the same compute budget. High learning rates can cause DGPMP2-ND to oscillate, as the obstacle factor may push trajectory points too far out of collision objects, causing the GP factor to increase suddenly, and vice versa. To prevent oscillations, β is decayed by a factor of 0.1 at every iteration after the critical error has reached zero (see Equation 2.19 and Listing 2.5). When the critical error is zero, the planned trajectory is feasible, as it is collision-free, reachable, and integrates seamlessly into the overall posterior trajectory. The remaining error terms h_{GP} and h_{traj} continue to be optimized at decayed learning rates to avoid reintroducing e.g. collisions or joint limit violations.

DGPMP2-ND converges considerably more quickly if the initial trajectory is close to the optimal trajectory. To further speed up convergence and avoid redundant computation, DGPMP2-ND outputs are cached for every planned motion skill $\bar{p}$ in the shadow program, and the posterior trajectory $\boldsymbol{\theta}^{\bar{p},i}$ at SPI iteration i is used to initialize the trajectory $\boldsymbol{\theta}^{\bar{p},i+1}$ at the next iteration. The underlying assumption is that changes, notably to the start state $\theta_0^{\bar{p},i+1}$, between SPI iterations are small. Empirically, the number of DGPMP2-ND iterations until convergence drops from up to 100 at the first SPI iteration to 2-5 for the last iterations, as the cached trajectory is already nearly optimal. As an additional benefit, DGPMP2-ND can continue trajectory optimization in the next SPI iteration if it failed to converge at the previous iteration, improving the overall likelihood of finding both feasible and optimal trajectories.

3.3.2 Experiments

SPI-DP is assessed in two real-world experiments on industrial and service robotics use cases. Both experiments focus on jointly solving trajectory and parameter optimization problems, with mutual interdependencies between parameterized and planned motion skills.

3.3.2.1 Collision-Free Household Pick-and-Place with Human Demonstration

In a first experiment, a household scenario is considered, in which a robot is tasked to pick up a cup from a table and place it into a cupboard, given one single human demonstration of the task (see Figure 3.17). In a variant of the same experiment, the robot is tasked to instead pick up and place a wine glass, given the same human demonstration. The experiment tests the following hypotheses:

1. SPI-DP plans collision-free, smooth pick-and-place motions for various target poses and object geometries.

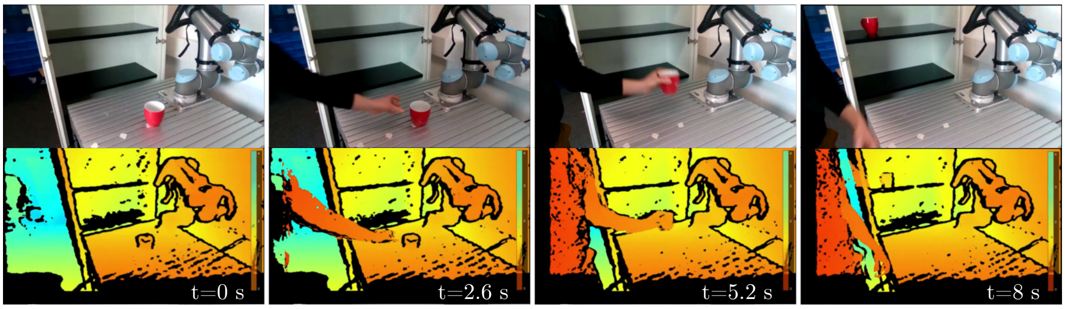


Figure 3.17: RGB (top) and depth images (bottom) of a human demonstration of a household pick-and-place task.

2. SPI-DP can jointly optimize robot programs to satisfy both motion-level constraints such as collision avoidance and similarity to a human demonstration, as well as task-level constraints such as the target pose.
3. SPI-DP can optimize robot programs based on a single human demonstration, even when the demonstrations have different pick-up and target poses than those required at runtime.

Experiment setup The experiment setup is shown in Figure 3.18 (right). A red cup is placed at a fixed position on a table, and is to be placed at one of four different target poses on two shelves of a cupboard. A UR5 collaborative robot,[9] ATI Gamma force-torque sensor[10] and Schunk pneumatic gripper[11] are used. 10 human demonstrations are collected using an Intel RealSense RGB-D camera (see Figure 3.17), covering different pick-up poses of the cup on the table as well as different target poses on two shelves of the cupboard. The 3D Cartesian trajectory of the cup was parsed by color segmentation; the resulting trajectories are shown in Figure 3.18 (red).

The source program is shown in Figure 3.19. The ARTM source program representation is used in this experiment. Two subprograms, `Pick Up` and `Place`, are linked by a `Move to State` transfer motion. Optimization of both transfer and placing motions are considered: A collision-free, smooth transfer motion is to be planned, which respects the additional constraints implicit in the human demonstration – notably, to keep the cup upright during transfer. In a second set of experiments, zero-shot transfer to a wine glass is considered, which additionally requires the optimization of the target pose.

`Move to State` is a planned motion skill, which defaults to a point-to-point motion in $\mathcal{C}$-space unless the optional `Path` parameter is specified. In this experiment, the `GoalState` parameter denoting the target configuration as well as the

---

[9] Universal Robots A/S, Odense, Denmark

[10] ATI Industrial Automation Inc., Apex, USA

[11] Schunk SE & Co. KG, Lauffen am Neckar, Germany

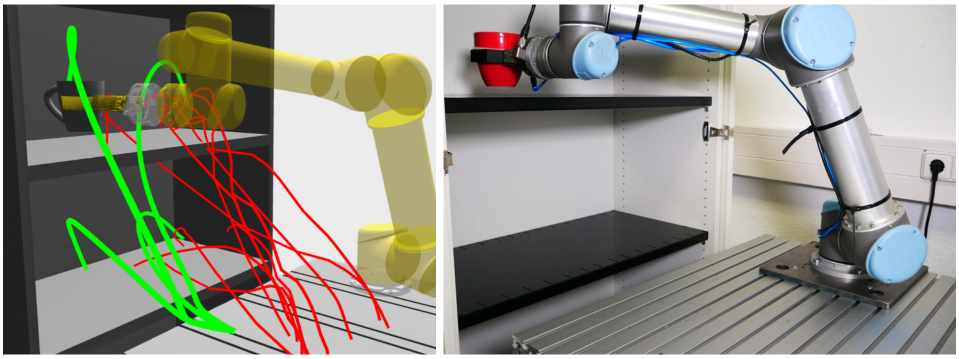


Figure 3.18: Left: 10 human demonstrations (red) and 4 exemplary trajectories planned by SPI-DP (green) for a household pick-and-place task. Right: Real-world execution of the optimized robot program.

motion trajectory `Path` are optimized. `Pick Up` contains a sequence of gripper and motion skills, which are not optimized in this experiment. The `Place` skill consists of a short linear approach motion followed by a `Move Linear Relative Contact` force-controlled contact skill (see Experiment 3.1.3.1 for a detailed description).

DGPMP2-ND is initialized with a mesh representation of the environment (see Figure 3.15 for a rendering). As the motions occupy a large fraction of the robot's workspace, the SDF is constructed with a voxel size of 10 mm, which proved sufficient for the application. DGPMP2-ND is configured with an update rate of $\beta = 0.4$ and patience $\varphi = 75$.

In a first series of experiments, the trajectory of the transfer motion is optimized. A shadow program for the pick-and-place program show in Figure 3.19 is constructed (see Figure 3.15 (gray)). The shadow skill for `Move Linear Relative Contact` is trained on 4000 real-world executions of randomly sampled executions of the `Place` subprogram. As no program parameters are optimized in this series of experiments, no SPI update is made, and trajectory optimization is performed in one forward pass through the shadow program. A total of 40 trials are performed, one for each combination of human demonstration and target pose.

In a second series of experiments, the cup is swapped for a wine glass. No new human demonstrations or training data are collected. In the collision environment, the mesh of the cup is swapped for a mesh of the wine glass, and the gripper geometry is changed to accommodate the new workpiece geometry. The collision environment and real-world setup are shown in Figure 3.20. The DGPMP2-ND hyperparameters remain unchanged. A total of 40 trials are performed, one for each combination of human demonstration and target pose.

Results In the first series of experiments, SPI-DP successfully optimized all 40 motion trajectories to be smooth and collision-free. The trajectories were initialized to point-to-point motions in configuration space, causing collisions in all 40 cases. Likewise, several of the human demonstrations contained collisions due to imprecise

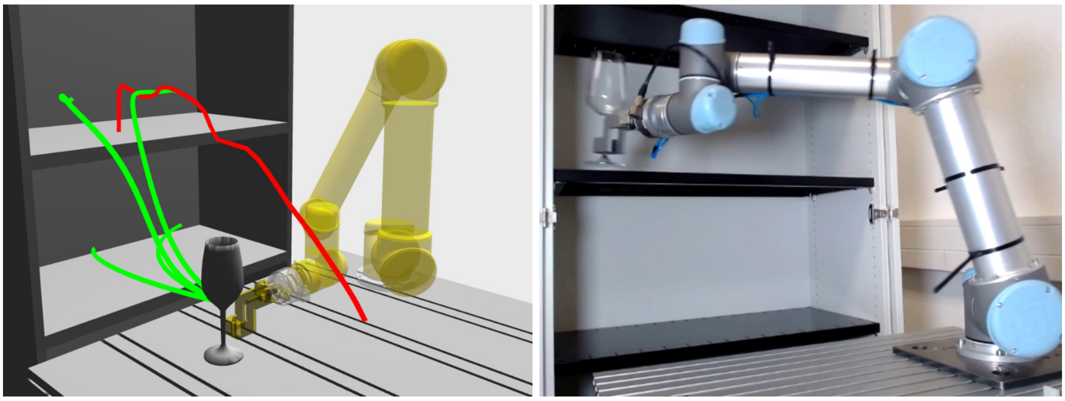


Figure 3.20: Left: 4 exemplary trajectories planned by SPI-DP (green) for placing a wine glass into a cupboard. Right: Real-world execution of the optimized robot program.

segmentation of the cup. After optimization, all trajectories were collision-free. Four example trajectories, one for each target pose, are shown in Figure 3.18. Due to the GP factor, the generated trajectories are very smooth and permit execution with high dynamics. The human demonstration acts as a regularizer on the planned motion, mainly ensuring that the cup is kept upright during manipulation. It was found that there is a direct trade-off between smoothness, collision avoidance and proximity to the human demonstration, as the Jacobians of the GP, obstacle and prior trajectory factors directly oppose each other – avoiding an obstacle will incur a GP penalty, and may push the trajectory farther from the human demonstration. Likewise, the planned motion may have to deviate far from the human demonstration to remain smooth and reach the goal, particularly if the queried target pose is on a different shelf than the demonstrated target. Figure 3.21 illustrates the influences of the respective factors on the optimized trajectory. The target pose was reached with a mean accuracy of 0.6 mm across all trials.

In the second series of experiments, the cup is swapped for a wine glass. Again, all trajectories were collision-free after optimization. Four exemplary trajectories to different target poses are shown in Figure 3.20 (left). As the wineglass is larger relative to the space between shelves, the planning problem is more challenging. Particularly for the top shelf, the planned trajectories closely approach the edge of the shelf, while avoiding collision. As the human demonstrations were performed with cups, they were less representative of the task, yet still regularized the orientation, keeping the wine glass upright for most of the trajectory.

Discussion The results give credence to the hypothesis that DGPMP2-ND permits the optimization of motion trajectories with respect to collision, smoothness and other motion-level constraints. DGPMP2-ND can be guided by a single human demonstration, even if the start state and goal pose passed as a skill parameter differ considerably from the demonstrated start and goal poses, and the demonstration

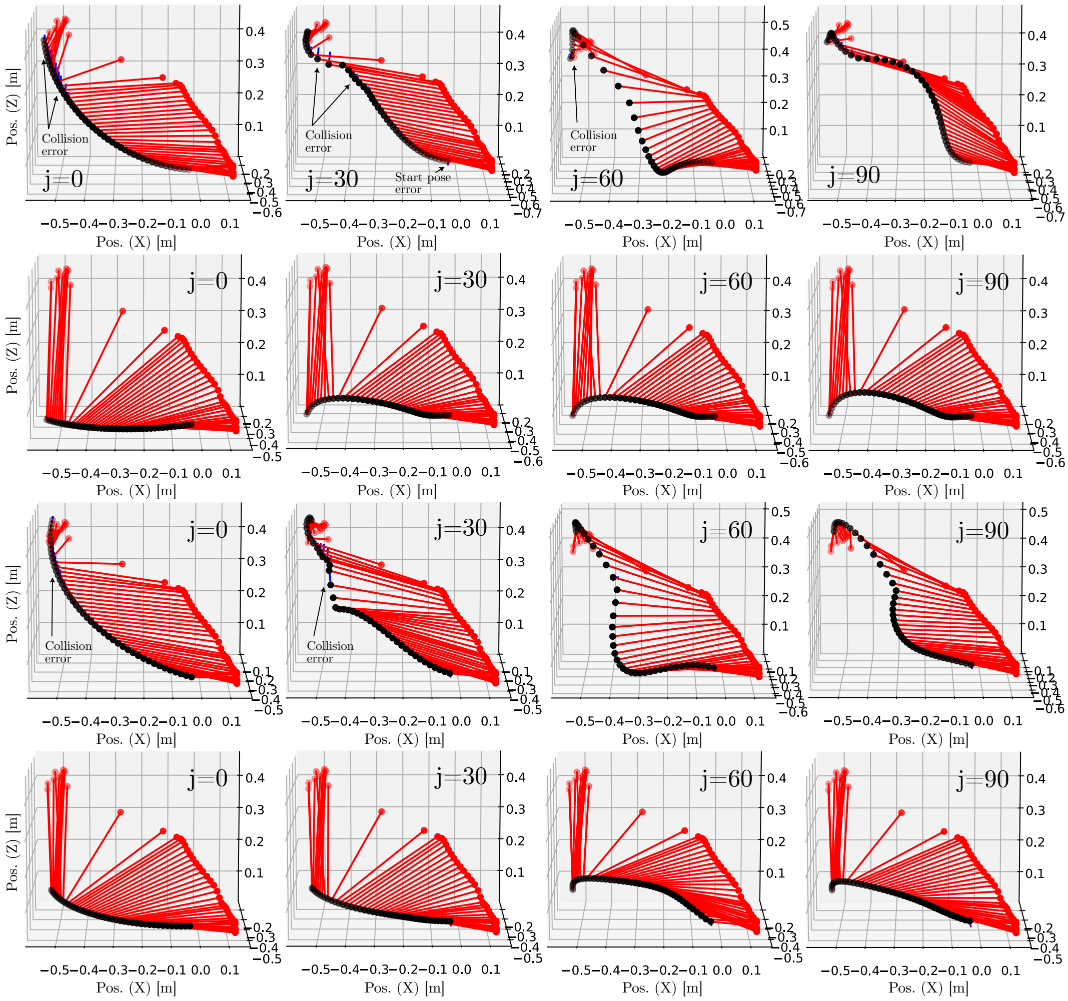


Figure 3.21: Collision (blue), prior trajectory (red) and start pose (purple) factors during DGPMP2-ND optimization for four different target poses, given the same human demonstration (red). The planned trajectory (black) is iteratively optimized to approximate a human demonstration, while guaranteeing collision freeness, adherence to start and goal poses, joint limits and smoothness. The optimized trajectory remains smooth despite a highly discontinuously sampled human demonstration.

	Hole found		Cycle time (s)	
	unoptimized	optimized	unoptimized	optimized
Hole 1	6/20	11/20	2.29	1.39
Hole 2	6/20	10/20	1.95	0.77
Hole 3	7/20	20/20	1.97	0.48

Table 3.4: Success rates and cycle times for force-controlled poka-yoke testing of screw holes before and after optimization.

is not collision-free. Balancing the GP, obstacle and prior trajectory factors, in particular, required considerable hyperparameter tuning. The appropriate factor weights σ_{obs}, σ_{traj} and σ_{GP} had to be balanced empirically. For this experiment, values of $\sigma_{\text{obs}} = 3e - 5$, $\sigma_{\text{traj}} = 0.4$ and $\sigma_{\text{GP}} = 0.3$ were found suitable. Too high values of σ_{obs} caused the planner to oscillate, causing large changes to colliding points on the trajectory and subsequent updates in the opposite direction due to the GP prior, often forcing the trajectory back into collision. Too large values of σ_{traj} could force the trajectory into collision, if the human demonstration collided with the environment. The appropriate factor weights depend on the update rate β, the required degree of smoothness, and the quality of the human demonstration. With the appropriate hyperparameters, DGPMP2-ND stably converges and is remarkably robust against very noisy and irregularly sampled demonstrations.

3.3.2.2 Force-Controlled Engine Block Poka-Yoke Quality Assurance

Experiment 3.3.2.1 demonstrated the ability of DGPMP2-ND to optimize motion trajectories with respect to motion-level objectives such as collision-freeness, smoothness or proximity to a human demonstration. A second experiment tests the hypothesis that SPI-DP is capable of *jointly* optimizing task- and motion level objectives. To that end, a quality assurance (QA) task is considered, in which a probing pin is inserted into several screw holes on an engine block. This is a common technique in poka-yoke (“mistake-proof”) manufacturing (Shimbun, 1988), which emphasizes preventing irrecoverable defects. By detecting e.g. insufficiently deburred or improperly drilled screw holes before assembly, mistakes can be corrected and discarding parts can be avoided. This experiment evaluates to what extent SPI-DP can optimize a robot program for task success and overall efficiency subject to stochastic process noise, while avoiding collisions with the workpiece.

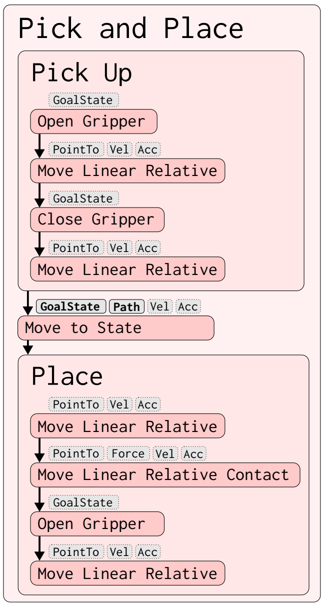


Figure 3.19: Source program for a household pick-and-place task.

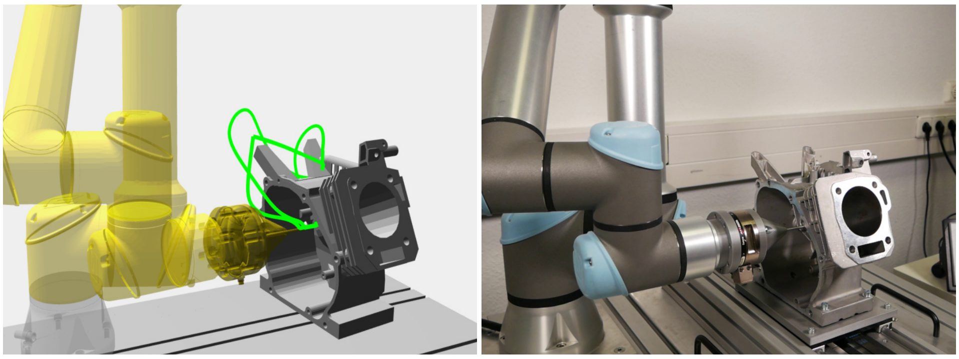


Figure 3.22: Left: Trajectory planned by SPI-DP (green) for a poka-yoke QA task. Right: Real-world execution of the optimized robot program.

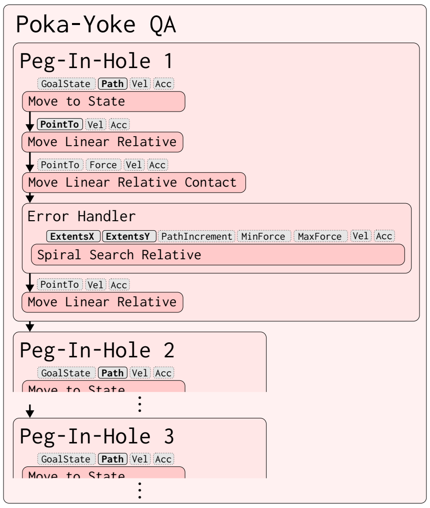


Figure 3.23: Source program for a poka-yoke QA task.

Experiment setup The experiment setup is shown in Figure 3.22 (right). An single-cylinder engine block is mounted on a linear axis, which is used to simulate stochastic process noise. A UR5 industrial manipulator[12] with an ATI Gamma force-torque sensor[13] are used. A metal probing tip is attached to the end effector via a 3D-printed adapter. The source program is shown in Figure 3.23. The ARTM program representation is used. The robot is tasked to approach three holes on the front, top and back sides of the engine block, respectively, via a `Move to State` planned motion skill. Due to the presence of process noise, the robot performs a force-sensitive contact motion (a `Move Linear Relative Contact` skill) to establish contact with the surface. If contact could be made, a spiral search motion (`Spiral Search Relative`) follows. The overall task objective is the minimization of a linear combination of task failure (Φ_{fail}, see Equation 3.6) and cycle time (Φ_{cycle}, see Equation 3.5). The `Path` parameters of the three `Move to State` skills as well as the `PointTo` parameter of `Move Linear Relative` and the `ExtentX` and `ExtentY` parameters of `Spiral Search Relative` are optimized.

The shadow program contains learned shadow skills for `Move Linear Relative Contact` and `Spiral Search Relative`. `Move Linear Relative Contact` is trained on 3200 real-world contact motions, while `Spiral Search Relative` is trained on 2600 real-world executions. At each run, process noise is simulated by uniformly sampling a spatial offset and translating the engine block accordingly via the linear axis. DGPMP2-ND is configured with an update rate of $\beta = 1.0$ and

---

[12] Universal Robots A/S, Odense, Denmark

[13] ATI Industrial Automation Inc., Apex, USA

patience $\varphi = 30$. SPI is configured with an update rate $\alpha = 0.001$. Outer-loop SPI is run for 15 iterations. A total of 20 evaluation trials are performed. For each trial, a random offset is sampled within a 4mm interval, applied via the linear axis and both the optimized and unoptimized robot programs are executed.

Results The resulting motion trajectory is shown in Figure 3.22 (left). The quantitative results are shown in Table 3.4. Across 20 trials, the optimized parameterization improves the likelihood of task success by 83 %, 67 % and 186 % for each of the three holes, respectively. The large difference in the number of successful insertions for each hole is due to different workpiece geometries around the hole: Holes 1 and 2 are surrounded by concave surfaces, which can cause the search motion to spiral away from the hole, and their hole diameter is smaller. Cycle times are reduced by 39 %, 60 % and 75 % respectively. These reductions are due to a combination of improved spiral search parameters, whereby a higher probability of finding the hole implies more spiral searches terminating early, reducing the expected cycle time per spiral, and optimized, shorter transfer motion trajectories. All optimized transfer motions are collision-free.

Discussion The results indicate that SPI-DP permits the optimization of robot program parameters for complex robot programs consisting of multiple, hierarchically composed subprograms, as well as the joint optimization of motion trajectories and robot skill parameters. Task-level metrics such as cycle time and task success are optimized along with motion-level objectives such as collision avoidance and smoothness. As in Experiment 3.3.2.1, DGPMP2-ND plans collision-free motions given a static 3D representation of the environment. With stochastic process noise, however, collision-freeness or motion-level optimality of planned trajectories cannot be guaranteed, as the environment at runtime may deviate from the environment representation used for planning. In Experiment 3.3.2.2, the magnitude of stochastic variations is sufficiently small to cause collisions. By an extension similar to that proposed by Kienle et al. (2024) for SPI, constructing the collision SDF of DGPMP2-ND from sensory data about the current environment may ensure that SPI-DP optimizes robot programs with respect to the current environment at hand.

3.3.3 Related Work

SPI-DP is a nested gradient-based optimization algorithm that integrates an iterative differentiable motion planner into a model-based first-order parameter optimizer. As such, SPI-DP is situated at the intersection of differentiable programming and model-based optimization.

3.3.3.1 First-Order Model-Based Optimization

From a bird's-eye view, SPI-DP performs gradient-based optimization over a partially learned model of robot dynamics, while incorporating a differentiably implemented, explicit model of robot kinematics and trajectory semantics. In related fields, several approaches have been proposed that leverage differentiable programming to create a forward model, which forms the basis for first-order model-based optimization. Jin et al. (2020) propose Pontryagin Differentiable Programming (PDP), a framework for differentiable optimal control. PDP introduces a differentiable optimal control loop, which permits the differentiation of a system's trajectory with respect to e.g. controller parameters. Beyond PDP, first-order, model-based approaches based on differentiable programming have received increased attention in the field of optimal control (Amos et al., 2018). Jin et al. (2020) evaluate PDP on a 2-DoF robot arm as well as a 6-DoF quadrotor. In their analysis, they highlight the central advantage of model-based first-order optimization: Their incorporation of prior knowledge (here, of optimal control theory) provides the optimizer with an inductive bias, enabling the achievement of "higher efficiency and capability than existing learning adn control methods" (Jin et al., 2020). As SPI-DP operates on a fundamentally similar principle, the same advantage applies: The GP, obstacle and other factors are inductive biases for the optimizer, and avoid the need for learning collision-free planning from data.

In model-based RL, the benefits of inductive priors for learning and optimization have led to increasing adoption of differentiable, explicit models in conjunction with learned DNNs. Okada et al. (2017) integrate a differentiable formulation of path integral optimal control into a neural network architecture, permitting the end-to-end learning of network and control parameters while providing the learner with an inductive bias for optimal control. Srinivas et al. (2018) propose Universal Planning Networks (UPNs), which integrate differentiable planning into a neural network architecture. UPNs contain a "gradient descent planner", which iteratively optimizes a plan to reach a given goal using gradient descent. The outer-loop learner then performs gradient descent over the "unrolled" computation graph of the inner planner. This double-loop policy learner resembles SPI-DP in the context of RL; unlike UPNs, however, SPI-DP plans in $\mathcal{C}$-space rather than latent space, permitting the intuitive and transparent specification of motion constraints. Similarly, physics-informed machine learning proposes to integrate differentiable, mathematical models of the laws of physics in ML models. Lutter et al. (2021) show that optimizing the parameters of differentiable models of rigid-body dynamics using offline RL requires less training data and produces more stable results than learning black-box neural policies. Hu et al. (2019b) propose a differentiable simulator for soft robots and demonstrate that gradient-based optimizers can compute viable actuation controllers in considerably fewer iterations than model-free RL.

3.3.3.2 Differentiable Motion Planning and Parameter Optimization

Differentiable motion planning permits the use of planning algorithms with known properties, e.g. convergence or performance guarantees, in conjunction with gradient-based optimization or learning methods. Ni and Qureshi (2024) leverage physics-informed neural networks, combining two ResNet-style DNNs with a differentiable implementation of the Eikonal equation for constrained motion planning (Ni and Qureshi, 2022). Yonetani et al. (2021) present a differentiable implementation of A* search for path planning. They integrate the differentiable planner with a neural network to realize an end-to-end learnable planning system, which can be trained to solve planning problems more efficiently than vanilla A*. Pogančić et al. (2020) propose a more general variant of this approach, by automatically constructing a differentiable analogue of a black-box combinatorial solver, which supports backpropagation of errors and computation of gradients, to facilitate integration into hybrid, neurosymbolic architectures. Hybrid planners combining neural networks and differentiable search- or optimization-based planners promise considerably faster planning times, as the neural components may steer the planner into regions of the planning space which are likely to contain a solution, based on data seen during training. Bhardwaj et al. (2020) propose a trainable module for DGPMP2 consisting of a neural network, which outputs the covariances for the GP, obstacle and other factors. This module is trained to produce a suitable trade-off between the individual factors for a given environment, start and goal configurations. SPI-DP exploits the differentiability of DGPMP2 in a different way: Instead of learning planning parameters, DGPMP2 is integrated into a DCG representing a larger, end-to-end differentiable robot program, for first-order program parameter and trajectory optimization. This usage does not, however, preclude the simultaneous optimization of planner parameters, which is a promising direction of future research.

SPI-DP leverages a differentiable model of robot kinematics, a differentiable representation of trajectories as a factor graph as well as a differentiable motion planner over this factor graph, integrated into a first-order optimizer for robot program parameters. While the degree of integration achieved by SPI-DP is unique, one state-of-the-art approach is particularly related to SPI-DP, both in scope and algorithmic detail. Toussaint et al. (2018) combine physics-informed, differentiable models with a principled, mathematical robot program representation to solve TAMP problems. They incorporate differentiable primitives, which impose motion-level constraints such as smoothness or impulse exchange, into the Logic Geometric Program (LGP) representation of robot programs (Toussaint, 2015). The TAMP problem is then solved by Multi-Bound Tree Search (MTBS), a multi-level search-based optimizer for LGPs that exploits gradient information (Toussaint and Lopes, 2017). The resulting LGP is optimal with respect to the task objectives, while respecting motion-level constraints. From a high-level perspective, LGP solving with differentiable primitives addresses the same problem as SPI-DP: Joint task- and motion-level optimization of robot programs. One significant difference is that

SPI-DP performs parameter optimization over a differentiable surrogate, and can therefore optimize program parameters and motion trajectories for near-arbitrary source programs.

3.3.4 Discussion

SPI-DP integrates the DGPMP2-ND differentiable motion planner with the SPI first-order parameter optimizer to jointly optimize program parameter and motion trajectories. As SPI-DP operates on the shadow program representation, it is applicable to arbitrary skill-based source program representations with planned motion skills (see Section 3.3.1.1). For robot programs with such skills, integrating differentiable motion planning into SPI is necessary to ensure end-to-end differentiability of the shadow program. One core advantage of SPI-DP specifically is that program parameters and motion trajectories are optimized *jointly* with respect to task objectives Φ and the motion-level constraints represented in the DGPMP2-ND factors. For tasks such as the QA task considered in Experiment 3.3.2.2, the impact of individual skill parameters on task-level objectives such as cycle time are highly correlated across the complete program. For search-based insertion under uncertainty, for example, cycle time depends largely on the expected completion time for search motions, which in turn depends on the target pose of the preceding approach motion; if that approach motion is a planned motion skill, the planned motion, in turn, depends on the target pose parameter. As the target pose parameter changes over the course of optimization, large adjustments to the approach motion trajectory may be required, as parts of the motion may become infeasible due to joint limit constraints or collisions with the environment. In short, in the presence of planned motion skills, coupling parameter optimization and motion planning ensures that task objectives are achieved while respecting motion constraints. For practical applications, SPI-DP has the additional advantage of enabling a degree of reactivity to dynamic environments, enabling replanning without requiring training or finetuning, provided that the SDF used by DGPMP2-ND represents the current environment.

Like most gradient-based methods, SPI-DP converges on local minima (Jin et al., 2020), both with respect to program parameters as well as motion trajectories. For certain challenging environments, such as narrow passages, SPI-DP may fail to converge on a set of motions and parameters that solve e.g. a reaching task. This limitation may be mitigated by "warm-starting" DGPMP2-ND with initial trajectories that already approximately solve the task (Lembono et al., 2020), which can be pre-planned by a non-differentiable, sampling-based planner. Likewise, DGPMP2-ND can be restarted with different initial trajectories to increase the probability of convergence in complex environments.

The entanglement of parameter and trajectory optimization gives rise to an additional, less evident limitation. In the current implementation, situations can occur in which the task objectives Φ may clash with motion constraints during optimization. It is possible, for example, that the outer optimizer (SPI) changes high-level program parameters so that the trajectory starts or ends in collision; this occurs if the skill before a planned motion ends in collision, or if SPI optimizes the goal pose parameter of a planned motion into collision. This is due to the fact that Φ does not reflect motion-level objectives such as collision-freeness, and SPI may move e.g. goal pose parameters into collision objects, which would make collision-free planning impossible. To address this limitation, a global collision-freeness objective may be added to Φ, in a manner similar to other, task-level objectives such as minimization of cycle time (see Equation 3.5). This would not require architectural or algorithmic changes, as Φ is evaluated for the complete posterior trajectory $\boldsymbol{\theta}^{\bar{P}}$, the collision SDF of the environment is already available, and the differentiable obstacle loss of DGPMP2-ND can be applied at the program level as part of Φ. The same reasoning applies for other motion-level constraints such as adherence to joint limits, which can be integrated into Φ in the same manner. Evaluation of SPI-DP with such global motion constraints is a promising avenue of future research.

Nested gradient-based optimizers such as SPI-DP “unroll” the inner optimization loop into an acyclic DCG and perform automatic differentiation on this unrolled graph. Jin et al. (2020) motivate PDP in part by showing that such “unrolling” is memory intensive and incurs performance penalties, as the unrolled DCG grows with the number of inner-loop SGD iterations. In the context of SPI-DP, it has been experimentally observed that the memory footprint grows with the number of inner-loop DGPMP2-ND iterations. Linear approximation of the inner-loop gradient as in FOMAML (Finn and Levine, 2017) or Reptile (Nichol et al., 2018) can considerably reduce the memory requirements of SPI-DP and will be considered in future work.

3.4 Discussion

SPI and its variants promises to optimize robot programs both with respect to program parameters as well as motion trajectories. The central innovation of SPI is that it performs first-order optimization over a learned, differentiable surrogate of a robot program. Most of SPI’s properties stem from the dual nature of the NRP program representation, which permits SPI to combine the virtues of black-box optimizers with first-order optimization approaches. As NRPs allow source programs to be represented in a wide range of skill-based program representations, SPI can optimize the parameters of near-arbitrary robot programs, without imposing representational requirements such as differentiability; at the same time, by optimizing over the shadow program DCG, SPI can leverage gradient information to steer the search along the direction of steepest descent. The dual nature of SPI as a black-box, first-order optimizer enables the direct application of SPI in real-world industrial settings, where non-differentiable source program representations predominate. Most saliently for industrial use, SPI is fundamentally compatible with industrial safety certification: As the source program representation is used for execution, which may have been audited and certified for safety, the program parameters optimized by SPI may be written back to the certified program. Often, re-certification will not be required, provided the optimal parameters lie in a given, pre-approved range.

The demonstrated use cases ranging from household pick-and-place to industrial assembly tasks highlight the fact that SPI does not have any algorithmic restrictions on a given task or family of tasks. SPI can be used to optimize, in principle, robot program parameters for arbitrary tasks, provided they are realized by a parameterized, skill-based source program, and that task objectives can be expressed as a differentiable function of the expected posterior trajectory. The experiments in this chapter illustrate a variety of task objectives, including minimization of cycle time, respect of force limits or similarity to a human demonstration, as well as combination of multiple, partly contradictory objectives. They illustrate that SPI is a general-purpose optimizer for robot programs that supports a wide variety of use cases ranging from the optimization of process indicators to human imitation and collision-free planning.

One considerable advantage of SPI as a model-based optimizer is that it solves optimization problems without requiring optimal labels; instead, models are trained on data of robot behavior “as-is”, and optimization is conducted over this learned forward model of the process. While this avoids human labels or RL-like active exploration, both of which are challenging to realize in real-world industrial or service scenarios, SPI does require training data that both accurately reflects robot behavior and covers sufficiently large regions of the parameter and state spaces. Both of these requirements imply real-world challenges. For many applications, collecting large amounts of observations is impossible or prohibitively expensive. Consider destructive, irreversible processes such as sanding or drilling, or robot actions that only rarely occur, such as recovery routines for infrequent errors. For such applications, only limited datasets are available. Large-scale pretraining in simulation considerably reduces the real-world data requirements of SPI (see Experiments 3.1.3 and 3.2.3). Follow-up work by Kienle et al. (2024) explores the integration of an additional vision modality, to make SPI conditional on images of the current environment. The pretraining of multimodal foundation models on diverse tasks is an avenue of ongoing and future work that promises to further reduce the real-world data requirements of SPI. The requirement for diverse training data is mitigated, if not altogether eliminated, by sequential transfer learning (see Experiment 3.2.3), albeit only at the finetuning stage. Future work is concerned with investigating the degree to which shadow programs generalize to inputs outside of their training data distribution.

From the perspective of human-machine interaction, SPI holds tremendous promise, but also poses a set of challenges. SPI permits the optimization of robot program parameters and motion trajectories with respect to near-arbitrary task objectives purely by optimization over a learned model. In practice, SPI avoids the time-consuming and expensive phase of iterative parameter tuning by human robot programmers and domain experts, particularly for tasks which are, for humans, challenging to observe and understand. Small-scale, fine-grained assembly tasks with tight tolerances, physical contact dynamics or stochastic noise such as the use cases considered in Sections 3.1.3 and 3.2.3 are salient examples; another class of optimization problems challenging to human programmers are multicriterial optimization problems with mutually opposing task objectives such as the examples considered in Section 3.3.2. For such applications, “programming by optimization” (Hoos, 2012) via data-driven optimizers such as SPI presents a better, faster and more economical alternative to human trial and error. However, while the integration of SPI into productive robot programming workflows removes the burden of iterative trial-and-error from program parameterization, it simultaneously introduces the challenge for robot programmers to use and configure SPI. The skills required to configure data collection, train ML models and choose suitable hyperparameters for an iterative optimizer are outside the scope of a typical industry practitioner’s experience. To help bridge this “skill gap” (Li et al., 2021), a GUI has been developed to guide the robot programmer through the program optimization process (see Alt et al. (2024d) and Chapter 5). It comprises intuitive user interfaces for the selection of the subprogram to be optimized, for the specification and analysis of training datasets, the training of a shadow program and the configuration and analysis of SPI optimization. The GUI employs XAI and data visualization techniques to make model performance, data quality and optimizer behavior as intuitive as possible for the user and offers two different levels of assistance (“guided” and “expert” modes), allowing users to select the mode corresponding to their level of AI expertise. A preliminary, small-scale user survey indicates that the proposed GUI enables AI novices to optimize robot program parameters for a real-world gear assembly task, with participants highlighting the usefulness and intuitive usability of the system (Alt et al., 2024d). A larger-scale user study of SPI in the context of industrial robot programming is required to provide more robust insight into the benefits and challenges of data-driven program parameter optimization in practical applications.

3.5 Conclusion

3.5.1 Summary

Section 3.1 introduced SPI, a first-order optimizer for parameterized robot programs. It leverages NRPs, differentiable forward models of robot control programs, to perform model-based, iterative optimization over a learned surrogate of the robot program. SPI supports various differentiable task objectives, including cycle time, task success, path length, and force/torque constraints, allowing for multicriterial optimization by combining multiple objectives. Experiments demonstrate the effectiveness of SPI in optimizing force-controlled search (Experiment 3.1.3.2) and contact motions (Experiment 3.1.3.1) for industrial robotics tasks. Results show that SPI successfully optimizes motion parameters to optimize target contact forces while minimizing cycle time, and can generalize across different surfaces and program representations. SPI presents a data-driven approach to robot program parameterization, promising to free up human programmers for higher-level tasks.

Section 3.2 introduced a lifelong learning approach that continuously updates the shadow program to reflect current robot-environment dynamics. The chapter addresses the challenge of efficiently training NRPs in stochastic environments and keeping them up-to-date with nonstationary noise processes. A sequential transfer learning scheme is proposed, where the shadow program is pretrained on a large dataset representing various environment distributions and then fine-tuned at runtime on a smaller dataset of the current environment. This approach enables data-efficient optimization of robot program parameters and adaptation to changing environment conditions. Experiments 3.2.3.1 and 3.2.3.3 demonstrate the ability of of SPI with sequential transfer learning to optimize robot program parameters with respect to stationary and nonstationary stochastic environments. Sequential transfer learning is found to outperform meta learning for data-efficient training of shadow models in stochastic environments.

Section 3.3 introduced SPI-DP, an extension of SPI for jointly optimizing robot program parameters and motion trajectories. SPI-DP integrates the DGPMP2-ND differentiable motion planner into the shadow program framework, allowing for optimization of planned motion skills while respecting constraints like collision avoidance and smoothness. SPI-DP performs double-loop gradient descent over the shadow program to optimize both task-level objectives and motion-level constraints. In Experiment 3.3.2.1, SPI-DP successfully plans collision-free, smooth trajectories for placing cups and wine glasses based on a single human demonstration, even when target poses differ from the demonstration. Experiment 3.3.2.2 shows the ability of SPI-DP to jointly optimize task success and efficiency in a poka-yoke QA task for engine block inspection. These experiments validate the effectiveness of SPI-DP in solving complex robotic manipulation problems that require both parameter and trajectory optimization.

3.5.2 Outlook

Since its first publication (Alt et al., 2021), SPI has been under active development and has been the subject of several scientific publications (Alt et al., 2022b; Alt et al., 2025). Kienle et al. (2024) introduce the Multimodal Trajectory Transformer (MuTT) neural network architecture as a drop-in replacement for the GRU-based neural networks that form the learnable components of shadow programs (see Section 2.4.1.4). MuTT permits the conditioning of shadow programs on an image of the current environment, and consequently the gradient-based optimization of robot program parameters for a given environment. In industrial grasping and assembly use cases, MuTT-based shadow programs permit SPI to generate optimized program parameters for the particular environment at hand, enabling the use of SPI in an on-line way without continuous retraining. Ongoing research investigates the use of MuTT as a "foundation model", enabling SPI for unseen tasks or environments under few-shot or zero-shot regimes.

Another line of ongoing research investigates the application of SPI in the context of real-world industrial robot programming. Section 5.1.1 of Chapter 5 presents a general-purpose workflow for AI-based robot programming, which maps SPI and its variants as well as the program synthesis approaches proposed in Chapter 4 to the lifecycle of a an industrial robot program. It positions SPI as a practical, data-driven alternative or supplement to parameter tweaking by human experts during the ramp-up phase of industrial robot workcells, and shows that SPI with lifelong learning can address several use cases in robot-based production processes, such as re-optimization after maintenance interventions or wear and tear compensation.

Human programmers are, and likely will remain for the foreseeable future, a crucial factor in industrial automation. To deploy AI-enabled robot programming at scale, intuitive human-machine interfaces must be developed to enable AI novices to use and interact with AI systems such as SPI. Section 5.1.2 introduces an Explanation User Interface (XUI) for SPI, which guides users through the model training and program optimization workflow while providing explanation and visualization features depending on the user's level of AI expertise. Future work will focus on larger-scale, representative user studies to validate, study and improve user interaction.

Beyond scientific study, SPI is subject to ongoing integration into a larger-scale commercial robot programming ecosystem. SPI has been patented (Alt et al., 2022a) and an early version of the SPI source code is available under an open-source license.[14] SPI is being integrated into the ArtiMinds RPS and Learning and Analytics for Robots (LAR) product families.[15] Integration into the RPS robot IDE permits the automatic optimization of robot programs in the ARTM program representation with SPI, offering robot programmers the option of optimizing program parameters through intuitive user interfaces. Integration into the LAR platform for industrial robot data enables automatic collection of robot data from a range of robot manufacturers as well as both seamless training of shadow models and model-based program parameter optimization on cloud servers or edge devices. In its commercial implementation, SPI contributes to the larger-scale push toward data-driven, flexible manufacturing as well as human-centric and resilient production in the contexts of Industry 4.0 and Industry 5.0 (Xu et al., 2021).

---

[14] https://github.com/benjaminalt/shadow-program-inversion

[15] ArtiMinds Robotics GmbH, Karlsruhe, Germany

CHAPTER 4

Interactive AI-Enabled Robot Program Synthesis

Programming is an act of bidirectional communication between the programmer and the technical system: Through the program, the programmer conveys their intent to the system, and the system's behavior can, in turn, be understood by the programmer by reading the program (see Chapter 1). This chapter addresses the challenges of program creation, the determination of a suitable sequence of robot skills and initial parameterizations to perform a given task.

When programming robots, programmers leverage a range of domain- and task-specific expertise. Robot programming requires a thorough understanding of robot capabilities, including available skills and physical abilities based on kinematics. Moreover, it requires a deep understanding of the problem domain. Programming an industrial robot to sand a workpiece, for example, requires understanding of what sanding means, implying both knowledge about what constitutes a well-sanded surface for the particular application – what is the *goal* to be achieved – and what *sequence of actions* is likely to lead to that goal. This includes knowledge of possible failure modes, and strategies to recover from them: What happens the sander is pressed onto the surface with too much force, or if it is moved across the surface at a velocity that is too fast or too slow? What happens if there is a bump in the surface, which requires special treatment? Answering these questions requires a profound, latent understanding of domain-specific and common-sense physics. In the context of sanding, it requires highly task-specific understanding of the physics of abrasion and how sanding pads of a given material interact with a given workpiece materials; but also a common-sense understanding of what a smooth surface looks and feels like.

Furthermore, programming requires mapping intended effects to the robot actions that can produce them, subject to kinematic constraints. Some actions a human can perform may not be feasible, or a sequence of several robot actions may be required to achieve the same goal. One example is tool use: Where the dexterity of the human hand may permit a human to pick up and seamlessly use a tool in a variety of ways, a robot may have to put down and regrasp the tool to perform a task due to kinematic constraints imposed by a gripper with few DoF.

Programming is an iterative activity: Programs may be read and modified many times after their initial creation. During the ramp-up phase of an industrial robot-based assembly line, for example, programs for individual robot workcells are iteratively improved until the e.g. quality or cycle time requirements of the production process are met. Likewise, programs are read by human auditors for safety certification, and adjusted again after deployment when production lines are reconfigured for new workpiece variants or hardware is exchanged for scheduled maintenance. The challenge of creating robot programs is then amplified by the additional requirement of creating programs in a way which facilitates later reading and editing, likely by people other than the initial programmer.

Programming requires overcoming the representational divide between the ways in which humans conceptualize, reason about and express physical processes, and the syntax and semantics of robot programs. This is challenging even for experienced programmers. It is aided somewhat by structured, explicit and graphical program representations; it would be aided further if human programmers could specify and interact with programs in a modality in which they intuitively reason and communicate, such as natural language. Algorithms and frameworks for robot program synthesis that enable humans to intuitively create and modify robot programs are an important step toward democratizing robotics, permitting users with little expertise to instruct robots what to do and correct them when they make mistakes. Use of algorithms and representations that permit humans to understand the resulting robot behavior, moreover, may foster trust, paving a way for increasing the acceptance of AI-based tools by human programmers.

Research questions While SPI addresses parameter inference through optimization over a surrogate model, this chapter focuses on the complementary challenge of program synthesis: creating the initial *program skeleton*, a preparameterized sequence of robot actions, to solve a given task. It addresses the overarching research questions:

1. How can humans express their intent in a way that permits precision and detail where required, while avoiding redundant or unneeded specification?
2. How can executable robot programs be synthesized from underspecified task descriptions provided by human programmers?
3. How can human expert knowledge be represented, stored and leveraged for the purpose of automatic or interactive program synthesis?
4. To what extent can explicit robot program representations support the synthesis of robot programs that can solve complex tasks while being interpretable by human programmers?

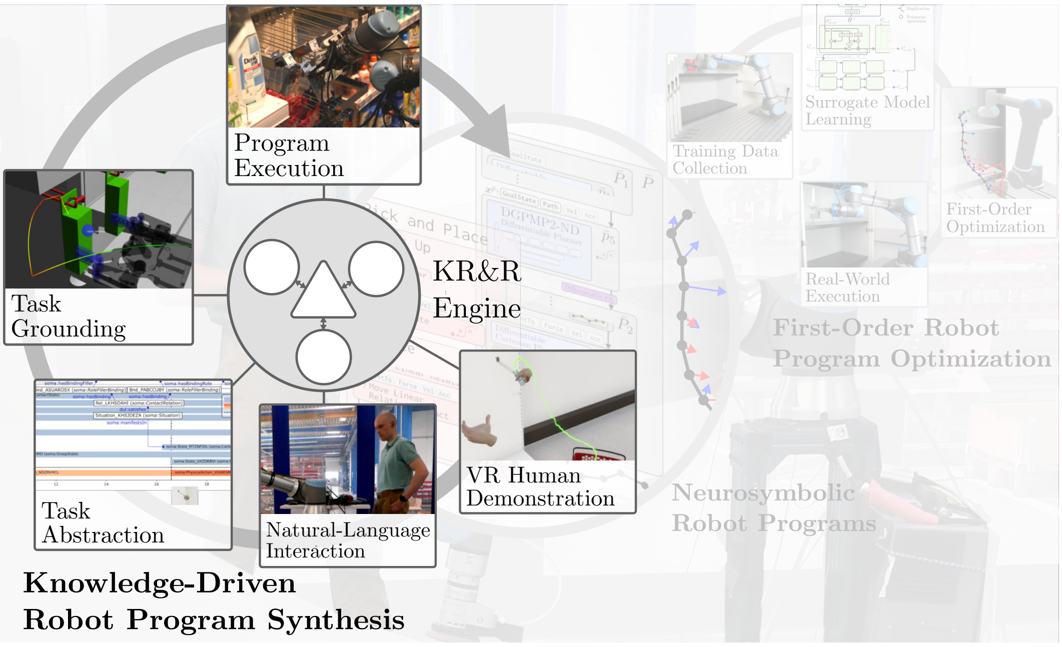


Figure 4.1: The MetaWizard family of robot program synthesis systems leverage explicitly represented, structured knowledge representations to create executable robot programs from high-level user inputs such as VR human demonstrations or natural-language interaction.

Overview To address these questions, this chapter introduces and studies the *MetaWizard* family of systems for interactive, knowledge-driven robot program synthesis. The central innovation of the MetaWizard systems is the notion of modular composition of symbolic and subsymbolic algorithms and knowledge sources. They permit the user to express their intents in high-level, intuitive representations such as VR demonstrations or natural language. They then leverage *semisymbolic reasoning*, such as hybrid reasoning over knowledge bases or Retrieval-Augmented Generation (RAG), to translate this description of user intent into an *underspecified plan* of robot actions, and to ground this plan in the real-world execution environment. The grounded plan is then translated into a robot program, which is executed on the robot.

This chapter introduces systems for both end-to-end and interactive program synthesis. In the context of neurosymbolic programming with NRPs, the MetaWizard family of program synthesizers assist robot programmers in the initial creation of NRPs given a high-level description of the task at hand. They synthesize NRPs by exploiting the symbolic program structure and known semantics of NRP source programs. To that end, they leverage existing task knowledge, encoded in symbolic representations of knowledge or data, to realize program synthesis for complex tasks without requiring data-driven learning or exploration. All three MetaWizard variants have a modular architecture, permitting the flexible integration of different interaction modalities, knowledge sources and reasoners. Their ability to generalize to novel tasks and domains gives the MetaWizards their name: From a user's perspective, a MetaWizard is a programming assistant ("wizard") for the synthesis of robot programs. From a technical perspective, the provision of structured knowledge *gives rise* to programming assistants for new tasks and domains, without requiring changes to the program synthesis system's algorithms or code. In this sense, MetaWizards are *metaprogramming systems* which, parameterized with structured knowledge, synthesize robot programs for the tasks and environments represented by this knowledge.

Outline Section 4.1 introduces the MetaWizard program synthesis system as well as the algorithms and data structures used to represent and reason about tasks, actions and plans. It demonstrates the ability of MetaWizard to synthesize NRP source programs from human VR demonstrations in the context of retail assistance. Section 4.2 proposes an MetaWizard2, a variant of MetaWizard that realizes an interactive robot programming paradigm, as well as algorithms and data structures for grounding underspecified action plans through dialogue with the user. The approach is validated on a surface treatment task in the context of robot-based remanufacturing. Section 4.3 introduces MetaWizardLLM, a MetaWizard variant that leverages pretrained LLMs for reasoning about high-level task sequences, and grounds generated plans in a structured, digital representation of the environment. MetaWizardLLM is validated on an industrial gear assembly task. Finally, Sections 4.4 and 4.5 contextualize and discuss MetaWizard, MetaWizard2 and MetaWizardLLM in the context of AI-enabled robot programming in general, and neurosymbolic programming in particular.

4.1 Knowledge-Driven Robot Program Synthesis

This section introduces the MetaWizard program synthesis system, presenting an extended account of work first published in (Alt et al., 2023).

4.1.1 Overview

MetaWizard is a program synthesis system capable of bootstrapping robot program skeletons to perform concrete manipulation problems given sparse task descriptions. It combines a structured, semisymbolic representation of task, domain and commonsense knowledge, a framework for human VR demonstrations of tasks, as well as a subsystem for grounding generated plans in simulations and real-world percepts (see Figure 4.2).

MetaWizard decomposes program synthesis into the following steps:

1. **User input:** The user describes the task at hand via an intuitive modality, such as a demonstration in VR or natural language. The task description is

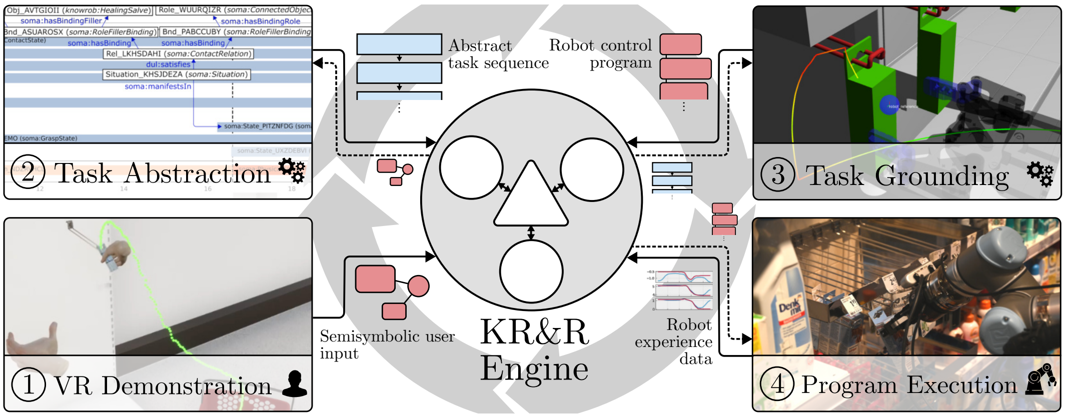


Figure 4.2: High-level overview of the MetaWizard system for knowledge-driven program synthesis.

not required to be complete; missing task-relevant details are resolved during task grounding (see below). The task description is parsed into a structured, hybrid symbolic-subsymbolic ("semisymbolic") representation of intended actions and effects and stored in a knowledge base. The process of parsing user input is described in detail in Section 4.1.3.

2. **Task abstraction:** User input is typically described in terms of human actions or capabilities, often referring to objects in a different environment (such as a VR environment) than the environment at hand. By reasoning over the knowledge base, the parsed task description is *lifted* to the abstract level. The resulting abstract task sequence is heavily underspecified and does not refer to, e.g., concrete robot actions to complete it. Section 4.1.4 provides a detailed overview.

3. **Task grounding:** The abstract task sequence is *grounded* in prior knowledge, simulations and real-world sensory data to form a fully specified NRP source program, a sequence of parameterized robot skills that solve the task with the given robot in the environment at hand. Task grounding is described in detail in Section 4.1.5.

4. **Program execution:** The source program is executed on the given robot hardware, and robot experience data can be fed back into the knowledge base. Section 4.1.6 details the execution pipeline.

Figure 4.2 illustrates the MetaWizard framework from an algorithmic perspective. It highlights the central role of KR&R in the proposed framework, which combines a semisymbolic knowledge base with reasoners operating on it to realize functionality such as deduction of facts from existing knowledge, induction of knowledge from experience data, or symbolic constraint-based planning of task sequences.

4.1.2 Knowledge Representation

MetaWizard uses the KnowRob KR&R engine (Beetz et al., 2018) as a knowledge repository and reasoning platform. KnowRob permits downstream systems to reason over a semisymbolic knowledge representation, combining subsymbolic modalities such as vision or robot trajectory data with symbolic representations of knowledge such as ontologies.

4.1.2.1 A Semisymbolic Representation of Knowledge

Triple store ▷

The central data structure in KnowRob is a knowledge graph that combines ontological with subsymbolic knowledge. It is realized via an Resource Description Framework (RDF) triple store, which stores data as subject-predicate-object triples and is integrated with subsymbolic sources of knowledge such as a database of spatial transforms.

RDF triple store At the data level, symbolic knowledge is represented as entries in the triple store. Consider the retail assistance scenario introduced in Chapter 1. The environment, such as the products on the shelves, their types and other properties, information about the robot as well as any other task-relevant or commonsense information is represented as RDF triples:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix dul: <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>
@prefix soma: <http://www.ease-crc.org/ont/SOMA.owl#>
@prefix dm: <http://knowrob.org/kb/supermarket.owl#>
@prefix mw: <http://www.artiminds.com/kb/metawizard.owl#>

dm:HealingSalve_0 rdf:type dm:HealingSalve
dm:HealingSalve rdfs:subClassOf dul:DesignedArtifact
dm:DMShelfMountingBar_0 rdf:type dm:DMShelfMountingBar
dm:DMShelfMountingBar rdfs:subClassOf mw:Hook
# ...
mw:UR5Robotiq_0 rdf:type mw:UR5Robotiq
mw:UR5Robotiq_0 srdl2-comp:hasBodyPart mw:UR5Arm_0
mw:UR5Robotiq_0 srdl2-comp:hasBodyPart mw:RobotiqGripper_0
mw:UR5Robotiq_0 mw:hasTCPName "tcp"
# ...
mw:hasTCPName rdfs:subPropertyOf soma:hasNameString
# ...
```

Listing 4.1: Example for a KnowRob knowledge base in RDF Turtle syntax (W3C, 2014). The loaded ontologies `supermarket.owl` and `metawizard.owl` are described in greater detail in Sections 4.1.2.3 and 4.1.7.

This extract of the knowledge base (KB) from Experiment 4.1.7 contains information about three individuals: An product (`dm:HealingSalve_0`), a hanger (`dm:DMShelfMountingBar_0`) and a robot (`mw:UR5Robotiq_0`). They represent physical entities that exist in the real-world environment. Moreover, the KB also contains information about them, such as their object types (e.g. `dm:HealingSalve`) or their components (e.g. `mw:RobotiqGripper_0`, referring to the gripper which, together with the robot arm `mw:UR5Arm_0`, makes up the robot `mw:UR5Robotiq_0`). Listing 4.1 illustrates several core features of KnowRob's RDF triple store.

First, entities, their types, their properties and relations between them are defined in terms of ontologies, such as the Socio-Physical Model of Activities (SOMA) (Beßler et al., 2021), DOLCE+DnS Ultralite (DUL) (Presutti and Gangemi, 2016), and others. The KB comprises the RDF representation of the complete ontology hierarchy, including upper-level ontologies. By including the complete ontology hierarchy in the triple store, the KB contains common-sense knowledge defined in higher-level ontologies such as DUL. Trivial-seeming relations such as '`dul:DesignedArtifact rdfs:subClassOf dul:PhysicalObject`' ("every designed artifact is a physical object") enable the definition of high-level reasoners that exploit the transitivity of object properties, permitting inferences such as "if every physical object has a mass, then every designed artifact must also have a mass".

Second, Both TBox and ABox information is stored in the same KB. The knowedge base contains e.g. information about class hierarchies, such as the TBox assertion '`dm:HealingSalve rdfs:subClassOf dul:DesignedArtifact`' declaring that the product category "healing salve" belongs to the broader category of designed artifacts (as opposed to other object categories, such as biological objects or substances). It also contains ABox assertions about indiviuals (e.g. '`dm:HealingSalve_0 rdf:type dm:HealingSalve`', the object `dm:HealingSalve_0` is an instance of the object category "healing salve") and the relationships between them (e.g. '`mw:UR5Robotiq_0 srdl2-comp:hasBodyPart mw:UR5Arm_0`', the robotic arm `mw:UR5Arm_0` is a part of the particular robot `mw:UR5Robotiq_0`).

Third, the knowledge base can be dynamically modified at runtime, and new ABox and TBox assertions can be added at any time. This is crucial for learning from experience, making inferences in dynamic environments, or implementing planners that modify a shared state.

Spatial data and semantic maps One of the core distinguishing features of KnowRob is that it is a *hybrid* symbolic-subsymbolic reasoning system. At the level of the knowledge base, this reflects in the integration of additional data structures for subsymbolic data, which is semantically connected to the symbolic data structure (the RDF triple store). One such data structure is the *transformation (TF) database*, a data structure for timestamped spatial transformations, including both the poses of objects in the environment as well as the transformations between the robot links. Both the RDF triple store and the TF database are implemented as key-value stores sharing the same database technology (MongoDB, 2023). Poses in the TF database and objects represented the RDF triple store are linked by the *semantic map*, a URDF representation of the environment as well as the agents acting in it, such as robots, which defines the (initial) poses of all known environment objects with respect to some reference coordinate system, forming a TF tree with the world origin at its

◁ *TF database*

◁ *Semantic map*

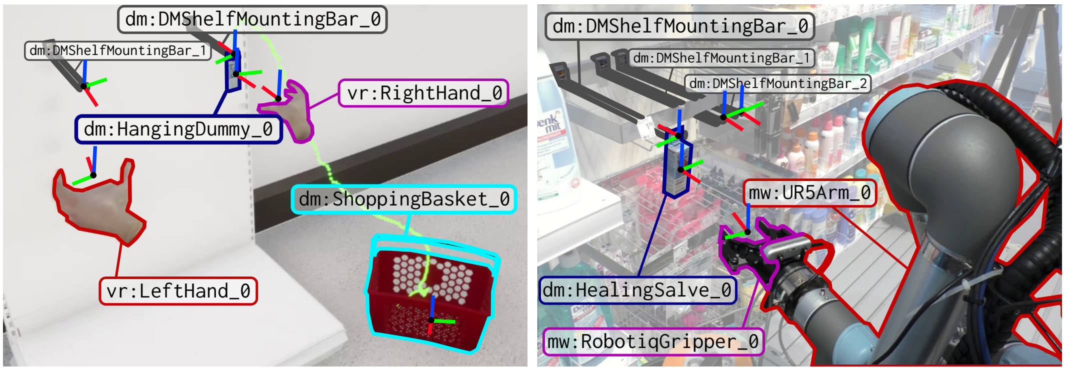


Figure 4.3: Semantic maps of VR (left) and real-world (right) supermarket environments.

root. Moreover, the semantic map defines kinematic constraints on objects, such as connections to other objects in kinematic chains, which can be accessed by hybrid reasoning systems. The semantic map links the TF database to the symbolic KB by associating string identifiers with the links and joints of objects and agents, which are in turn linked to individuals in the RDF triple store via properties such as `soma:hasNameString`. In the triple store shown in Listing 4.1, the TCP of robot `mw:UR5Robotiq_0` is linked to the literal “`tcp`” via `mw:hasTCPName`, a subproperty of `soma:hasNameString`. The string “`tcp`” is a key for lookup in the semantic map and TF database, permitting hybrid reasoners to access the robot’s TCP pose at a given time. Figure 4.3 illustrates the semantic map for both a VR and real-world environment in the context of a retail assistance task. By using the RDF triple store, TF database and semantic map, hybrid reasoners can perform spatio-temporal reasoning, inferring e.g. that an object was grasped by the robot or contained or otherwise constrained by another object. Tenorth and Beetz (2017) and Pangercic et al. (2012) describe the semi-symbolic knowledge representation of KnowRob in greater detail.

Prolog knowledge base An additional central feature of KnowRob is the tight integration between the semisymbolic knowledge base and the Prolog language and reasoning framework (Wielemaker et al., 2012). KnowRob permits the querying and modification of the KB via Prolog at runtime. To that end, KnowRob provides a MongoDB backend for Prolog, which exposes the RDF triple store and TF database to Prolog. Via the *KnowRob query language*, a Prolog language extension that provides Prolog predicates for accessing and modifying the KB. The predicate `kb_call(P)` is `true` if predicate or list of predicates `P` is true for the current state of the KB. `kb_project(P)` is always `true` and has the side effect of inserting predicate `P` into the KB. `kb_unproject(P)` is always `true` and retracts `P` from the KB. The predicate `holds(S, P, O)` is `true` if the subject-predicate-object triple ‘`S P O`’ is in the RDF triple store. In combination, `holds`, `kb_project`, `kb_unproject` and `kb_call` permit the dynamic assertion, retraction and retrieval of knowledge by

Prolog programs, and natively expose the RDF triple store to the Prolog runtime, enabling Prolog reasoners to perform unification over variables grounded in the RDF triple store. Moreover, it permits the initialization and modification of parts of the KB at runtime, which is particularly useful for knowledge that is hard to define in OWL. Consider the following example:

```
 1 ?- kb_project([holds('Object_0', rdf:'type', owl:'NamedIndividual'),
 2                holds('Object_0', rdf:'type', dm:'LaundryDetergent')]).
 3 true.
 4
 5 ?- kb_call(holds('Object_0', rdf:'type', T)).
 6 T = dm:'LaundryDetergent' ;
 7 T = owl:'NamedIndividual'.
 8
 9 ?- kb_unproject(holds('Object_0', rdf:'type', dm:'LaundryDetergent')).
10 true.
11
12 ?- kb_call(holds('Object_0', rdf:'type', T)).
13 T = owl:'NamedIndividual'.
```

Listing 4.2: Code example for dynamic knowledge assertion and retraction in the KnowRob knowledge base.

`kb_project` on line 1 asserts that `Object_0` is an individual of type `dm:LaundryDetergent` into the knowledge base by inserting the corresponding triple into the RDF triple store. Namespace expansion of e.g. ‘`rdf:`’ into the corresponding URLs is handled by the implementation of `kb_project`. `kb_call` on line 5 retrieves the asserted knowledge from the knowledge base by unifying `P` with the the triples. After the call to `kb_unproject` on line 9, the knowledge base no longer contains the fact that `Object_0` is of type `dm:LaundryDetergent`.

The KnowRob query language introduces two new operators that simplify the projection and retrieval of knowledge for complex predicates. The *projection operator* +> replaces the Prolog operator `:-`, asserts all statements made in the body of the predicate into the semisymbolic knowledge base and makes the predicate available for use with `kb_project`. The *retrieval operator* ?> grounds all statements made in the body of the predicate in the semisymbolic knowledge base, and makes the predicate available for use with `kb_call`. The combined operator ?+> makes the predicate available for both projection and retrieval. Consider the following example:

```
has_time_interval(Event, StartTime, EndTime) +>
    new_iri(Interval, dul:'TimeInterval'),
    has_time_interval(Event, Interval),
    holds(Interval, soma:'hasIntervalBegin', StartTime),
    holds(Interval, soma:'hasIntervalEnd', EndTime).

has_time_interval(Event, StartTime, EndTime) ?>
    has_time_interval(Event, Interval),
    holds(Interval, soma:'hasIntervalBegin', StartTime),
    holds(Interval, soma:'hasIntervalEnd', EndTime).
```

Listing 4.3: Code example for predicates defined with KnowRob’s projection and retrieval operators.

The predicate `has_time_interval` associates an `Event` with a start and end time. The respective definitions of projection and retrieval differ in that for projection, the prolog variable `Interval` must first be assigned an Internationalized Resource Identifier (IRI), which identifies it in the RDF triple store. Subsequent calls to `kb_call(has_time_interval('Event_0', StartTime, EndTime))` and `kb_project(has_time_interval('Event_0', 10.2, 11.5))` retrieve or project a start and end time for a given event `'Event_0'`. Use of the projection and retrieval operators permit the creation of complex reasoning routines that read from and write to the semisymbolic knowledge base, and enable seamless interaction with native Prolog programs: Prolog variables grounded through retrieval are regular prolog variables, and `kb_call` and `kb_project` support unification and can be used like native Prolog predicates. MetaWizard's reasoners for task abstraction and program synthesis are largely implemented as Prolog predicates using the KnowRob query language extension (see Section 4.1.4).

4.1.2.2 An Ontological Representation of Common-Sense and Domain Knowledge

MetaWizard represents both symbolic domain knowledge as well as commonsense knowledge as ontologies. When loaded, they are converted into an RDF representation by KnowRob and are made available to reasoners as part of the semi-symbolic knowledge base. When designing ontologies, a principle of parsimony was followed, and only entities that were actually needed to solve a practical use case (here, Experiment 4.1.7) were modeled as ontological concepts, without loss of generality: Due to inheritance from general-purpose ontologies such as the Socio-Physical Model of Activities (SOMA) (Beßler et al., 2021) and DUL (Presutti and Gangemi, 2016), the ontologies can be populated with additional concepts in the future, should they be needed for different use cases.

Domain ontology for robot programming In Alt et al. (2023), my colleagues an ontology for knowledge-driven robot programming (`metawizard.owl`) is introduced that provides knowledge and fundamental concepts required for robot program synthesis. It extends SOMA, which is itself a domain ontology for everyday activities with a focus on robotics and and which models high-level, general-purpose concepts such as actions, processes, states or objects (e.g. `soma:PhysicalAction`, `soma:Physical Process`, `soma:State` and `soma:PhysicalObject`), as well as relations between them (e.g. `soma:hasAction`, `soma:isEndLinkOf`). The robot programming ontology provides robot programming-related concepts at the finer level of granularity required by reasoners for practical program synthesis problems:

- It fleshes out the **SOMA role hierarchy** with concepts such as `mw:GripperRole`, `mw:Supporter` (children of `soma:LocatumRole`, the active parts in a relation), `mw:GraspedObject` or `mw:SupportedObject` (children of `soma:RelatumRole`, the passive parts in a relation), which permit the specification or inference of the semantics of manipulation actions with respect to the participants (robots and objects) involved in them.

- It adds **general-purpose object categories** not provided by SOMA, such as `mw:Hook`, or **object features**, such as `mw:Base`, `mw:GraspPoint`, `mw:Hole` or `mw:Tip` (children of `soma:RelevantPart`) and corresponding object properties such as `mw:hasGraspPoint` (subproperty of `dul:associatedWith`).
- It defines concepts for **robot kinematics** that are not modeled in SOMA at the required level of granularity, such as `mw:IndexFinger` and `mw:Thumb` (children of `mw:soma:Finger`) for a two-finger gripper, as well as corresponding object properties such as `mw:hasFinger` (subproperty of `dul:hasPart`).

Application ontology for synthesis of skill-based robot programs Tasks in SOMA are defined at a very abstract level contains only very high-level tasks such as `soma:EndEffectorPositioning`, `soma:PickingUp` etc. KnowRob uses the ARTM for program synthesis (see Section 4.1.5), which defines a broadly useful set of high-level tasks and low-level skills. An ontology of the ARTM (`artm.owl`) for use with KnowRob is proposed (Alt et al., 2023). It defines both primitive and composite task definitions: Primitive tasks such as `artm:MoveToPoint`, `artm:Grasp` or `artm:RelativeMotion` correspond one-to-one to robot skills, here `Move to Point`, `Grasp` or `Move Linear Relative`. Composite tasks such as `artm:InsertHoleOntoPeg` correspond to complex task sequences, here an approach-search-insert sequence for a variant of a peg-in-hole task, where a workpiece with a hole is threaded onto a peg. A total of 27 different manipulation tasks are represented in the ontology.

Ontological model of actions, tasks and situations DUL conceptualizes actions as *events*, entities that occur at a given time or during a given time interval. `dul:Event` is a composite and can be related to other `dul:Events` via the `dul:hasConstituent` relation, one of the general-purpose object properties for composition in DUL. Objects (`dul:Object`), but also humans (`dul:Person`) can be *participants* in events. SOMA adds the additional nuance that actions (`dul:Action`) are *intended* in that they are disjoint from accidents (`soma:Accident`). As a consequence, actions have at least one agent as participant, such as a person or robot (`dul:PhysicalAgent`). Actions give rise to *situations* (`dul:Situation`), relational contexts in which the world and the objects and agents in it satisfy some description (`dul:Description`). In SOMA, situations are modeled to *manifest in* (`soma:manifestsIn`) events, such as actions and states (`dul:State`).

◁ *Actions*

◁ *Situations*

`dul:Task` is a `dul:EventType` that *classifies* (`dul:classifies`) an event. Actions *execute* tasks.[1] The semantic difference between actions and tasks in DUL is

◁ *Tasks*

---

[1] Note that the ontological definition of `dul:Task` differs from the definition of a task in Section 2.2.2.1. This chapter, the DUL definition of a task is used.

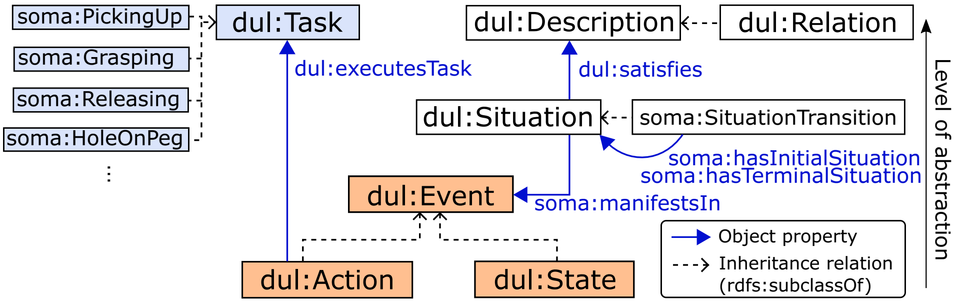


Figure 4.4: The semantics of events, tasks and situations modeled by SOMA (Alt et al., 2023).

one of abstraction: Actions are events that have occurred, are occurring or will occur in the world, while tasks are “social objects” that humans or other cognitive agents use to conceptualize, communicate and reason about actions. For example, a reaching task may be executed by a robot action of simply extending its arm, or performing a planned motion in a cluttered environment. Section 4.1.5 proposes a reasoning mechanism for solving this *task grounding* problem: Finding a mapping from abstract tasks to concrete robot actions to be performed in the real world, and relating task parameters to action. The inverse problem, determining what task is executed by a given action, is also inherently ambiguous: A may extend its arm to perform a reaching task, but also to push an object or point at something. Section 4.1.4 proposes a reasoning mechanism to solve this *task abstraction* problem. Program synthesis with MetaWizard hinges on finding effective solutions to interpret the tasks implied by e.g. human demonstrations, which are events performed by the human in a given environment, and translating them to a different set of actions to be performed by a robot in a different environment. Figure 4.4 illustrates the SOMA model of events, tasks and situations.

Task grounding ▷

Task abstraction ▷

4.1.2.3 A Hybrid Representation of Task Knowledge

To devise a knowledge-based reasoner capable of synthesizing executable robot programs, the semantics of tasks must be well-defined. The SOMA and ARTM ontologies define what tasks *exist*, but do not define what the tasks *mean* in terms of robot behavior or changes to world state: How is the world different after the robot has grasped an object? What does it mean to have grasped successfully, and how does grasping affect downstream tasks? A *hybrid* task representation, that defines the task taxonomy in an ontology and defined task semantics as Prolog predicates over concepts from the ontology, promises to concisely represent complex task semantics (Alt et al., 2023). The specification of task semantics in terms of Prolog predicates facilitates the straightforward implementation of reasoners for task abstraction and task grounding, similar to how the differentiable shadow program representation facilitates the straightforward implementation of first-order optimizers for robot programs.

MetaWizard follows the established approach in the task planning literature of defining tasks in terms of their preconditions, runtime conditions and postconditions. With the ontological task taxonomy loaded into the semisymbolic knowledge base, and the integration of the semisymbolic knowledge base in Prolog via the KnowRob query language, tasks such as `soma:PickingUp` can be succinctly defined:

```
satisfies_pre(Act, soma:'PickingUp') :-
    % precondition 1: Some object O1 grasped
    has_initial_situation(Act, S1),
    object_grasped(O1, Gripper, S1),
    % precondition 2: O1 supported by something
    has_initial_situation(Act, S2),
    object_supported(O1, O2, S2).

satisfies_run(Act, soma:'PickingUp').

satisfies_post(Act, soma:'PickingUp') :-
    % postcondition 1: O1 still grasped
    has_initial_situation(Act, S1),
    object_grasped(O1, Gripper, S1),
    has_terminal_situation(Act, S2)),
    object_grasped(O1, Gripper, S2),
    % postcondition 2: O1 not supported anymore
    forall(has_terminal_situation(Act, S3)),
        \+ object_supported(O1, O2, S3)).
```

Listing 4.4: Definition of a `soma:PickingUp` task in terms of pre-, runtime and postconditions.

The predicates `has_initial_situation(A, S)` and `has_terminal_situation(A, S)` are `true` if `S` manifests at the beginning or end of action `A`, respectively (see Figure 4.4). They, as well as the predicates `object_grasped` and `object_supported`, are written in the KnowRob query language to perform reasoning over the semisymbolic knowledge base. `satisfies_pre(A, TT)` is `true` if action `A` satisfies the predconditions of task type `TT` (here `soma:PickingUp`), i.e. if the specified set of conditions hold at the beginning of `A`. The preconditions are represented as a conjunction of predicates in the body of `satisfies_pre`. Here, for an action to satisfy the preconditions of `soma:PickingUp`, the object `O1` to be picked up must be grasped and supported by some other object. The postconditions expressed in `satisfies_post` are satisfied if, at the end of the action, `O1` is still grasped, but not supported by any other object anymore – it has been picked up. Note the use of the universal quantifier `forall`: As situations in SOMA are relational contexts ("substrates" over which descriptions, such as relations, are defined (see Figure 4.4)), multiple situations can manifest at the same time, each serving as a context for some subset of the relevant descriptions of the scene. When specifying that some condition is *not* to hold at the beginning, during or after an action, it is necessary to specify that this condition is false for all situations in that time scope. `soma:PickingUp` does not have any runtime conditions, and `satisfies_run` is trivially `true`. For an example with salient runtime conditions, consider the definition of `artm:Sliding`:

```
satisfies_pre(Act, artm:'InsertHoleOntoPeg') :-
    % precondition 1: Some object O1 grasped
    has_initial_situation(Act, S1),
    object_grasped(O1, Gripper, S1),
    % precondition 2: O1 is not touching anything other than the gripper
    forall(has_initial_situation(Act, S2),
            objects_touch(O1, O2, S2) -> O2 = Gripper; true).

satisfies_run(Act, artm:'InsertHoleOntoPeg') :-
    % runtime condition 1: O1 grasped
    has_runtime_situation(Act, S1),
    object_grasped(O1, Gripper, S1).

satisfies_post(Act, artm:'InsertHoleOntoPeg') :-
    % postcondition 1: Grasped object supported by a Peg
    has_terminal_situation(Act, S1),
    object_grasped(O1, Gripper, S1),
    has_terminal_situation(Act, S2),
    object_supported(O1, O2, S2),
    kb_call(instance_of(O2, mw:'Peg')).
```

Listing 4.5: Definition of an `artm:InsertHoleOntoPeg` task in terms of pre-, runtime and postconditions.

`artm:InsertHoleOntoPeg` represents an insertion task in which an object is slid onto a peg. In a retail assistance use case, this corresponds e.g. to a product being placed on a rod-shaped hanger. Here, the preconditions are that an object is grasped, and is currently not in contact with anything but the gripper. The postconditions are that the object is still grasped, but now supported by a peg. The runtime condition is that the object remains in the gripper. Note that the pre-, runtime and postconditions do not impose constraints on how the task is being achieved, but rather on what relations between objects are to hold.

```
object_supported(Supportee, Supporter, Situation) ?+>
    binary_related(Supportee, Supporter, Situation,
                   mw:'SupportRelation', soma:'SupportedObject',
                   soma:'Supporter'),
    Supportee \== Supporter.

binary_related(Thing1, Thing2, Situation, DescriptionType,
               RoleType1, RoleType2) ?>
    is_individual(Description),
    instance_of(Description, DescriptionType),
    satisfies(Situation, Description),
    holds(Description, soma:'hasBinding', Binding1),
    holds(Description, soma:'hasBinding', Binding2),
    Binding1 \= Binding2,
    holds(Binding1, soma:'hasBindingFiller', Thing1),
    holds(Binding1, soma:'hasBindingRole', Role1),
    instance_of(Role1, RoleType1),
    holds(Binding2, soma:'hasBindingFiller', Thing2),
    holds(Binding2, soma:'hasBindingRole', Role2),
    instance_of(Role2, RoleType2).

binary_related_in_situation(Thing1, Thing2, Situation, DescriptionType,
    RoleType1, RoleType2) +>
    ...
```

Listing 4.6: Semantic reasoning routines in the KnowRob query language.

Listing 4.6 illustrates the implementation of the semantic reasoning routines as Prolog predicates with the KnowRob query language, that underpin the task definitions. `object_supported(Supportee, Supporter, Situation)` is true if the `mw:SupportRelation` holds between `Supporter` and `Supportee` in `Situation`. `object_supported` affords both querying and projection. `binary_related(Thing1, Thing2, Situation, DescriptionType, RoleType1, RoleType2)` is `true` if the relationship between entities `Thing1` and `Thing2` satisfies some description of a given type (such as `mw:SupportRelation`), in a given `Situation`. `binary_related` illustrates the way in which SOMA models the roles participants play in actions: In a description of a situation, participants can be associated (`soma:hasBinding`) with a role, which they play in this situation. In the terminal situation of the action of hanging a product onto a hanger, for example, the peg plays the role of the supporter (a subclass of `soma:LocatumRole`, the active part of an interaction), while the hanging object plays the role of the supportee (a subclass of `soma:RelatumRole`, the passive part of an interaction).

4.1.2.4 Narrative-Enabled Episodic Memories

This chapter proposes a framework for knowledge-driven program synthesis from human VR demonstrations. In this context, Section 4.1.4 proposes an approach to extract an understanding of abstract tasks from concrete experience data. As an underlying data structure, MetaWizard leverages Narrative-Enabled Episodic Memories (NEEMs), a semisymbolic representation of robot or human experience data. NEEMs are semantically enriched execution traces of agents performing actions in an environment (Beetz et al., 2018). Individual actions, motions, objects in the environment and the agents themselves are stored in the semisymbolic knowledge base, affording the same kinds of semisymbolic reasoning over robot experiences as e.g. task (see Section 4.1.2.3) or commonsense knowledge (see Section 4.1.2.2).

◁ *NEEMs*

A NEEM is a data structure with three components:

1. A **semantic map** of the environment, comprising the robot, environment objects and other agents, as well as any a priori known relationships between them, such as kinematic constraints or affordances;
2. an **event timeline**, that represents the performed actions, the situations manifesting in them as well as the involved participants and their roles in terms of the above described model of actions and situations;
3. and a **TF database** containing the timestamped positions and orientations of task-relevant frames such as object origins, the robot end-effector or the hands of human agents, throughout the entire episode.

The structure of NEEMs directly mirrors that of the KnowRob semisymbolic knowledge base. As such, NEEMs can be inserted directly into the knowledge base and made available to reasoners or learning algorithms. The event timeline of a NEEM collected in VR (see Section 4.1.3) is shown in Figure 4.5. It consists of a sequence of actions (orange) as well as states (blue) with concrete start and end timestamps, here displayed along the X axis. In SOMA, actions are events in which participants effect some change in the world, while states are events in which some properties of or relation between participants remains static. Via their start and end timestamps actions and states can be associated with the corresponding motions performed by the agents involved in them. In the MetaWizard system, NEEMs are further semantically enriched by adding information about interactions between agents and objects using the situation-description model outlined in Section 4.1.2.2 (Alt et al., 2023).

Each state in a NEEM manifests at least one situation, which satisfies some description about the world. In Figure 4.5, an packet of healing salve is taken out of a shopping basket by a human VR agent and placed onto a hanger. The state `soma:State_PITZNFDG`, for example, represents the state of contact between the hanger and the object. It begins when the object first touches the hanger and persists until the end of the episode, as the object remains on the hanger. It manifests a situation in which a contact relation holds between the object and the hanger; in that relation, both hanger and object act as `soma:ConnectedObjects`. Most NEEMs contain temporally overlapping states, as generally more than one relation hold at the same time. States can manifest more than one situation; the semantics are such that all situations manifesting in a state do so for the complete duration of the state.

Actions are associated with a `soma:SituationTransition`, itself a type of situation, which relates a set of situations at the beginning of the action (the *initial situations*) to a set of situations at the end (the *terminal situations*). The initial situations for an action are the union of all situations manifesting in states that are ongoing at the beginning of the action. Terminal situations are the union of all situations manifesting in states ongoing at the end of the situation. Along with a set of `soma:SituationTransitions` and the associated initial and terminal situations, actions are associated with *runtime situations*, manifesting in states which are ongoing for the entirety of the action. Note that a given situation may be initial, runtime and terminal situation of an action. The predicates `has_initial_situation(A, S)`, `has_runtime_situation(A, S)` and `has_terminal_situation(A, S)` used throughout the semantic task definitions (see Listing 4.5) yield these situations for a given action `A`.

4.1.3 From VR Human Demonstrations to NEEMs

The central objective of MetaWizard is to permit humans to intuitively instruct robots *what* to do, without requiring a specification of *how* to do it – for a given task, the reasoning system should infer what actions the robot is to perform, in what order and with what parameterization. This permits the user to program robots

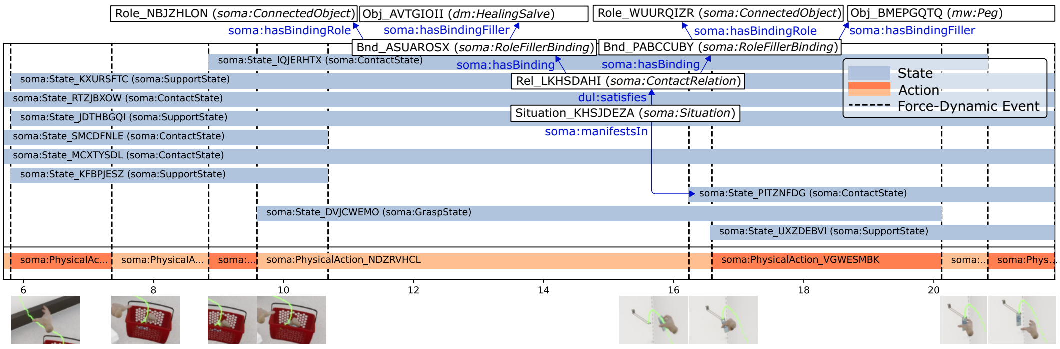


Figure 4.5: Event timeline of a VR NEEM (Alt et al., 2023). It associates a series of states (blue) and actions (orange) with semantic information, such as which objects were in contact with each other in which situations, and what roles objects and agents played in interactions. Time progresses from left to right along the X axis.

e.g. by demonstrating the task in VR. Compared to real-world demonstrations, VR demonstrations permit to record the complete ground-truth state of the agent and environment, permitting the VR engine to parse e.g. contact relations between objects without the need for sophisticated perception algorithms. Moreover, as the environment is virtual, users can demonstrate tasks for arbitrary environments, even if they do not have access to the real-world physical environment. MetaWizard uses NEEMs to represent user demonstrations. VR demonstrations are recorded in Unreal Engine 4 (Epic Games, 2019) using the RobCoG framework (Haidu et al., 2018). RobCoG permits users to record human demonstrations in realistically rendered virtual settings with simulated physics, permitting the demonstration of contact-rich physical interactions with the environment, such as threading an object onto a hanger (see Experiment 4.1.7). Haidu and Beetz (2021) present a pipeline for the automatic segmentation and semantic annotation of human VR demonstrations, which is used to convert low-level VR data into a semantically annotated *execution trace*. Using a grammar over force-dynamic events such as contacts, detected from the simulated force measurements and collision models provided by the VR environment, semantically meaningful state and action types are detected. For example, if an object is in contact with another object and has near-zero vertical velocity, it is detected to be in a "supported-by" state. If an object is moved while having contact only with the VR avatar, a "transporting" action is detected. The execution trace contains the detected states and actions, along with the corresponding time intervals, as well as the timestamped trajectories of all moving objects in the scene. To convert the execution trace provided by RobCog into a NEEM, a semantic parser has been developed and is available open-source[2] (Alt et al., 2023). The parser traverses the execution trace and creates a NEEM according to the following algorithm:

◁ *Execution trace*

---

[2] https://github.com/ease-crc/vr-neem-converter

1. Construct the semantic map: Assert individuals for the human VR avatar and all environment objects into the knowledge base, and associate corresponding kinodynamic information such as object meshes and physical properties, if available.
2. Assert an individual for the top-level task (e.g. `artm:PickAndPlaceTask`).
3. For each state in the execution trace, assert an individual for the semantically equivalent SOMA state type in the knowledge base, as well as the corresponding situation, participants and roles.
4. For each action in the execution trace, assert an individual for the semantically equivalent SOMA action type in the knowledge base, as well as the corresponding participants.
5. Every state change in the must have been brought about by some action. Traverse the timeline and assert a new anonymous action (an individual of type `soma:Action`) whenever a state begins or ends, unless there is already an action ongoing during that interval. After this step, there is exactly one typed or anonymous action ongoing at every point in time (see Figure 4.5 (orange)).
6. For each action in the knowledge base, assert the corresponding situation transition.

The resulting knowledge base, together with the semantic map and TF database, constitutes the NEEM, a semantically rich, semisymbolic representation of a human demonstration. Note that the semantic annotation is necessarily incomplete. Beyond the high-level task, it does not contain information on the task level: The tasks, or human intent, represented by the demonstrated actions must be inferred, or interpreted, in light of commonsense and domain knowledge. Moreover, the action sequence contains anonymous actions, whose semantics are exclusively defined via the associated situation transitions. For example, when a contact relation of an object with the shopping basket ends, it is unknown to the VR NEEM parser by what exact action that contact was ended – whether the object was picked up by a person or fell out of the shopping basket, for example. Such semantic information can be inferred by subsymbolic reasoning over the NEEM, in the context of commonsense and domain knowledge.

4.1.4 Task Abstraction

The inference of robot programs given human VR demonstrations involves the cognitive task of transferring the actions demonstrated by a human avatar in a simulated environment to a robot with possibly different kinematics, a constrained set of skills, and a different environment. For program synthesis, then, the objective cannot be to directly mimic the demonstrated *actions* – rather, the synthesized robot program should execute the demonstrated *tasks*. To that end, a program synthesis system must extrapolate the intended tasks from the demonstrated actions, in a process of *semantic lifting*: The demonstrated events must be interpreted in light of the available background knowledge to yield a series of tasks, along with the corresponding task parameters, which can in turn be *grounded* again to produce an executable robot program, which results in a series of robot actions that execute the task in the environment at hand (see Section 4.1.5).

◁ *Semantic lifting*

MetaWizard proposes a simple, but effective method of task abstraction (Alt et al., 2023). With the hybrid task definition outlined in Section 4.1.2.3 and NEEMs as a data structure for human demonstrations, both task knowledge and the demonstration are loaded into the same knowledge base and available to the KnowRob KR&R engine. At the level of an individual action, task abstraction can be encapsulated in the following predicate:

```
interprets_to(Act, Tsk) :-
    has_task_type(Tsk, TskType),
    satisfies_pre(Act, TskType),
    satisfies_run(Act, TskType),
    satisfies_post(Act, TskType),
    parameterize_task(Act, TskType, Tsk).
```

Listing 4.7: The predicate `interprets_to` enables the abstraction from a given action to the intended task via the Prolog unification algorithm.

`interprets_to` unifies a variable `Tsk` with a task individual such that a given action `Act` plausibly executes `Tsk`, in accordance with the knowledge base. To that end, `has_task_type` unifies `TskType` with an candidate task type (subclass of `dul:Task`, e.g. `soma:Grasping`); then, if demonstrated action `Act` satisfies the pre-, runtime and postconditions for `TskType` (see Listing 4.5), the task is *parameterized* with high-level task parameters. By backtracking, the unification algorithm successively generates candidate task types until pre-, runtime and postconditions are satisfied. Note that task abstraction is not one-to-one: As the knowledge base is, by practical necessity, incomplete (it is impossible to specify tasks without ambiguity), a demonstrated action may execute several different task types. Successive queries to `interprets_to` eventually yield parameterized task individuals for all tasks. MetaWizard performs task abstraction lazily, i.e. queries `interprets_to` only if task grounding or program synthesis failed and a different candidate task is required.

At the task abstraction stage, tasks are parameterized at the abstract level. The task type `soma:Grasping`, for example, has the grasped object as a task parameter. The determination *which* object from the target environment is to be grasped is left to the task grounding pipeline (see Section 4.1.5). Rather, an anonymous individual of the same object type as the object performing the role of the grasped object (`soma:RelatumRole`) is asserted into the knowledge base and attached to the generated task via the `dul:hasParameter` object property. Unifying `interprets_-`

`to(Act, Tsk)` for every action `Act` in the demonstration yields a candidate task sequence for the complete demonstration:

```
actions_interpret_to(ActionSeq, TaskSeq) :-
    findall(Act,
            (member(Act, ActionSeq),
            interprets_to(Act, Tsk)),
            TaskSeq).
```

Listing 4.8: Abstraction from a demonstrated sequence of action to a candidate task sequence via the Prolog unification algorithm.

Repeated evaluation of `actions_interpret_to` yields all possible task sequences `TaskSeq` that action sequence `ActionSeq` can be interpreted to execute, given the knowledge base.

4.1.5 Task Grounding

To synthesize an executable robot program, the generated task sequences must be grounded to the robot and environment at hand. For a `soma:Grasping` task, for example, task grounding must identify the target object in the real-world environment, and infer all other parameterizations such as collision-free approach and depart motions required to execute the task. MetaWizard first translates the task sequence to an *underspecified plan*, which is then grounded by reasoning over a simulated environment representation and real-world sensor data.

Underspecified plan ▷

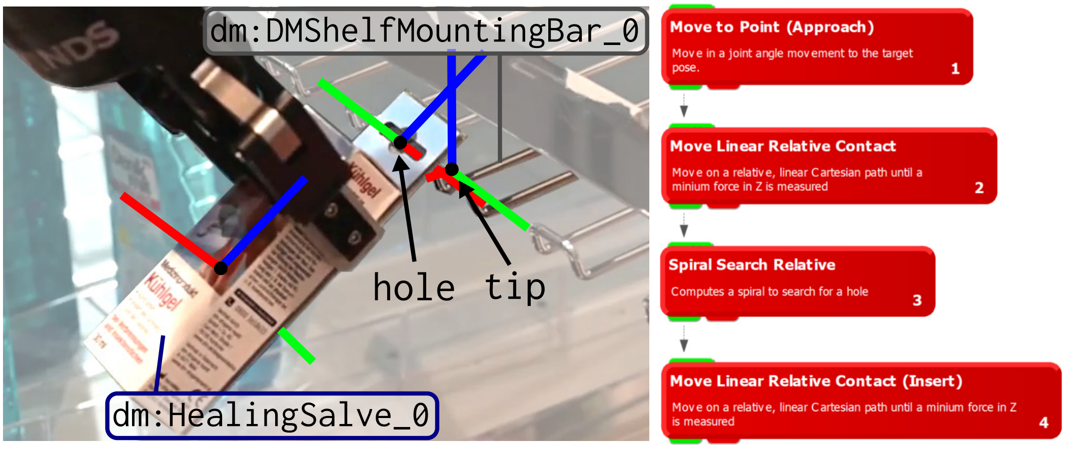
4.1.5.1 Underspecified Plans

The CPL is a semisymbolic robot program representation, that combines a textual syntax with the semantics of task-based programming (Beetz et al., 2010). The CPL represents robot programs, or *plans*, in the Cognitive Robot Abstract Machine (CRAM) cognitive architecture (Beetz et al., 2023). The CPL is designed explicitly to accommodate the fact that robot programs are, by necessity, *underspecified* when they are created: Some information required for execution, such as the precise location of objects in the scene, may not be available at planning time. Certain decision-making processes must be performed at runtime, either through reasoning over a knowledge base (e.g., determining the opening direction of a door) or through perceptual processes (e.g., locating a target object in the environment). At the highest level, a CPL plan is a data structure composed of *designators*, symbolic representations of entities which are grounded, or *resolved*, with concrete symbolic or subsymbolic meaning at runtime (McDermott, 1991). CPL action designators are conceptually similar to subprograms in the ARTM, and contain, in turn, other designators, such as *object designators*, *location designators*, *motion designators* or lower-level action designators. Executing a CRAM plan corresponds to resolving the designator hierarchy. MetaWizard leverages CPL designator resolution as a task grounding mechanism (Alt et al., 2023).

```
class HoleOnPegActionDescription(ActionDesignatorDescription):
"""
'Inverse Peg-In-Hole': Inserting an object with a hole onto a peg
"""
def __init__(self, hole_obj: ObjectDesignator, peg_obj: ObjectDesignator, resolver="grounding"):
    self.hole_obj = hole_obj
    self.peg_obj = peg_obj
```

```
def ground_hole_on_peg(desc: HoleOnPegActionDescription):
"""
Ground HoleOnPeg
"""
# Ground possibly ungrounded ObjectDesignators hole_obj and peg_obj
peg_obj_grounded = desc.peg_obj.reference()
hole_obj_grounded = desc.hole_obj.reference()

# HoleOnPeg is decomposed into 4 motions:
# 1. Move to Point (Approach)
hole_pose_world = Frame(affine_from_pycram_pose(feature_pose(hole_obj_grounded["name"], "hole")), WORLD_FRAME)
world_T_tcp = hole_pose_world  # Set TCP to CS of hole
world_T_tip = Frame(affine_from_pycram_pose(feature_pose(peg_obj_grounded["name"], "tip")), WORLD_FRAME)
# Approach 2 cm above tip
approach_pose_tip = Frame(transform_from(np.eye(3), [0.0, 0.0, 0.02]), world_T_tip)
approach_pose_world = pycram_pose_from_affine(approach_pose_tip.affine_world())
# ...
# 2. Move Linear Relative Contact
# ...
# 3. Spiral Search Relative
# ...
# 4. Insert
if instance_of(peg_obj_grounded["name"], "mw:Hook"):
    # If the peg is bent (hook-shaped), zero side-force Move Linear Relative Contact -> Move Linear Relative to rotate
    # 4.1 Move Linear Relative Contact along -Z of the tip feature
    mlrc_2_motion = [0, 0, -0.01, 0, 0, 0, 1]  # 1 cm in -Z, in TCP coordinates
    mlrc_2_min_force = 1.0  # 1 N
# ...
# Instantiate the skill sequence
yield super(HoleOnPegActionDescription, desc).ground()
```

Listing 4.9: PyCRAM description of a `HoleOnPeg` designator for “inverse” peg-in-hole insertion, and abbreviated grounding routine to create and parameterize an executable ARTM skill sequence.



Move to Point (Approach)
Move in a joint angle movement to the target pose. 1

Move Linear Relative Contact
Move on a relative, linear Cartesian path until a minium force in Z is measured 2

Spiral Search Relative
Computes a spiral to search for a hole 3

Move Linear Relative Contact (Insert)
Move on a relative, linear Cartesian path until a minium force in Z is measured 4

Figure 4.6: Grounded object designators and features for the peg and hole objects (left) and generated ARTM robot program for “inverse” peg-in-hole (right).

Listing 4.9 illustrates PyCRAM designator grounding for an “inverse peg-in-hole” subtask (a `HoleOnPeg` action designator). In the grounding routine (`ground_hole_on_peg`), first the two object designators for the peg as well as the manipulated object are grounded, including the respective locations of their `tip` and `hole` features. Given these features, parameters for lower-level motion designators are derived, such as TCP poses or relative motion targets. Resolution yields a parameterized sequence of ARTM skills to insert the object onto the peg (see Figure 4.6).

Action designators The task representation introduced in Section 4.1.2.3 allows for a direct conversion of candidate task sequences into underspecified plans. As tasks and CPL action designators share the same level of abstraction, conversion is straightforward: For each task in the sequence, one action designator is instantiated, along with object or location designators for the task parameters. For instance, a task of type `soma:PickingUp` involves two objects: the locatum (primary object to be picked up) and the relatum (secondary object supporting or containing the locatum). This task translates directly to a `PickingUp` action designator with one object designator, `target`, corresponding to the locatum. To implement this approach, PyCRAM,[3] the Python implementation of the CPL, has been extended to support 27 different task types covering a range of robotic actions, including prehensile (e.g., `Grasping`, `PickingUp`, `Placing`) and force-controlled manipulation (e.g., `HoleOnPeg`, `Sliding`, `Retracting`). Notably, for each task type defined in the knowledge base, a corresponding PyCRAM action designator is defined.

Grounding object and location designators Grounding an action designator recursively grounds the associated designator hierarchy. `PickingUp`, for example,

---

[3] https://github.com/cram2/pycram

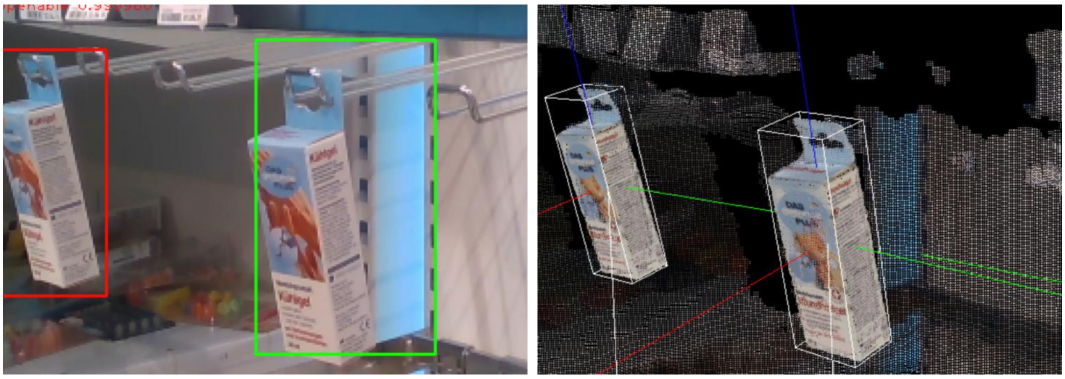


Figure 4.7: Object designator grounding with RoboSherlock: Detected object bounding boxes in the RGB (left) and point cloud (right) images of the environment (Alt et al., 2023).

has two object designators, `target` and `support`. Each object designator, in turn, has several properties, such as an object type and ID, as well as a location designator `pose` designating the pose of the object in the robot workspace. In the context of program synthesis from VR demonstrations, `PickingUp` is underspecified because both object designators are underspecified: While the object type is known, the concrete designated object instance in the real-world execution environment is a priori unknown, and must be determined by some reasoning mechanism. MetaWizard resolves the ID and name attributes of object designators via search for objects of the given object type in the semantic map of the execution environment or, if the corresponding object detector has been configured in the RoboSherlock perception system (Beetz et al., 2015; Mania et al., 2021), searches for candidate objects in a camera image of the real-world environment (see Figure 4.7). RoboSherlock is integrated with the KnowRob KR&R engine, acting as a real-world extension of the knowledge base and permitting queries, such as the existence of objects of a given type, in terms of the KnowRob query language. Likewise, location designators are grounded by lookup of an object's position in the TF database or, if the corresponding detector is available, by querying RoboSherlock. Beyond the location of objects in the environment, a mechanism for grounding *object-relative* location designators, e.g. the relative poses of object features such as screw holes or the tip of a peg, has been implemented.

Grounding action designators Once all associated object and location designators are resolved, the action designator itself is resolved. Action designators are resolved to sequences of motion designators, which represent atomic motions to be executed by the robot. In CRAM, motion designators are resolved by planning motions and executing them dynamically. To ensure that human programmers can view, edit and interact with the robot program in an intuitive manner, a new backend for designator resolution was implemented, which resolves motion designators by transpiling them to ARTM skills (Alt et al., 2023). The RPS robot IDE is then used for collision-free motion planning, 3D visualization, robot code generation and execution, as well as validation and potential editing of the synthesized program by the human robot programmer. A `PickingUp` action designator, for example, is resolved by first resolving its associated `target` object designator; then, a suitable grasp pose on the object is determined. Here, a precomputed grasp pose has been stored as an object feature in the knowledge base and is retrieved; alternatively, a grasp pose can be computed dynamically via a grasp planner. The task `soma:PickingUp` has two postconditions: The target object remains in the gripper and is not in contact with another object. Consequently, the `PickingUp` action designator computes a depart region corresponding to the free-space region around the object with a given padding, which is used to parameterize a `MoveToRegion` motion designator. Finally, the `MoveToRegion` motion designator is resolved to a `Move to Region` ARTM skill, which plans a collision-free motion into the depart region. Skill parameters, such as velocities or setpoints for a force controller, are inferred from the knowledge base. The parameterized skill is inserted into the candidate ARTM program and motion planning is performed, given a collision model of the world reflecting the current state of the knowledge base. After grounding all contained designators, the resulting motions are validated in a kinematic simulator with support for contact-rich motions and grasps, and the postconditions of the action designator are checked. If they are not met (e.g. if the planned motion for `PickingUp` is not collision-free), action designator resolution returns with an error.

Action designator resolution is implemented in such a way as to preserve the hierarchical plan structure; i.e., action designators containing other action designators are resolved to ARTM subprograms, and motion designators are resolved to atomic ARTM skills which are inserted at the appropriate place in the program hierarchy. Like task abstraction, task grounding is inherently ambiguous: A given object designator, for example, could be resolved to several possible objects in the environment. For this reason, action, object and location designators are resolved lazily, yielding one valid grounding at a time. Designator resolution can be performed repeatedly to eventually exhaust all possible groundings. If some designator cannot be resolved, e.g. if an object of the required type cannot be detected in the environment, MetaWizard backtracks and attempts to ground the designator again, until all options are exhausted, in which case it attempts to ground the designator one level above in the designator hierarchy. If no valid grounding for the plan can be found, MetaWizard backtracks through task abstraction until a task sequence (and corresponding plan) is found that can be resolved.

4.1.6 Program Execution

Once all action designators of the plan have been grounded, the completed plan is grounded and a corresponding ARTM source program has been created. That program can be visualized in the RPS robot IDE, read and edited by human experts if

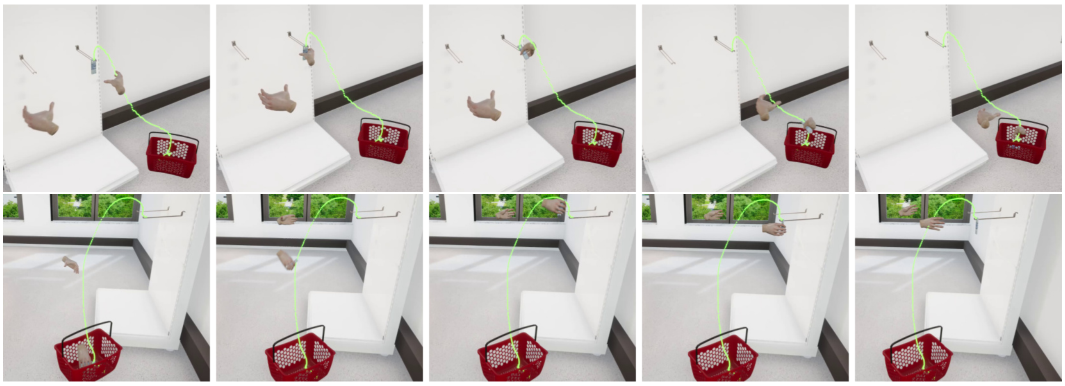


Figure 4.8: VR human demonstrations for force-controlled fetch-and-place (top) and peg-in-hole (bottom) (Alt et al., 2023).

needed, and executed on the robot. The program execution pipeline of MetaWizard consists of four basic steps:

1. **Load the candidate ARTM program and collision environment** into the RPS robot IDE. The collision environment is loaded from the semisymbolic knowledge base according to the semantic map, updated with the real-world object poses grounded by RoboSherlock (Mania et al., 2021).
2. **Visualize the candidate program** in the simulated 3D environment. Program execution continues only if the user approves the visualized simulation.
3. **Compile the candidate program** to executable robot code for the robot hardware at hand.
4. **Execute the generated robot code** on the real robot.

4.1.7 Experiments

To validate the MetaWizard framework on a set of real-world scenarios, a retail assistance use case is considered. The experiment tests the hypothesis that MetaWizard permits the synthesis of robot programs for complex manipulation skills given one single human demonstration of the task in VR. Two force-sensitive fetch-and-place tasks are considered, in which the robot takes an object and places it onto a peg, and vice versa. A quantitative evaluation is conducted under laboratory conditions (see Section 4.1.7.1). For both use cases, qualitative validation is conducted in a realistic supermarket environment (see Section 4.1.7.2).

Application scenarios In a first application scenario, a robot is tasked to take an object off a peg and place it in a basket. This task requires grasping of the object, force-controlled extraction of the object from the hanger, as well as the collision-free placing of the object into the basket. In a second application scenario, a peg-in-hole task is considered, in which an object is threaded onto a peg. This tasks involves grasping of the object, collision-free transfer to the target peg, force-controlled search for the tip of the peg and force-controlled insertion onto the peg. Both scenarios are representative of retail robotics tasks such as shopping assistance and shelf-filling, and were considered in the context of a research project aiming to assist elderly people or people with disabilities in supermarkets.[4]

Application ontology and semantic maps To support the experiment use cases, an application ontology was developed (`supermarket.owl`, prefix `dm`), which defines the required classes and individuals for the experiment. The respective spatial layouts of the real-world and VR supermarket environments are specified in the semantic maps illustrated in Figure 4.3. Five individuals of type `dm:DMShelfMountingBar` are defined, representing five pegs mounted on a shelf, each of which has a feature of type `mw:Tip` denoting the tip of the peg. The pegs, as well as a shopping basket (an individual class `dm:ShoppingBasket`) are present in both the real-world and VR environments, albeit at different locations. The real-world supermarket contains a target object of type `dm:HealingSalve`, a small cardboard box containing a tube of salve for treating burns, which has a feature of type `mw:Hole` representing the hole on a tab by which it is hung on a peg. The VR supermarket environment instead contains an object of type `dm:HangingDummy`, a box-shaped dummy object.

VR human demonstrations For each of the two application scenarios, 30 human demonstrations were collected at the VR lab of the IAI at the University of Bremen. Demonstrations were performed for three variants of `dm:HangingDummy` with object geometries differing in width, height and depth as well as 10 different peg poses. Figure 4.8 shows two of the collected VR demonstrations. All demonstrations are parsed into NEEMs (see Section 4.1.3).

4.1.7.1 Quantitative Evaluation

In a first set of experiments, MetaWizard was used to generate robot programs for all 60 human demonstrations (Alt et al., 2023). To ensure reproducibility and comparability between runs, programs were executed on real hardware under controlled conditions.

Experiment setup The physical experiment setup is shown in Figure 4.9. A UR5 collaborative manipulator[5] with an flange-mounted force-torque sensor[6] and a

---

[4] Research project ILIAS (2019-2022), funded by the German Federal Ministry of Education and Research under grant #01DR19001B.

[5] Universal Robots A/S, Odense, Denmark

[6] ATI Industrial Automation Inc., Apex, USA

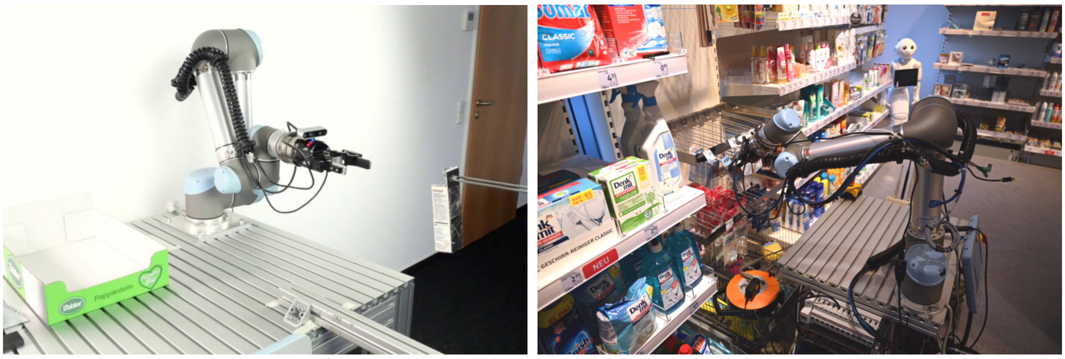


Figure 4.9: Laboratory (left) and realistic (right) supermarket environments (Alt et al., 2023).

	Demonstr.	Abstr.	Grounding		Exec.
	# Demos	# Cands.	Plan succ.	Task succ.	Succ.
O1	10	29	27 (93%)	27 (93%)	26 (90%)
O2	10	60	52 (87%)	44 (73%)	35 (58%)
O3	9	18	18 (100%)	18 (100%)	18 (100%)

Table 4.1: Quantitative results for a force-sensitive fetch-and-place task (Alt et al., 2023). For three different objects (O1, O2, O3), the number of human demonstrations (# Demos), number of generated task sequences (# Cands.), number of successfully grounded plans (Plan succ.), number of robot programs achieving the task objectives in the simulation (Task succ.) and number of successfully executed robot programs (Succ.) are listed.

Robotiq 2F-85 parallel gripper[7] is mounted on a table. The box of healing salve is reinforced with a 3D-printed backplate to withstand repeated grasping, insertion and retrieval from the peg. A metal hanger from a German drugstore chain[8] is mounted to the table at a fixed position.

Each of the 60 human demonstrations is parsed to a NEEM and inserted into a knowledge base. During task abstraction, all possible candidate task sequences are generated and grounded to executable source programs. As the environment is known and reset to its original state after each program execution, object and location designators are grounded with known objects and features from the knowledge base.

Results The results for the force-controlled fetch-and-place scenario are summarized in Table 4.1. Out of 29 VR human demonstrations, a total of 107 candidate task sequences were generated. 79 out of the 107 candidates (74%) were executed

---

[7] Robotiq Inc., Lévis, Canada

[8] dm-drogerie markt GmbH + Co. KG, Karlsruhe, Germany

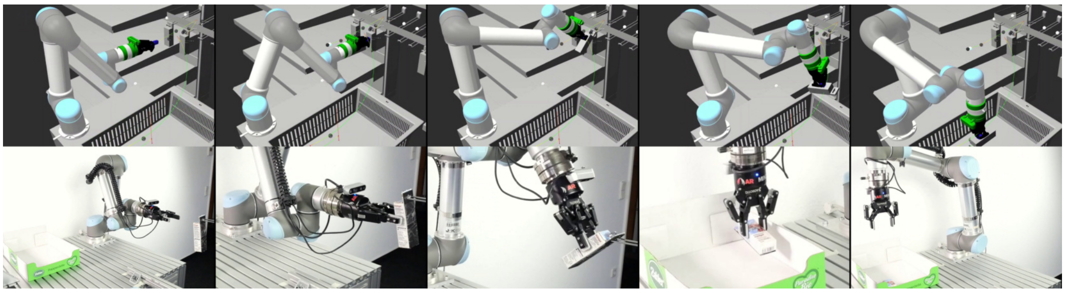


Figure 4.10: Simulated (top) and real-world (bottom) execution of a generated robot source program for force-controlled fetch-and-place in a lab environment (Alt et al., 2023).

successfully. One out of the 30 original human demonstrations was excluded from the analysis due to a glitch in the VR engine, which caused the object to briefly disappear from the avatar's hand, appear on the ground and then re-appear in the avatar's hand. This behavior was inconsistent with the knowledge base and prevented the reasoner from finding a plausible task sequence for the demonstration. The majority of the variance in the generated number of task sequences across objects stems from imperfect detection of force-dynamic events such as contacts or support states, presumably due to imperfect physics simulation. For example, support states are sometimes interrupted and later resume, even though the object was continuously supported. 10 candidate task sequences (9%) fail to be grounded to ARTM programs, largely due to unreachable target poses given the robot and gripper kinematics. An additional 8 candidate programs (7%) fail to place the object into the basket in the simulation. This reflects the fact that MetaWizard does not plan a sequence of tasks to achieve the objectives of a high-level task, but rather finds a sequence of actions that replicates a demonstrated sequence of object relations and force-dynamic events. 74% of all generated task sequences were executed successfully, achieving the task objectives in the real-world environment. This includes at least one task sequence for each human demonstration. For 10 generated candidate programs (9%), execution failed. In all cases, this was due to the challenging physics of the task: The long metal peg flexed when force was applied, and made the robust extraction of the peg with force-controlled motion skills difficult. Moreover, the use of predefined object poses imposed the challenge of robustly grasping the object, as the object pose on the peg varied by up to 1 cm after resetting the environment. An execution of a generated robot source program in both simulated and real-world environments is shown in Figure 4.10.

Scenario 2 considered the complementary task of threading an object onto a peg. The results are summarized in Table 4.2. Given 30 VR demonstrations, a total of 67 candidate task sequences are generated, of which all but 2 are successfully grounded, and two additional task sequences do not achieve the task objectives (the object is placed on the peg) when tested in the simulation. 58 generated robot source programs (87%) are executed successfully in the real-world environment.

	Demonstr.	Abstr.	Grounding		Exec.
	# Demos	# Cands.	Plan succ.	Task succ.	Succ.
O1	10	22	20 (91%)	18 (82%)	18 (82%)
O2	10	20	20 (100%)	20 (100%)	15 (75%)
O3	10	25	25 (100%)	25 (100%)	25 (100%)

Table 4.2: Quantitative results for a force-controlled peg-in-hole insertion task (Alt et al., 2023). For three different objects (O1, O2, O3), the number of human demonstrations (# Demos), number of generated task sequences (# Cands.), number of successfully grounded plans (Plan succ.), number of robot programs achieving the task objectives in the simulation (Task succ.) and number of successfully executed robot programs (Succ.) are listed.

The core challenge was the force-controlled threading of the object onto the peg. Due to flexing during insertion, the position of the tip of the peg varied by up to 1 cm after resetting the environment, making it challenging for force-controlled search skills such as `Spiral Search` to find the peg. Moreover, the tip of the peg points upward in a direction nearly perpendicular to the remainder of the peg, requiring the force-controlled insertion skill to compensate forces and torques while rotating the object around the tip, which was prone to oscillations due to the length of the peg. For each human demonstration, at least one program could be successfully executed. An execution of a generated robot source program in both simulated and real-world environments is shown in Figure 4.11.

Discussion The results demonstrate that executable robot source programs can be synthesized from single human VR demonstrations by reasoning over a semisymbolic knowledge base. MetaWizard bridges the gap between VR and real-world environments by abstracting away from concrete objects and events to infer a plan at the task level, which is then refined, or grounded, to the robot and environment at hand. For this reason, the same mechanism, without any algorithmic changes or even learning, could be used to infer robot programs for different object geometries and peg poses. The results highlight that for each successfully grounded and executed task sequence, several other candidate task sequences are generated for which grounding, simulation or execution fails. This is evidence of a fundamental trade-off facing program synthesis approaches that rely on symbolic knowledge representations. The knowledge base must find a balance between generality and specificity: It must be general enough to cover all relevant task variants or noise in e.g. the human demonstrations, while being fine-grained enough to afford generation of robot programs that solve a given, specific task with the required degree of precision. A comparatively general definition of task semantics, for example, permits to cover a wide range of task variants with a compact knowledge base, but may be too permissive and lead to the generation of “false positives”, task sequences

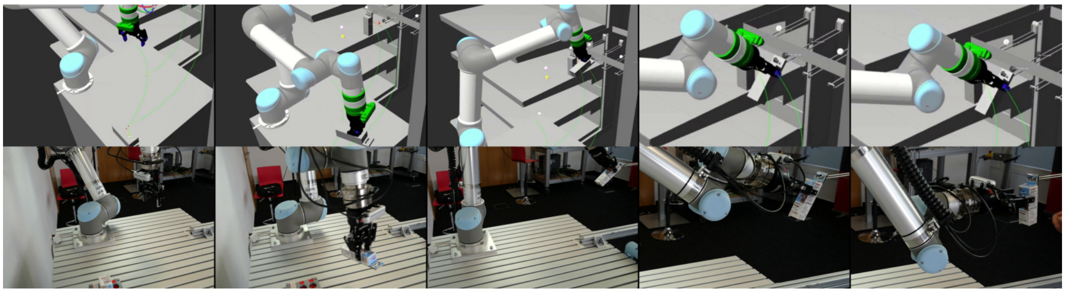


Figure 4.11: Simulated (top) and real-world (bottom) execution of a generated robot source program for force-controlled peg-in-hole in a lab environment (Alt et al., 2023).

that cannot, in fact, be executed in the environment at hand. On the other hand, a very fine-grained task definition may only afford task abstraction for very specific environments or demonstrated actions, requiring the addition of new rules to the knowledge base whenever the environment changes or actions are demonstrated in a slightly different way. Architectures such as MetaWizard use a “generate and test” principle to resolve this trade-off and accept some inefficiency (here, rejecting some of the generated candidate task sequences during grounding or simulation) to avoid overspecification.

The experiments also evidenced that results can be further improved by the integration of sensory modalities for grounding. In the first application scenario, for example, vision-based detection of the object, rather than grounding its pose from the knowledge base, would likely have improved task success. Likewise, in the second application scenario, detecting the tip of the peg via vision would have avoided execution failures. Qualitative validation in a supermarket environment (see Section 4.1.7.2) corroborates these findings. Likewise, the results can be further improved by the integration of learning into the framework. MetaWizard grounds candidate plans in predefined force-controlled skills that are general-purpose, and therefore cover a wide range of use cases, but at the same time are not adapted to the concrete task at hand. In the context of the peg-in-hole scenario, for example, learning an insertion strategy for the hook-shaped peg used in the supermarket environment would likely have led to even more robust results, and can be implemented on top of MetaWizard without requiring changes to the program synthesis architecture itself.

4.1.7.2 Qualitative Validation

Experiment 4.1.7.1 evaluated MetaWizard under laboratory conditions. To validate the framework in a realistic environment, the experiments are replicated in a realistic supermarket environment at the retail robotics lab of the IAI at the University of Bremen (Alt et al., 2023).

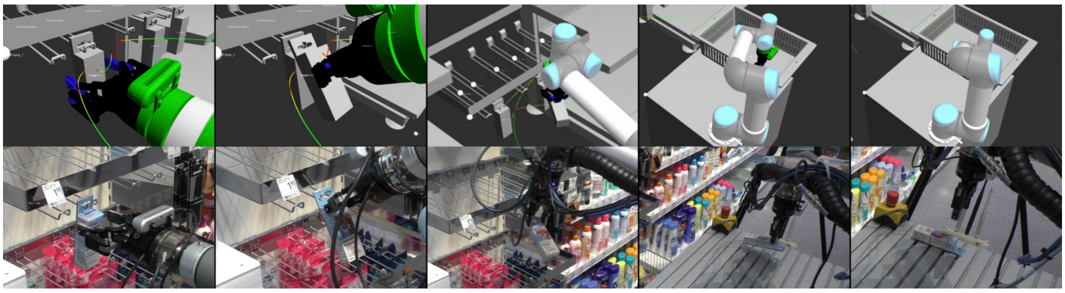


Figure 4.12: Simulated (top) and real-world (bottom) execution of a synthesized robot program for a supermarket fetch-and-place task in a realistic environment (Alt et al., 2023).

Experiment setup The semantic map of the supermarket environment is shown in Figure 4.3 (right). A UR5 collaborative manipulator[9] with an flange-mounted force-torque sensor,[10] a Robotiq 2F-85 parallel gripper[11] and a RealSense D435 RGB-D camera[12] is mounted on a moving base. For this experiment, the position of the base is fixed. The experiment was conducted in a laboratory for retail robotics, which featured a supermarket environment nearly identical in layout, lighting and furniture to a small branch of a large German drugstore chain,[13] comprising a selection of the original product range. For each of the two use cases, one human demonstration was chosen at random and program synthesis was performed. The RoboSherlock perception framework (Beetz et al., 2015; Kenghagho Kenfack et al., 2020) is used to ground object designators in real-word 3D camera images, notably to identify and locate the box of healing salve in the scene. As in Experiment 4.1.7.1, object-relative location designators, such as grasp points, are grounded in precomputed features via the knowledge base.

Results For both application scenarios, program synthesis resulted in robot source programs that could successfully execute the task. Figures 4.12 and 4.13 show the simulation and real-world execution for each scenario, respectively. For each application scenario, program synthesis was performed twice, each time with a different initial object pose. RoboSherlock accurately detected the pose of the object and could be queried via the KnowRob engine without requiring changes to MetaWizard. Due to the crowded collision environment and small inaccuracies in the perception pipeline, notably with respect to the orientation of the object, task grounding required several steps of backtracking until a collision-free candidate program was found.

---

[9] Universal Robots A/S, Odense, Denmark

[10] ATI Industrial Automation Inc., Apex, USA

[11] Robotiq Inc., Lévis, Canada

[12] Intel Corporation, Santa Clara, USA

[13] dm-drogerie markt GmbH + Co. KG, Karlsruhe, Germany

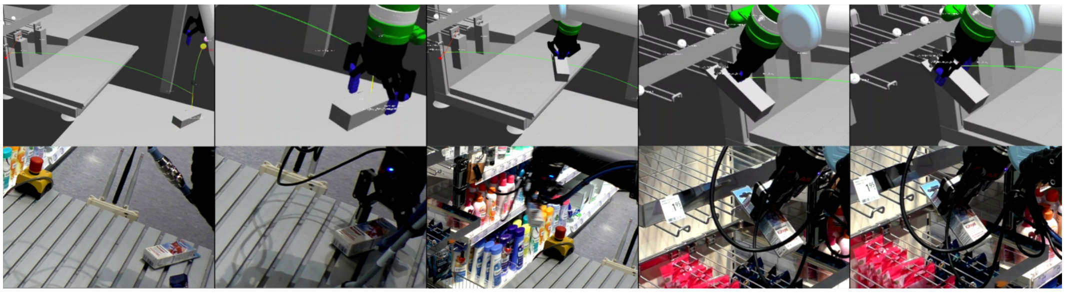


Figure 4.13: Simulated (top) and real-world (bottom) execution of a synthesized robot program for a force-controlled peg-in-hole task in a realistic environment (Alt et al., 2023).

Discussion Due to very limited time in the supermarket laboratory, only two trials per scenario could be performed. While not statistically significant, the results conceptually validate the integration of perceptual grounding via RoboSherlock into MetaWizard: In all four trials, the pose of the target object deviated from its default pose in the semantic map, and the object could be grasped and manipulated without manual changes to the knowledge base. Notably, RoboSherlock was integrated as a drop-in reasoner backend in KnowRob, using the same knowledge representations and data structures as MetaWizard and requiring no changes to the task abstraction or task grounding mechanisms. Extending the use of perceptual grounding to ground all location designators, such as object features, can further improve MetaWizard's generalization abilities.

4.1.8 Related Work

MetaWizard addresses the challenge of synthesizing executable robot code from intuitive inputs such as VR human demonstrations. It shifts the user-facing aspect of programming from specifying *how* the robot is to perform a task to specifying *what* the robot is to do, inferring the *how* from a knowledge base containing task-specific, domain-specific and commonsense knowledge. There is a rich literature on robot program synthesis and its constituting subproblems such as task recognition – the semantic understanding of e.g. demonstrated actions – and task execution, the conversion of abstract task structures into real-world robot actions.

4.1.8.1 Knowledge-Based Robot Program Synthesis

Program synthesis requires an understanding of the user's intent – what the robot ought to do – given some user input that does not directly, or not exhaustively, specify that intent. MetaWizard solves the problem of task recognition by abstraction from concrete, demonstrated events (human actions in VR) to tasks with known, well-defined semantics (Alt et al., 2023). To that end, it uses symbolic representations of tasks and events, which are defined in terms of SOMA (Beßler et al., 2021) and derived ontologies, as well as the KnowRob KR&R engine for semisymbolic reasoning over a knowledge base containing symbolic knowledge and subsymbolic data, such as robot trajectories (Beetz et al., 2018). In the KnowRob ecosystem, several related approaches propose leveraging explicit, structured task, domain and commonsense knowledge to understand the intent of human actions and synthesize corresponding robot programs. Haidu and Beetz (2016) present a pipeline for action recognition from VR human demonstrations, which inserts VR data into the semisymbolic knowledge base and uses Prolog reasoning routines to semantically classify cooking tasks. MetaWizard proposes a more structured task representation, in which all tasks are defined in terms of their pre-, post- and runtime conditions and which yields all candidate task sequences consistent with the VR demonstration. Haidu and Beetz (2021) introduce the VR demonstration and action recognition pipeline used by MetaWizard, in which elementary actions are classified based on force-dynamic events.

Ramirez-Amaro et al. (2017) propose a program synthesis system within the KnowRob ecosystem that is the technological and intellectual predecessor of MetaWizard. It uses the KnowRob semisymbolic KR&R engine to infer the goals pursued by humans as they demonstrate kitchen tasks, and then plans a sequence of robot actions to achieve those goals. Their action recognition pipeline semantically segments the demonstration timeline by detecting whether or not the human hand is moving, whether a tool is used, whether an object is acted on and whether an object is in the human's hand. The segmented timeline is then translated into individuals the semisymbolic knowledge base. Bates et al. (2017) extend this work to support VR human demonstrations. For execution, goals are directly mapped to robot tasks, which in turn consist of sequences of primitive robot skills. MetaWizard is based on the same foundational principle of task abstraction followed by task execution. MetaWizard uses lower-level force-dynamic events to segment the demonstration timeline at finer granularity, and models tasks in terms of pre-, runtime- and postconditions, permitting higher generality. Moreover, its task model and execution pipeline supports complex manipulation tasks such as force-controlled insertion or retraction.

Patton et al. (2024b) propose PROLEX, an approach to robot program synthesis that is structurally similar to MetaWizard in that it first generates a set of program "sketches" – underspecified plans –, and then "completes" – grounds – the sketch in the human demonstration. PROLEX generates a sketch by converting demonstrations into strings of characters and generating a regular expression that matches them. It then produces executable programs by finding character strings that match the synthesized regular expression, and leverages an LLM as a heuristic to prioritize programs that are more likely to be executed successfully. Like MetaWizard, PROLEX generates multiple tasks sequence candidates for a given demonstration, and eliminates infeasible plans during task grounding. One central difference to MetaWizard is that PROLEX grounds plans by unifying them over multiple demonstrations, i.e. by finding an fully specified sequence of actions that matches the regular expression synthesized from the demonstrations. MetaWizard instead grounds candidate plans in the current environment at hand, allowing for generalization to an environment different from that during demonstration while still generating specific plans for that new environment. Another difference is that PROLEX generates task sequences that *syntactically* match the demonstrations, while MetaWizard performs task abstraction using a semantic task definition.

4.1.8.2 Neural Robot Program Synthesis

Recent approaches to robot program synthesis have leveraged deep neural representations, particularly LLMs and Transformer-based multimodal foundation models, to synthesize robot programs. Zitkovich et al. (2023) propose Robotics Transformer 2 (RT-2), an end-to-end neural approach to robot program synthesis. RT-2 translates a textual description of a task and an image of the current environment into a unified token representation, which is transformed by an LLM into low-level robot control commands, such as end-effector pose or joint angle increments. RT-2 relies on implicit representations of actions and the environment. Task-specific and commonsense knowledge is represented in latent space, implicitly encoded in the weights of the Vision Transformer (Dosovitskiy et al., 2020) used to encode the environment image into text tokens, and the LLM used to generate the output action sequence, both of which were pretrained on web-scale, general-purpose datasets. Several related end-to-end neural program synthesizers have been proposed. CaP (Liang et al., 2023) uses an LLM to synthesize Python robot control code from textual natural-language descriptions of a task, by composing and parameterizing high-level functions from a low-level robot control API. Unlike RT-2, which generates low-level motion commands, CaP generates textual robot programs that can be read and modified by human programmers. SayPlan (Rana et al., 2023) use an LLM to perform semantic search in a 3D scene graph to generate high-level plans from natural-language task descriptions, and then use the same LLM to iteratively refine the plan until it is executable in a simulator. It uses a similar generate-and-test pattern to MetaWizard, generating candidate plans from the human instruction which are then refined to fit the target environment. With "ChatGPT for robotics", Vemprala et al. (2023) propose to use ChatGPT (OpenAI, 2023), an off-the-shelf general-purpose chat LLM, to synthesize executable robot code purely via prompt engineering (Marvin et al., 2024), i.e. structuring the textual input to the LLM in a way that encodes the user's intention, information about the environment as well as any additional knowledge that may be required for the task. Similar to SayPlan, they propose an iterative refinement procedure, by which the LLM repeatedly iterates over previously generated robot code to successively improve robot behavior. ProgPrompt (Singh et al., 2023), in a manner similar to CaP, proposes a prompt structure designed to provide the code-generating LLM with the required knowledge about available actions, the objects in the environment. They include example programs in the prompt, which provide the LLM with the implicit, contextual knowledge about the semantics of the available control primitives. Recent architectures such as Octo (Ghosh et al., 2024), GROOT (NVIDIA, 2024b) or RFM-1 (Covariant, 2024b) build on large-scale pretrained VLA models to generate outputs in different modalities, such as joint angles, textual robot code or natural-language descriptions, given inputs in the same or different modalities. Cross-modality reasoning is enabled by pretraining on several web-scale, partially multimodal datasets, as well as modular network architectures that combine pre-trained, modality-specific encoders and decoders with a shared, jointly trained backbone (Ma et al., 2024).

End-to-end neural program synthesizers like RT-2 do not require humans to provide the system with explicit knowledge by virtue of learning, and generalize well by leveraging highly general, implicit representations learned over a wide variety of training data. Unlike end-to-end neural systems, MetaWizard uses a modular architecture with distinct reasoning steps (task abstraction and task grounding), and relies on explicit knowledge representations and reasoning routines. Modular program synthesis systems with explicit knowledge representations and reasoning mechanisms make it comparatively easy for humans to fundamentally alter or reliably influence the behavior of the system without retraining it on new data, by adding new knowledge to the knowledge base. Moreover, end-to-end neural systems can be prone to "hallucinations", implausible generation results indicating a lack of deep semantic understanding (Huang et al., 2023b; Gunjal et al., 2024). Moreover, explicit systems such as MetaWizard afford mechanistic interpretability, allowing the user to understand the chain of reasoning steps by which the system arrived at a particular conclusion (Bereska and Gavves, 2024; Burkart and Huber, 2021).

4.1.9 Discussion

MetaWizard performs robot program synthesis by reasoning over a semisymbolic knowledge base. It provides an intuitive, automatic way to synthesize robot source programs from human VR demonstrations. From a programmer's perspective, MetaWizard alleviates the need for task- or domain- specific expertise or programming experience, as this knowledge is encoded in the knowledge base prior to program synthesis.

A general challenge faced by program synthesis systems is to find knowledge representations and reasoning algorithms that are sufficiently general to cover a wide range of applications, while at the same time permitting inference of program structures and parameterizations that are specific enough to solve concrete use cases. MetaWizard resolves this trade-off by combining a highly generalizable, ontological model of knowledge with hybrid data structures such as semantic maps and the TF database, which permit the grounding of abstract tasks in concrete, numerical data about the current environment. Neural approaches face the same tradeoff: They must be trained on sufficiently diverse datasets to permit generalization across use cases, yet must generate robot programs that solve a precise, exact task in a highly performant way. They address it by domain-specific finetuning, few-shot task learning, or reinforcement learning from human feedback (Ouyang et al., 2024). MetaWizard does not require upfront training on large datasets or finetuning on task-specific data, making it particularly suited for applications in which acquiring training data is prohibitively expensive. In Section 4.2, MetaWizard is applied in the context of such an application. Instead, it shifts the burden of knowledge specification to the metaprogrammer, who must encode prior knowledge in a structured knowledge base.

MetaWizard's complete program synthesis pipeline is explainable, and the reasoning behind given decisions – such as what events in the human demonstrations led to a particular task being included in a candidate task sequence – can be made transparent by following the chain of unification and backtracking operations that underlie task abstraction and task grounding. From a metaprogramming perspective, this is particularly useful to explain erroneous synthesis results, and add missing knowledge or fix contradictory assertions. From a user's perspective, it fosters trust in the behavior of the system. The conception of explanatory mechanisms and corresponding user interfaces for complex KR&R systems like MetaWizard is a promising avenue of future work.

The data structures used to represent human demonstrations merit additional discussion. The use of NEEMs as an input modality allows for program synthesis from human demonstrations, but can also represent robot experience data (Beetz et al., 2023). NEEMs, by virtue of axiomatizing actions and motions in terms of ontologies, permit downstream systems to abstract away from the particulars of the executing agent or the execution environment. NEEMs as an input modality enable MetaWizard to synthesize robot programs given demonstrations from other robots for cross-platform learning. The use of NEEMs, as well as the representation of task knowledge in terms of the KnowRob semisymbolic knowledge base and the use of CRAM plans as an intermediate representation for task grounding integrate MetaWizard in the CRAM cognitive architecture as a component for program synthesis and metaprogramming.

Many program synthesis systems, including MetaWizard, map demonstrated action sequences to tasks at a higher level of abstraction. In MetaWizard, task abstraction assumes that each action executes one task. This restricts the knowledge representation to model tasks and actions at the same level of granularity, and may require an unnecessarily fine-grained task model. A more intuitive model of tasks and actions would allow e.g. the possibility of a sequence of actions executing some common task. Likewise, task grounding assumes that for each task, there is a corresponding skill or predefined skill sequence. To achieve good generalization, this requires considerable modeling upfront, as a wide variety of tasks must be considered. The integration of planners that generate skill sequences to achieve a given task, can decrease the overhead of modeling, as only the primitive skills must be modeled. Recent works in the TAMP community propose to use PDDL planners to sequence primitive TAMP operators on the fly (Vu et al., 2024), avoiding the need to explicitly model skill sequences beforehand, or to learn primitive operators which are then sequenced by a planner (Silver et al., 2021). Integration of task planners or integrated task and motion planning into MetaWizard could further reduce the burden of modeling and increase generalization while keeping the knowledge base compact.

MetaWizard, in its current iteration, does not incorporate learning. Knowledge is added to the knowledge base upfront, via the semantic map of the environment and the task definitions, and during program synthesis by way of the human demonstration, which asserts new facts into the knowledge base. New knowledge is derived from these assertions at inference time, but that knowledge is not retained for future use. Learning from past program synthesis results would enable MetaWizard to yield more likely candidate plans first, cover situations the human designers of the knowledge base did not specify, or yield executable robot programs that solve tasks with increasing reliability or efficiency. NEEMs are expressly designed to support learning from humans, from other robots, but also from the system's own experience (Beetz et al., 2023). Learning mechanisms can be integrated into MetaWizard in several ways. Mechanisms that update the knowledge base in a persistent way, such as automatic derivation of new ABox and TBox axioms from experience or demonstration (Petrucci et al., 2016; Khadir et al., 2021), can be integrated in a drop-in manner, without requiring changes to the remainder of the system. Likewise, neural learning approaches, such as use of CNNs, Vision Transformers (Dosovitskiy et al., 2020) or Neural Scene Graphs (Ost et al., 2021) can be integrated as grounding mechanisms into the task grounding scheme in a manner similar to RoboSherlock, to improve task grounding as the system gains experience.

All program synthesis frameworks face the challenge that the real-world environment may significantly deviate from the model assumed during program synthesis. This is particularly true for dynamic environments, which may change during runtime, or programs that involve interaction with other agents, whose behavior cannot be fully modeled. CRAM addresses this challenge by resolving CPL plans online, grounding each action just before executing it (Beetz et al., 2023). This principle of just-in-time grounding can be implemented by MetaWizard without significant architectural change, but implies eschewing the generation of complete, fully specified programs, which can be reviewed and possibly certified by human experts before execution. For application domains in which dynamic environments are the rule, such as household assistance tasks, just-in-time grounding may be preferable, while for industrial robotics applications, fully offline grounding is preferred. Even with just-in-time grounding, execution failures are to be expected. Failure handling has been supported by the CPL since its inception (Beetz et al., 2010), and has to date been realized by predetermined failure handling routines. Meywerk et al. (2022) propose a mechanism for the automatic generation of tests to ensure that all known failure modalities are handled. Extending MetaWizard to automatically synthesize subprogram for failure handling is a promising and important line of future work.

4.2 An Interactive Robot Programming Assistant

The vision of the "robot colleague" – a robot assistant that can be taught and instructed like a human coworker – has been the subject of scientific study and the aim of many research and engineering efforts since the inception of the field of robotics (Nyholm, 2024). The ability to *communicate* with robots in a bilateral conversation, by means of natural language, has been emphasized as a crucial component of this vision, and repeatedly found to be beneficial for the acceptance of robots by human coworkers (Dautenhahn, 2007; Oistad et al., 2016; Strazdas et al., 2020). The vision of conversational human-robot interaction naturally extends to robot programming: Programs are written, read, executed, simulated and refined in constant back-and-forth interaction between programmer and machine. AI-enabled programming systems like MetaWizard promise to store expert knowledge and make it available to non-experts through automatic program synthesis. Interactive robot programming, realized by AI algorithms, can enable users to access this knowledge in an intuitive manner.

As a system for automatic program synthesis from human demonstrations, MetaWizard as introduced in Section 4.1 shifts the burden of programming away from creating robot programs and towards the upfront specification of task and domain knowledge in the knowledge base, as well as the recording of good human demonstrations of the task. For some applications, however, collecting human demonstrations may be highly challenging or even infeasible: Destructive processes such as sanding or cutting, for example, are challenging to simulate in VR, and real-world demonstrations may be prohibitively expensive, particularly if workpieces are unique. For these applications, an interactive programming paradigm may be better suited. This section introduces MetaWizard2, an interactive programming assistant based on MetaWizard that guides users through the robot programming process by interactive dialogue. The work described in this chapter was first presented by Alt et al. (2024c).

4.2.1 Overview

Figure 4.14 outlines an interactive assistant for robot programming. The user instructs the assistant with a rough description of the task objective – in Figure 4.14, the task consists of sanding a wind turbine rotor blade. Through natural-language dialogue with the user, MetaWizard2 incrementally grounds its knowledge base, drawing from user-supplied information, commonsense knowledge, as well as planning and perception modules. The synthesized robot program is executed on the robot and the user is asked for feedback. The process is repeated until the task is complete.

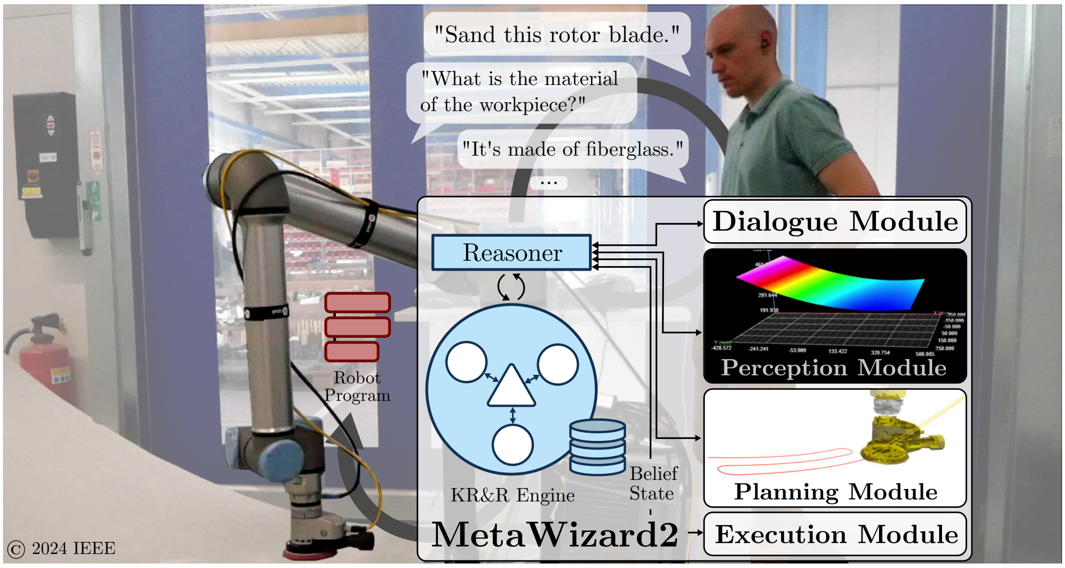


Figure 4.14: MetaWizard2 co-creates executable robot programs by interactive dialogue with a human programmer (Alt et al., 2024c).

The central algorithmic contribution of MetaWizard2 is a flexible, extensible metaprogramming system for robot tasks. High-level tasks are modeled as underspecified workflows, which include not just robot (sub-)tasks resulting in physical robot actions, but also interactive tasks such as obtaining feedback from a human expert. The metatask hierarchy is grounded *at runtime* by querying sensor data or asking questions to the user. MetaWizard2 is designed around three guiding principles:

1. **Metaprogramming:** Task hierarchies are modeled as knowledge in a KB, and executable NRP source programs are synthesized by reasoning over this KB. Generalization to novel tasks should require only the addition of new knowledge to the KB.

2. **Proactive grounding through dialogue:** The program synthesis system should be “aware” of knowledge that may be missing to ground a task, and proactively ask questions to the human user to fill such gaps in the KB as they are discovered.

3. **Just-in-time task grounding:** Subtasks should be grounded just before they are executed. Task grounding and execution should be interwoven to react e.g. to unexpected changes in the environment.

Where MetaWizard emphasizes offline, fully autonomous planning of robot source programs with distinct task abstraction, task grounding and program execution phases, MetaWizard2 emphasizes interactive grounding and execution, keeping the human user in the loop as a source of knowledge and expertise.

4.2.2 Metatask Representation

MetaWizard2 uses the KR&R backend of MetaWizard, based on the KnowRob semisymbolic knowledge base and reasoning engine (Beetz et al., 2018). To realize dialogue-based programming, MetaWizard2 proposes to model tasks not as atomic units, but rather as constituents of an overarching *metatask*. The metatask models not just subtasks to be executed by the robot, such as grasping an object or sanding a surface, but also cognitive or communicative tasks required to be executed by MetaWizard2 itself, such as performing path planning or asking a human user for input. It acts not only as a model of some real-world task to be achieved, but also as a model of MetaWizard2's own process of achieving the task. Metaprogramming, i.e. specifying or changing the behavior of MetaWizard2 itself, is reduced to adding metatask information to the KB. This permits MetaWizard2 to generalize between profoundly different task types without requiring changes to the codebase of MetaWizard2.

Metatask ▷

The metatask model is defined in `metawizard2.owl` (short `mw2`), a domain ontology that extends SOMA (Beßler et al., 2021). A metatask is represented as an individual of class `mw2:MetaTask` of type `dul:Task` in KnowRob's semisymbolic knowledge base. As a running example, a refabrication task is considered, in which the rotor blade of a wind turbine is sanded down by a robot after repairs to the surface. The use case of robot-based refurbishing of wind turbine blades is considered in greater detail in Experiment 4.2.4.1. An application ontology for surface treatment tasks, (`surface_treatment.owl`, short `sft`) extends `mw2` and defines the metatask type `sft:Sanding`. A metatask has a set of *metatask parameters*, *workpieces*, *tools* and *subtasks*.

Metatask parameters The metatask parameters comprise task-specific, symbolic or subsymbolic information associated with the metatask. `sft:Sanding`, for example, has the metatask parameters `sft:CuttingDepth` representing the amount of overall material, in mm, to be removed, as well as a symbolic parameter `sft:Finish` with values `rough` or `fine`. Metatask parameters are related to the metatask via the object property `dul:hasParameter`, and are available to the reasoner during task grounding at all levels of the task hierarchy.

Workpieces The workpieces of a metatask are all objects manipulated or otherwise interacted with by the robot during execution of the metatask. For `sft:Sanding`, the object to be sanded is the only workpiece. Workpieces are related to the metatask via the object property `mw:hasWorkpiece`.

Tools The tools of a metatask are all tools used by the robot during the execution of the metatask. For `sft:Sanding`, the sander is the only tool. Tools are related to the metatask via the object property `mw:hasTool`.

Subtasks The subtasks of a metatask are the lower-level tasks, such as path planning, simulation, program execution or user interaction, that are required to execute the metatask. Subtasks are sequenced by the predicate `successor(T, S)` in the KB, indicating that task type `T` is succeeded by task type `S`. Succeedence relations are asserted into the KB as TBox axioms and apply to all instantiated subtasks of the given types. Figure 4.15 illustrates the definition of a metatask's subtasks in MetaWizard2 by the example of `sft:Sanding`. The left-hand side shows a "one-shot" version of a sanding task that performs one pass over the surface. The right-hand side shows sanding as an iterative process, which asks the user after every pass whether the surface has the desired quality, whether parameters (here, the intended surface finish) should be adapted, and repeats the process until the user is satisfied. Listing 4.10 shows the programmatic definition of the metatask in Prolog. Adaptation of a sequential to an iterative metatask is achieved purely by adding additional assertions to the KB (highlighted in blue in Figure 4.15 and Listing 4.10). Each subtask has an associated `workflow`, following the SOMA workflow model, that specifies the concrete steps MetaWizard2 must perform to execute the subtask. In Figure 4.15, the subtask `sft:ChooseFinish` has one associated workflow step, `sft:AskUser`. Associating a workflow with each subtask adds a layer of flexibility in the model that make the metatask, to a degree, independent from the concrete cognitive, computational or physical tasks required to execute it: In some contexts, `sft:ChooseFinish` may also be realized by reasoning over the KB instead of direct user interaction.

4.2.3 Metatask Grounding

Like abstract candidate task sequences generated by MetaWizard (see Section 4.1.4), the metatask is *underspecified*. While for most tasks, some TBox information such as the required tool type may be available, ABox information such as the concrete tool to be used for the task, the workpiece to operate on or subtask parameters such as planned motions must be grounded in the real-world environment through perception, user interaction and planning. Metatask grounding is done via semisymbolic reasoning over the knowledge base, using additional perception, planning and dialogue modules to provide ground-truth information at runtime. The dialogue and execution modules are described below. Application-specific perception and planning modules are described in the context of Experiment 4.2.4.1.

4.2.3.1 Dialogue module

The dialogue module provides a mechanism for interactive symbol grounding through natural-language user interaction. It consists of a natural language processing (NLP) pipeline for text-to-speech and speech-to-text conversion, abstracted behind an interface providing high-level methods for grounding numerical or symbolic parameters, choosing values among a set of categories, etc. The dialogue module is integrated into MetaWizard2's KR&R engine via a Python API.

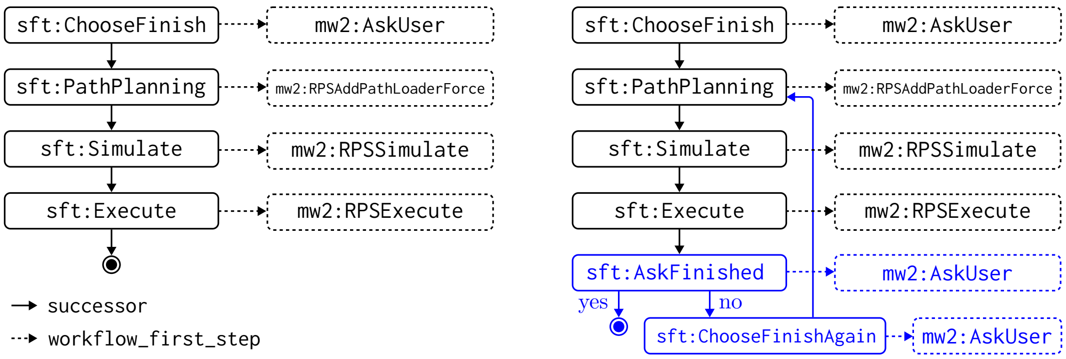


Figure 4.15: The MetaWizard2 metatask representation, at the example of a sanding task. Subtasks are chained by a succeedence relation and are each associated with a workflow, detailing concrete steps required to execute them. One-shot sanding (left) can be extended to an iterative process (right) by asserting additional succeedence relations (blue).

```
successor(sft:'ChooseFinish', Succ) :-
    true -> atom_string(Succ, sft:'PathPlanning').
successor(sft:'PathPlanning', Succ) :-
    true -> atom_string(Succ, sft:'Simulate').
successor(sft:'Simulate', Succ) :-
    true -> atom_string(Succ, sft:'Execute').
successor(sft:'Execute', Succ) :-
    true -> atom_string(Succ, sft:'AskFinished').
successor(sft:'AskFinished', Succ) :-
    kb_call(parameter_value(done_flag, X)),
    X = not_done,
    atom_string(Succ, sft:'ChooseFinishAgain').
successor(sft:'ChooseFinishAgain', Succ) :-
    true -> atom_string(Succ, sft:'PathPlanning').
successor(sft:'Execute', Succ) :-
    true -> atom_string(Succ, sft:'AskFinished').
successor(sft:'AskFinished', Succ) :-
    kb_call(parameter_value(done_flag, X)),
    X = not_done,
    atom_string(Succ, sft:'ChooseFinishAgain').
successor(sft:'ChooseFinishAgain', Succ) :-
    true -> atom_string(Succ, sft:'PathPlanning').
```

Listing 4.10: Metatask definition of a sanding task via Prolog predicates in the semisymbolic knowledge base. One-shot sanding can be extended to an iterative process by asserting additional succeedence relations (blue).

Verbalization of user queries Queries by the KR&R engine are three-tuples (S, T, P) consisting of a query string S (the *prompt*) provided by the KR&R engine, the query type T and query parameters P. Two query types are supported, *option* and *parameter*. *Option* queries ask the user for a choice between a set of given options, while parameter queries ask the user for a numerical value with an associated unit. For an *option* query, P is the list of (string) options. For a *parameter* query, P is a tuple containing the parameter type (`float`, `int` etc.) and the unit (mm, degrees etc.). To ground the metatask parameter `sft:CuttingDepth` of `sft:Sanding`, for example, MetaWizard2 queries the dialogue module with the tuple ("How much material should be removed?", `float`, mm). Queries are output to the user via standard text-to-speech utilities (`pyttsx3`, Bhat (2020)).

Parsing of natural-language user responses To parse natural-language user responses, a hybrid speech recognition pipeline is proposed (Alt et al., 2024c):

1. **Speech-to-text conversion:** User speech is converted to text via the Google Speech Recognition API (Google, 2024).
2. **Tokenization and embedding:** The `spaCy` library (Honnibal and Montani, 2024) is used to chunk the text into semantic units. The semantic chunks are embedded into a vector space.
3. **Semantic matching:** Based on their semantic roles and vector embeddings, chunks are matched to symbols to be grounded. To ground `sft:CuttingDepth`, for example, the chunk "point seven" is recognized as a quantity, and "inches" is recognized as a unit. The parsed parameter value, converted to the desired unit, is returned to MetaWizard2. For *option* queries, the option with the smallest distance to its nearest-neighbour chunk in embedding space is returned.

MetaWizard2 then asserts the grounded parameter values or options into the knowledge base.

With the dialogue module, MetaWizard2 has a general-purpose backend to ground variables that it cannot ground in its knowledge base. Whenever MetaWizard2 fails to unify the arguments of a predicate, it can query the dialogue module to supply user-provided values for the uninstantiated arguments and assert them into the KB so that subsequent unification attempts will succeed.

4.2.3.2 Execution Module

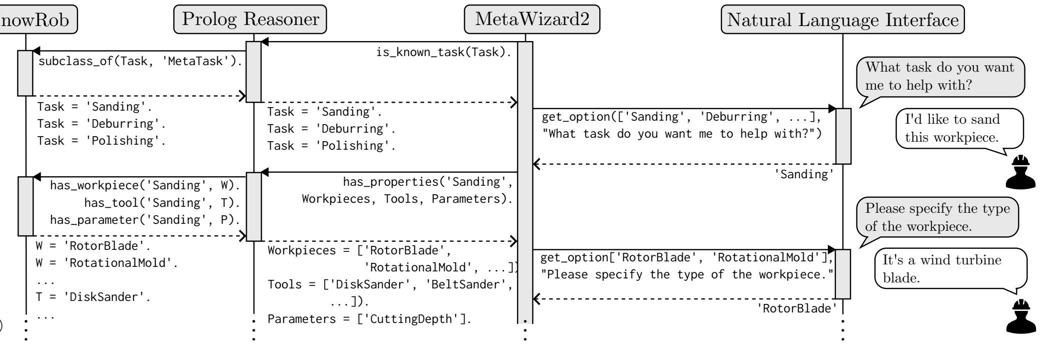
The execution module provides capabilities for executing and/or simulating grounded robot source programs. The default execution module of MetaWizard2 is a Python wrapper of the ArtiMinds RPS industrial robot IDE (Schmidt-Rohr et al., 2013). The RPS provides collision-free path planning and can also serve as an elementary planning module, allowing MetaWizard2 to plan e.g. collision-free approach or depart motions. Complex, application-specific planners can be integrated via dedicated planning modules, such as the planner for surface treatment tasks introduced in the context of Experiment 4.2.4.1.

4.2.3.3 Metatask Grounding

Given the metatask definition and the dialogue module, Metatask grounding requires to first ground the metatask parameters, workpieces and tools, before incrementally grounding and executing the subtask sequence.

Grounding of parameters, workpieces and tools When MetaWizard2 is first started, the metatask at hand is unknown and represented by an anonymous individual in the knowledge base. Queries to ground the metatask type will fail (as no type for the individual has been asserted), so the user is asked to specify the metatask type. Note that queries and user responses are articulated in natural language (e.g. "What task can I help with?", and "I want you to sand this injection mold."). The parsed reply (e.g. `sft:Sanding`) is asserted into the KB, and subsequent KB queries about the metatask type and associated background knowledge will succeed. This paradigm of creating anonymous individuals and "filling out" missing information about them by asking the user for information, if it cannot be inferred from the KB, is MetaWizard2's default mode of knowledge discovery. Unification of the predicate `has_parameter(TT, P)` yields all parameters `P` for given metatask type `TT`, which are each grounded, in turn, by unification with available domain knowledge in the KB or user interaction, should no such knowledge be available. The workpieces and tools of the metatask are grounded in the same manner. Interactive grounding of objects, such as workpieces and tools, is realized by *option* queries, which provide the user with a choice between (known) objects in the environment that correspond to criteria, such as class constraints, defined in TBox assertions as part of the metatask definition (e.g. `sft:Sanding` requires a tool of type `sft:Sander`). Objects in the KB are associated with the corresponding 3D meshes, which are required for e.g. collision-free motion planning during subtask grounding. Once all workpieces and tools are grounded, the kinematic robot model and object meshes are instantiated in the planning environment, here the ArtiMinds RPS. Parameter, tool and workpiece properties are available during grounding of subsequent task types. Figure 4.16 illustrates the interactive grounding of metatask parameters, workpieces and tools.

Subtask grounding and execution The subtask sequence is grounded by successively querying the `successor` predicate on each subtask of the metatask. For each subtask, the associated workflow is traversed and each workflow step is executed in turn. Each workflow step corresponds to a predefined handler routine, which executes the workflow step using the planning, perception execution or dialogue modules, the general-purpose KB, or other external utilities, depending on the


KnowRob
Prolog Reasoner
MetaWizard2
Natural Language Interface
is_known_task(Task).
subclass_of(Task, 'MetaTask').
Task = 'Sanding'.
Task = 'Deburring'.
Task = 'Polishing'.
Task = 'Sanding'.
Task = 'Deburring'.
Task = 'Polishing'.
get_option(['Sanding', 'Deburring', ...],
"What task do you want me to help with?")
What task do you want me to help with?
I'd like to sand this workpiece.
'Sanding'
has_properties('Sanding', Workpieces, Tools, Parameters).
has_workpiece('Sanding', W).
has_tool('Sanding', T).
has_parameter('Sanding', P).
W = 'RotorBlade'.
W = 'RotationalMold'.
...
T = 'DiskSander'.
...
Workpieces = ['RotorBlade', 'RotationalMold', ...]
Tools = ['DiskSander', 'BeltSander', ...]).
Parameters = ['CuttingDepth'].
Please specify the type of the workpiece.
get_option['RotorBlade', 'RotationalMold'],
"Please specify the type of the workpiece."
It's a wind turbine blade.
'RotorBlade'

© 2024 IEEE

Figure 4.16: MetaWizard2 grounds symbols in its knowledge base by natural-language interaction with a human programmer (Alt et al., 2024c).

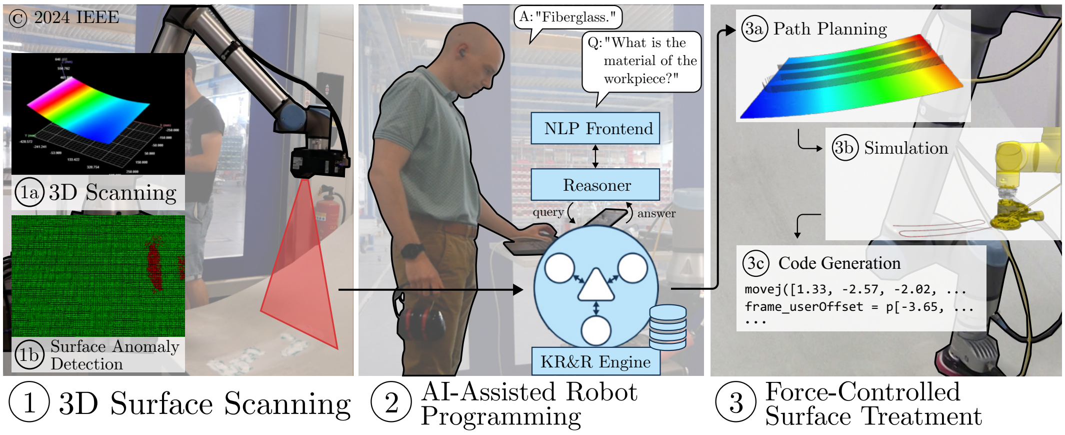
workflow step. The subtask `sft:ChooseFinish`, for example, has the only workflow step `mw2:AskUser`, which uses the dialogue module to ask the user for the desired surface quality. Experiment 4.2.4.1 details the grounding and execution of the more complex subtask `sft:PathPlanning` in the context of sanding, which requires domain-specific perception and planning modules.

4.2.4 Experiments

MetaWizard2 leverages a semisymbolic knowledge base and a flexible metatask representation to store domain-specific task knowledge and makes that knowledge accessible to human users for program synthesis via natural-language interaction. Robotic surface treatment tasks such as sanding, grinding or deburring require a particular degree of domain-specific expertise, as they are contact-rich, dynamic processes that are hard to simulate and where e.g. quality criteria are often highly context-dependent and difficult to quantify. In many applications such as polishing, the task objective is for a surface to “look good”, which is a subjective criterion judged by domain experts. As MetaWizard2 centers around human-robot co-programming, it holds particular promise in application domains such as robotic surface treatment, where iterative workflows with continuous human involvement predominate. Experiment 4.2.4.1 applies MetaWizard2 to wind turbine refabrication. Section 4.2.4.2 discusses the generalization of MetaWizard2 to related sanding and deburring tasks.

4.2.4.1 Refabrication of Wind Turbine Blades

To validate MetaWizard2 on a real-world industrial scenario, the refabrication of wind turbine blades is considered. The rotor blades on wind turbines are subject to constant wear and tear from wind, precipitation, lightning and other factors. As wind turbines age, their power output decreases by 1.8% per year (Staffell and Green, 2014). The design lifespan of wind turbine blades is 20 years, with many blades being replaced before that time (Liu and Barlow, 2017). Wind turbine



```
movej([1.33, -2.57, -2.02, ...
frame_userOffset = p[-3.65, ...
...
```

Figure 4.17: MetaWizard2 was used to synthesize executable robot programs (3) given vision sensor input (1) and natural-language user interaction (2) (Alt et al., 2024c).

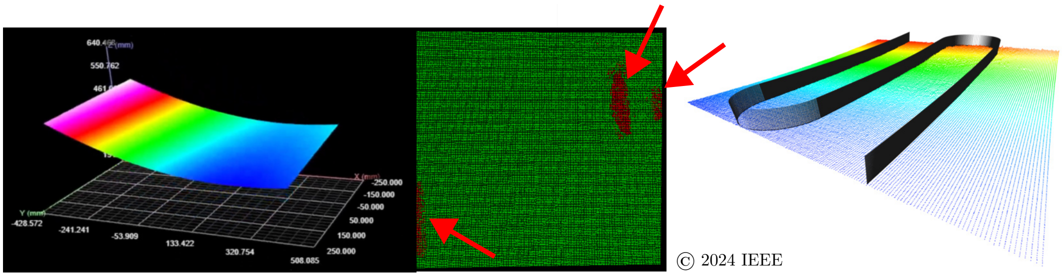
blades are made from composite materials, with an outer shell of coated fiberglass. At the time of writing, no industrial-scale method for recycling wind turbine blades exists (Lund and Madsen, 2024), making a strong case for automated robotic remanufacturing. Remanufacturing of wind turbine blades comprises filling holes, dents or uneven regions of the blade surface with fiberglass putty, and subsequently sanding the surface until a uniform curvature and smooth surface is achieved. Like for most surface treatment tasks, no quantifiable metric for task success has been defined. Rather, human experts must look at and touch the surface to determine whether it is "smooth enough". This is amplified by the fact that there is little literature on robotic wind turbine remanufacturing (Franko et al., 2020; Cieslak et al., 2023; Jiang et al., 2023; Stöckl et al., 2023), and no dedicated literature focusing on the sanding process. In the RoboGrind[14] project, MetaWizard2 is used to structure and store knowledge of human surface treatment experts, and to enable the largely automated creation of robot programs for sanding wind turbine blades.

Perception module A domain-specific perception module for sanding tasks has been developed to provide ground-truth, annotated sensor data as a basis for subsequent task and motion planning. The surface of the workpiece is scanned using a robot-mounted laser line scanner.[15] Surface defects such as holes, dents or bumps are automatically identified by statistical outlier detection (Rusu and Cousins, 2011). I refer to Stöckl et al. (2023) and Alt et al. (2024c) for further details. In the real-world experiment setup, scanning required physical reconfiguration of the robot (see Figure 4.21). For this reason, scanning was realized as an offline process, and queries to the perception module return the last scan result, a point cloud of

---

[14] Research project RoboGrind (2021-2023), funded by the German state of Baden-Württemberg under grant #BW1_0079/01.

[15] Gocator 2490, LMI Technologies Inc., Burnaby, Canada



© 2024 IEEE

Figure 4.18: Left: Scanned point cloud and annotated defect regions (center, enlarged) of a wind turbine blade section. Right: Planned tool path for sanding a section of a wind turbine blade.

the surface with annotated defect regions. Figure 4.18 shows the scanned surface of a wind turbine blade as a raw (left) and annotated (center) point cloud.

Planning module A domain-specific planning module for sanding tasks has been developed. When queried with the point cloud of the surface and the properties of the sanding tool, such as the pad diameter of an orbital sander, the planner generates a 6D tool path comprising positions and orientations to cover the surface in a meandering path with a given amount of overlap. Tool orientations are kept normal to the surface and large, sudden changes in position and orientation are avoided to ensure a homogeneous sanding result. I refer to Raible et al. (2023a) and Alt et al. (2024c) for further details. Figure 4.18 (right) shows a planned tool path for sanding a wind turbine blade section.

Execution module Sanding of wind turbine blades requires robust force control. While the curvature of the workpiece is comparatively homogeneous, the thin fiberglass is prone to vibrations during sanding, which are not absorbed well by the lightweight collaborative arm used in the experiments. For robust force control, the main sanding motion is performed by a `Path Loader Force` ARTM skill for hybrid force-position control, which overlays a planned Cartesian trajectory with a force profile, allowing a set amount of deviation from the planned trajectory to stay within given force limits. The planned Cartesian trajectory is provided by MetaWizard2 via the planning module, while the force setpoint along the Z axis is computed by reasoning over the knowledge base.

Knowledge base The metatask model for iterative sanding is shown in Figure 4.15 (right). The application ontology `surface_treatment.owl` models common tools for sanding, several example workpieces comprising the rotor blade at hand, as well as common materials such as aluminum, fiberglass, wood and steel. The class hierarchy for tools, descendants of `mw:RobotTool`, is connected to the metatask definition via the Prolog predicate `viable_tool`, which permits metatasks to restrict the range of viable tools for the given metatask via TBox axioms. `sft:Sanding`, for example, can only be performed with tools descendant from `sft:Sander`. Tools have associated geometry information, such as a 3D model and a pose (offset and orientation) for attachment to a robot end-effector. Workpieces are associated with a material, as well as an optional 3D model, which can be used for collision-free path planning. For many sanding tasks, material removal largely depends on the contact force during sanding, the end-effector velocity, properties of the tool such as rotation speed, the angle of attack of the tool against the surface as well as the material of the workpiece. For this experiment, an OnRobot collaborative disk sander[16] was used with a pre-set rotation speed (6000 RPM) and angle of attack (5°). Determining the appropriate end-effector velocity and contact force to achieve a desired amount of material removed requires considerable expertise with robotic sanding in general and task-specific knowledge in particular, which cannot be expected from the user of the system. For this reason, the predicate `mat_to_vel_f(M, F, V, T)` permits the KR&R engine to unify an end-effector velocity `V` and force `F` with material `M` and metatask `T`. Expert knowledge can be added to the knowledge base upfront, as illustrated for `sft:Fiberglass` for small amounts of material removed (`fine` finish):

```
mat_to_vel_f(sft:'Fiberglass', 10, 20, MainTask) :-
    kb_call([
        has_parameter(MainTask, P),
        has_type(P, sft:'finish'),
        parameter_value(P, 'fine')
    ]).
```

This permits human experts to build up a knowledge base of suitable task parameterizations from past experience, and avoids requiring the user to specify such parameters at program inference time.

Laboratory experiments The perception, planning and execution modules were evaluated under laboratory conditions on three identical 500 mm x 750 mm wind turbine blade sections (Alt et al., 2024c). Each section is partitioned into four concave segments containing surface defects such as scratches or dents. Each segment was scanned, defects were filled with putty, and the segment was scanned again. For each defect region, several sanding passes were performed with MetaWizard2, and the resulting surface was scanned a third time. Besides evaluating the robustness of the execution module, the sanding trials help determining suitable parameterizations for the force controller to achieve the desired surface quality – i.e., empirical grounding of `mat_to_vel_f`, as no applicable data was available in the literature. The contact force F_z, angle of attack α and number of successive sanding passes were varied to determine suitable values for validation in an operational environment (see below).

Out of 96 total defects, 69 defects were correctly detected by the perception module, with 27 false negatives and 7 false positives. False negatives were mainly

---

[16] OnRobot GmbH, Soest, Germany

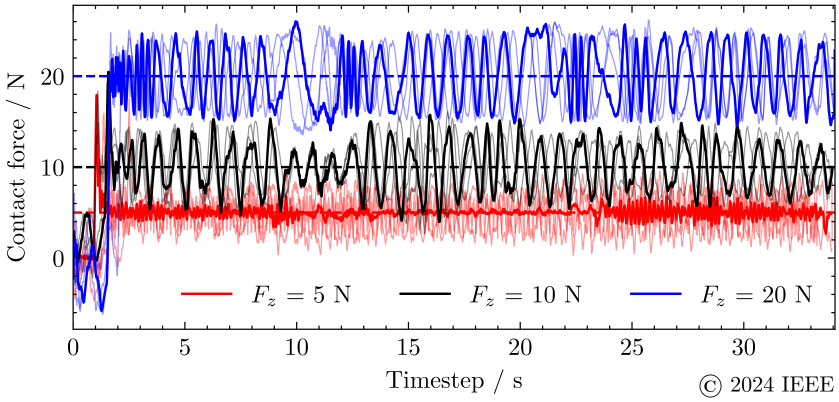


Figure 4.19: Measured force trajectories for 3 different controller parameterizations (force setpoints F_z). Four trajectories per parameterization are plotted (Alt et al., 2024c).

RMSE / mm	MAE / mm	MAX / mm
0.372 ($\sigma = 0.030$)	0.348 (0.034)	1.046 (0,330)

Table 4.4: Root MSE between the planned tool path and the detected surface point cloud (Alt et al., 2024c).

defects in regions with dark discolorations due to dirt, which caused holes in the point clouds. The planning module planned suitable tool paths for all 12 sections. The resulting Cartesian tool trajectory deviates from the ground-truth surface by around 0.4 mm, depending on the error metric (see Table 4.4), which is more than sufficiently accurate for hybrid force-position control. 69% of sanding attempts were successful. Failed attempts were exclusively due to exceeding the force safety limits of the UR10e collaborative arm, that has not been designed to absorb the vibrations that occur during sanding of fiberglass. Measured force trajectories for each of the three tested force setpoints are shown in Figure 4.19. The effect of control parameters on surface roughness metrics is shown in Table 4.3. For subsequent real-world validation, a force setpoint of 10 N and shallow angle of attack $\alpha = 2^\circ$ was chosen for fine sanding, while a contact force of 20 N was chosen for rough sanding. Figure 4.20 sh ows the achieved surface qualities for two surface segments. At forces above and including 10 N, vibrations led to measurably oscillating behavior of the force controller, which did not affect the resulting surface quality.

Real-world validation To validate MetaWizard2 under real-world conditions, a field test was conducted at a partner company specializing in robotic surface treatment.[17] The experiment setup is shown in Figure 4.21. A large section of a wind

---

[17] SHL AG, Böttingen, Germany

Parameters			Metrics	
F_z / N	α / °	Passes	$R_{\mathrm{a}1}$ / μm	$R_{\mathrm{a}2}$ / μm
10	5	5	199.57	122.44
10	**2**	**10**	**169.45**	**104.25**
5	5	5	123.46	162.02
5	2	10	225.01	119.82

Table 4.3: Evaluation of the surface roughness after filling (R_{a1}) and after sanding (R_{a2}) for four different parameter sets (Alt et al., 2024c).

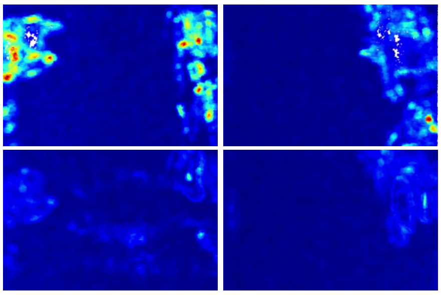


Figure 4.20: Surface roughness before (top) and after (bottom) sanding for two different, filled blade segments (Alt et al., 2024c).

turbine blade is secured on the floor. A UR10e collaborative robot arm[18] is equipped with a laser line scanner[19] for perception. For surface treatment, an OnRobot collaborative sander[20] is mounted to the same robot, though perception and sanding could, in principle, also be realized by two different robots. MetaWizard2 guides the user through the sanding process. To determine the performance of the perception, planning and execution modules for different surface characteristics, two trials are conducted. In the first trial, the rotor blade is scanned as-is, and two passes over the surface are performed. Then, fiberglass putty is applied to the surface to fill in any remaining holes, creating a much rougher surface. Two further passes over the surface are performed.

MetaWizard2 performed well under real-world conditions. The perception module was robust against uncontrolled lighting conditions and provided suitable surface point clouds both before and after putty was applied. It was found that the optical characteristics of the putty necessitated the application of a reflective spray, which did not alter the surface characteristics during sanding in an observable way. The NLP pipeline was robust against the noisy industrial environment. Both rough and fine sanding could be performed as intended, requiring a total of 8 natural-language user interactions per trial. Listing B.1 provides a transcript of an interaction.

Discussion The experiment validates MetaWizard2 as a robot programming system capable of solving real-world surface finishing tasks. Under real-world conditions, the user to configure and execute a challenging remanufacturing task without requiring expert knowledge or human demonstration at runtime. Rather, the task and relevant task knowledge could be specified beforehand by domain experts. Iterative sanding with different target surface characteristics demonstrates

---

[18] Universal Robots A/S, Odense, Denmark

[19] Gocator 2490, LMI Technologies Inc., Burnaby, Canada

[20] OnRobot GmbH, Soest, Germany

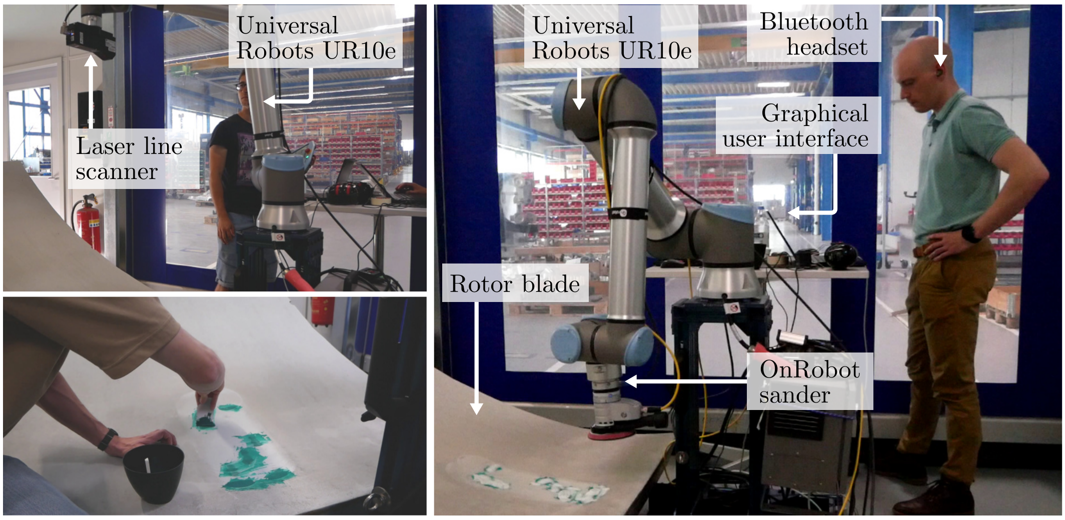


Figure 4.21: Hardware setup for wind turbine refabrication with MetaWizard2. The rotor blade section is scanned with a flange-mounted laser line scanner (top left). Surface defects are filled with fiberglass putty (bottom left). The surface is sanded with a collaborative robot in an interactive process controlled through natural-language interaction (right).

the benefit of interactive, dialogue-based grounding at runtime, as robot programs can be altered (here: reparameterized) to adapt to changing circumstances, such as different user requirements (here: rough v. fine sanding). The experiment provided a first validation of MetaWizard2. A comprehensive evaluation should address a range of diverse tasks and task variants and include scenarios that require more dynamic human-machine interaction, such as handling of errors or acting in dynamic multi-agent environments.

4.2.4.2 Generalization to Related Tasks

One of the core features of MetaWizard2 is the flexible and extensible metatask representation that permits the definition of tasks and workflows via assertions to a knowledge base. MetaWizard2 has been applied to two related surface treatment tasks, purely by adding or modifying knowledge in the KB.

Deburring of gears In a first experiment, the task consists of deburring a gear. The planning and real-world execution environments are shown in Figure 4.22. For this application, the same metatask sequence was used as for sanding of wind turbine blades. A new planning module was added, that uses a commercial planner for automatic path generation on 3D models[21] to generate a planned tool path. The hardware setup consisted of a Fanuc LR Mate 200iD industrial manipulator[22]

---

[21] The CAD2Path planner of ArtiMinds RPS (Schmidt-Rohr et al., 2013)

[22] FANUC, Oshino-mura, Japan

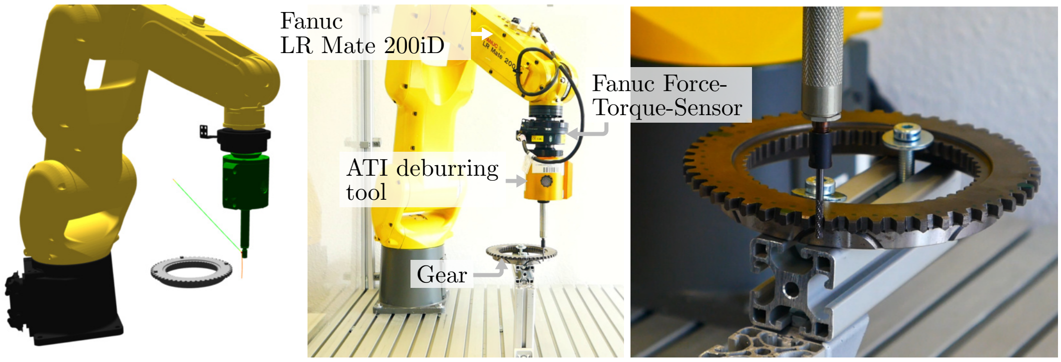


Figure 4.22: Simulated (left) and real-world (center) experiment setup for gear deburring with MetaWizard2.

```
MetaWizard2 : What task do you want me to help with?
Programmer :  Deburring.
MetaWizard2 : Please specify the material of the workpiece.
Programmer :  It's steel.
MetaWizard2 : I've set the material of the workpiece to steel.
MetaWizard2 : Please specify the type of the workpiece.
Programmer :  It's a large gear.
MetaWizard2 : I've set the type of the workpiece to large gear.
MetaWizard2 : What tool should the robot use?
Programmer :  Use an ATI deburring tool.
MetaWizard2 : Tool ATI deburring tool chosen. I've inserted the 3D model at
 the default position relative to the robot.
MetaWizard2 : What is the desired finish?
Programmer :  Rough.
# Program is simulated and executed.
MetaWizard2 : Is another run required?
Programmer :  Yes.
MetaWizard2 : Should I change the finish type?
Programmer :  No, thanks.
# Program is simulated and executed.
MetaWizard2 : Is another run required?
Programmer :  No, that's it.
```

Listing 4.11: Transcript of a dialogue between MetaWizard2 and a human programmer for a deburring task (see Experiment 4.2.4.2).

equipped with a flange-mounted force-torque-sensor and a deburring tool.[23] The workpiece, robot, and tool were added to the KB along with their corresponding 3D and kinematic models. MetaWizard2 solved the task with a total of 5 natural-language user interactions for a first pass, and one user interaction for a subsequent pass. A transcript of the user interaction is shown in Listing 4.11. Note that the symbol grounding mechanism allows humans to answer in a natural manner, including additional phrases beyond the grounded concepts (e.g. “Use an ATI deburring tool.”)

---

[23] ATI Industrial Automation Inc., Apex, USA

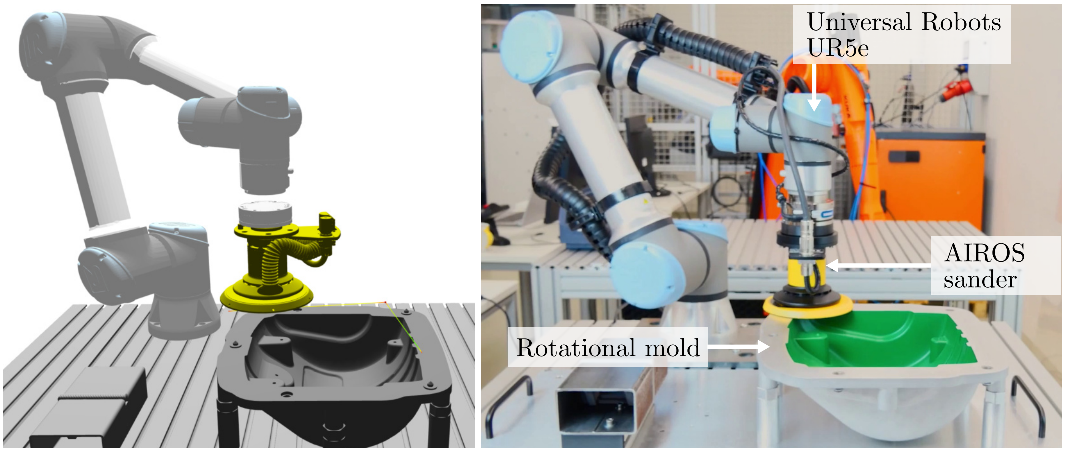


Figure 4.23: Simulated (left) and real-world (center) experiment setup for sanding of rotational molds with MetaWizard2.

Sanding of rotational molds A second experiment considers the task of sanding a rotational mold. The planning and real-world execution environments are shown in Figure 4.23. To realize this application, the iterative workflow shown in Figure 4.15 (right) was changed to a one-shot workflow (left) that terminates after sanding. A UR5e collaborative manipulator[24] was equipped with an AIROS disk sander.[25] The workpiece, robot, and tool were added to the KB along with their corresponding 3D and kinematic models. The same planning module was used as for gear deburring. MetaWizard2 solved the task with a total of 6 natural-language user interactions. A transcript of the interaction is provided in Listing B.2 of the appendix.

4.2.5 Related Work

MetaWizard2 is rooted in the tradition of interactive, “wizard”-based robot programming systems. A majority of interactive programming systems are centered around GUIs for task-based programming (Ajaykumar et al., 2021). The interaction in most graphical programming systems is driven by the human programmer, who specifies and parameterizes a program by a combination of drag-and-drop interaction, kinesthetic teaching on simulated or physical robots, and textual finetuning of robot code. MetaWizard2 realizes a truly interactive programming paradigm, in which the programming system actively asks natural-language questions to the human programmer to ground symbols in the KB. MetaWizard2 must then be contextualized with regard to *conversational* program synthesis systems and *multimodal* systems that combine natural language with visual perception, graphical programming and other modalities.

---

[24] Universal Robots A/S, Odense, Denmark

[25] Mirka Ltd., Jeppo, Finland

4.2.5.1 Multimodal Robot Programming Systems

Huang and Cakmak (2017) introduce Code3, a rapid programming system for mobile manipulators. Its structure resembles that of MetaWizard2 in that it is modular and multimodal, with dedicated modules for visual perception, drag-and-drop graphical robot programming, and kinesthetic PbD. They empirically show that multimodal programming enables non-roboticists to program robots to perform mobile manipulation tasks. In a similar vein, Quintero et al. (2018) propose an interactive, multimodal robot programming system, that combines visualization in augmented reality with gesture-based user input, audio feedback and natural-language voice commands, at a level of modularity and scope similar to MetaWizard2. However, their use of natural language is limited to simple commands, and their system is focused on the intuitive generation and adaption of motion trajectories, rather than structured programs. Several robotics research platforms (Chen et al., 2010; Higy et al., 2018; Asfour et al., 2019) as well as commercially available robots (Pandey and Gelin, 2018; Ionescu and Schlund, 2021) offer built-in support for natural-language interaction, though that support is typically limited to speech-to-text or text-to-speech-based command interfaces, rather than full-fledged language-based programming. Buchina et al. (2016) and Buchina et al. (2019) propose a robot programming system that translates natural-language descriptions of robot tasks to a formal robot program representation that can be translated to an executable robot program for a NAO robot. Like MetaWizard2, it relies on an intermediate, structured program representation, but its scope remains limited to open-loop control. They find that while participants found natural-language to be an intuitive modality for interaction, they also found it challenging to create programs that use abstraction, e.g. to avoid code duplication for repetitive tasks. Most saliently, "the primary finding of the usability tests is that the users require feedback from the robot" – i.e. that the ability of the user to specify their intent via a natural modality is not sufficient for real-world intuitive robot programming, and that an interactive programming pattern centered around bidirectional communication would considerably improve the programming experience.

4.2.5.2 Interactive Robot Programming Systems

One central aim of interactive robot programming systems is to achieve *natural* human-robot interaction in programming. Gorostiza and Salichs (2011) define naturalness in interactions in the words of Wilbur Schramm: "Communication has become to be thought of as a relationship, an act of sharing, rather than something someone does to someone else" (Schramm, 1954). They propose an interactive system for sequencing predefined robot skills that parses human natural-language inputs based on a grammar. Detected symbols are then semantically matched to robot skills, control flow structures and parameters. User dialogue occurs online, interweaved with execution, and the system is able to communicate parts of its current belief state back to the programmer. Beschi et al. (2019) present CAPIRCI, a robot programming system comprising a chat interface for high-level robot programming. CAPIRCI actively asks questions to the programmer ("How many times do I have to perform this task?") at an abstraction level very similar to MetaWizard2. Based on the chat, a graphical robot program is constructed, that can then be adapted via a GUI. Details about the underlying data structures, or whether any learning or KR&R methods were used, are not known.

Several publications put particular focus on natural-language dialogue for symbol grounding. Deits et al. (2013) propose a dialogue-based system that emphasizes the robot's ability to ask clarifying questions to the human user, e.g. to resolve ambiguous instructions. Like MetaWizard2, their system uses natural-language interaction for symbol grounding, but places additional focus on the inverse problem of grounding unconstrained natural language in real-world percepts. They propose to use grounding graphs (Tellex et al., 2011), probabilistic graphical models of the relationship between language concepts and real-world percepts. Rosenthal and Veloso (2011) approach interactive robot programming by modeling human programmers as "observation providers", resources available for robots to obtain information about their environment, but with an associated cost of asking, as the human will be interrupted and must allocate time to respond. They integrate this model into a POMDP representation to plan robot policies that incorporate human interaction to acquire information. In their model, the robot proactively initiates user interaction, and natural-language human-robot dialogue is an explicit part of the program representation. Thomason et al. (2016) use natural-language interaction to enable robots to learn grounding of linguistic symbols. They propose a version of the "I Spy" linguistic game, in which a "learner" and a "teacher" take turns. The teacher offers a description of an object and the learner guesses which object the teacher meant, until the learner guesses correctly. Human and robot switch roles, allowing the robot to learn both by guessing and by observing the human's guesses. In the context of robot programming, his symmetric teacher-learner relationship holds great promise for teaching robots the perception skills required to solve novel tasks. Cakmak and Thomaz (2012) approach dialogue-based robot programming from an Active Learning perspective. They aim to enable robots to ask *good* questions to ask for object labels, request demonstrations or ask for the relevance of a feature in the context of a given task. They study human question-asking and question-answering behavior to derive formulae for questions that are most likely to elicit useful responses, and translate those learnings into a question-asking system for a robot. Their approach is unique in that they address the technical question of enabling a *robot* to ask good questions by studying how *humans* ask questions, highlighting the need for studies of human behavior in order to develop good technical robot programming systems.

4.2.5.3 Robot Programming with Vision-Language-Action Models

Criticism of grammar-based approaches (Chen et al., 2010; Gorostiza and Salichs, 2011; Buchina et al., 2016) is they require the human programmer to express concepts in terms of an application-specific grammar with restricted syntax and vocabulary. Deep LLMs and multimodal VLA models promise an alternative: Via training on web-scale language and action datasets, deep neural architectures can map linguistic concepts to visual percepts or robot actions for a wide range of tasks and domains (Li et al., 2024). Ahn et al. (2022) were among the first to explicitly recognize the potential of LLMs for using LLMs as a general-purpose platform for symbol grounding. SayCan uses an LLM to score the likelihood that a given robot skill will make progress toward achieving a high-level objective specified in natural language. SayCan uses natural language both as an interface representation to obtain inputs from a human user, but also as an internal representation for reasoning and for explaining the resulting program to the programmer. Wu et al. (2023c) propose SymbolLLM, a neurosymbolic reasoning system capable of extracting symbolic semantics of activities from images. They use a vision-language model to extract symbolic descriptions of images of scenes, and process the parsed symbols in a symbolic reasoning systems to derive fuzzy logic rules describing the observed activity. Hsu et al. (2023) propose LEFT, a system that uses an LLM to ground human queries in multiple modalities such as 2D and 3D images, human motions and robotic manipulation actions, but produces code in a logic-based programming language that can then be executed in a logic programming system. By querying it in natural language, human programmers can use LEFT to elicit the latent knowledge implicit in LLMs, and to make it accessible as executable logic rules which can then be used for symbolic reasoning. LLMs, however, only produce *approximately correct* grounding results that reflect the comprehensiveness and correctness of their training dataset (Pavlick, 2023). Combining LLMs with symbol processing systems to neurosymbolic cognitive architectures may improve the overall trustworthiness of the system, while maintaining its flexibility (Jokinen, 2024).

The use of large-scale deep neural models promises not only more generalizable symbol grounding, but also novel, interactive programming paradigms. “ChatGPT for robotics” (Vemprala et al., 2023), for example, proposes to take the human programmer out of the deployment-improvement loop, and rather place them “on” the loop, giving high-level natural-language instructions to a programming system in which a LLM takes the primary role of instructing the robot. Similarly, several end-to-end systems have been proposed, that directly synthesize executable robot code (Liang et al., 2023; Singh et al., 2023; Wu et al., 2023a) or low-level robot actions (Reed et al., 2022; Ghosh et al., 2024; O’Neill et al., 2024) from natural-language task descriptions. While showing impressive generalization and high-level language understanding abilities, current end-to-end program synthesis approaches often lack the causal understanding (Ashwani et al., 2024), commonsense physics knowledge (Yildirim and Paul, 2024) and “system 2”-level reasoning abilities (Bellini-Leite,

2024) required to generalize to highly specialized domains, such as many industrial robotics applications, for which little to no training data is included in web-scale datasets. They do, however, demonstrate that natural language is not only an intuitive modality for humans to interact with AI systems, but also a potentially promising internal representation for knowledge representation within AI systems.

4.2.6 Discussion

MetaWizard2 is a modular robot program synthesis system centered around dialogue-based symbol grounding as well as a metatask representation that models both knowledge of the task itself as well as of the programming process to achieve at the task. At the architecture level, modularity ensures that modules for perception, reasoning, interaction and program generation can be swapped to support e.g. other sensory modalities, planners or reasoners. Due to the explicit metatask representation, metaprogramming – changing the programming workflow – is reduced to adding or modifying knowledge in the knowledge base, and does not require reprogramming. As such, MetaWizard2 can be extended to incorporate metacognition, i.e. automatic reasoning about the programming process itself, in future work (Beetz et al., 2023).

The current iteration of MetaWizard2 grounds symbols by natural-language interaction during the programming process. It addresses the core challenge of symbol grounding, the matching of natural-language concepts to real-world entities, by nearest-neighbor search in a word embedding space. While this is a universal approach that avoids domain- or concept-specific heuristics and, depending on the quality and breadth of the used embedding, generalizes to arbitrary domains, it cannot account for complex semantics that spread across several syntactic chunks. LLMs, that ingest input data spanning a context window of tens to hundreds of thousands of tokens, promise an even more general and robust approach for concept matching. Section 4.3.1.1 describes the use of LLMs for concept matching in the context of MetaWizardLLM.

Likewise, MetaWizard2 can be extended by neurosymbolic vision-language models like SymbolLLM (Wu et al., 2023c) or LEFT (Hsu et al., 2023) to automatically ground the environment in symbols or symbolic rules and continuously update the knowledge base. In this way, the behavior of MetaWizard2 can be extended to dynamically react to changes in the environment, avoiding the need for explicit perception routines as part of the metatask. For the considered sanding use case, for example, MetaWizard2 asks the programmer whether the sanding procedure is finished or whether another pass is required. A large-scale neurosymbolic vision-language model could learn a symbolic rule for task success over time, that relates the perceived state of the environment to task success and avoids repeated prompting of the human programmer. Likewise, as suggested by Rosenthal and Veloso (2011), an explicit model of the human programmer as an information resource could enable MetaWizard2 to reason about which situations warrant human dialogue, and in which situations heuristics or commonsense knowledge should be used for grounding.

4.3 Prompt-based Program Synthesis with Large Language Models

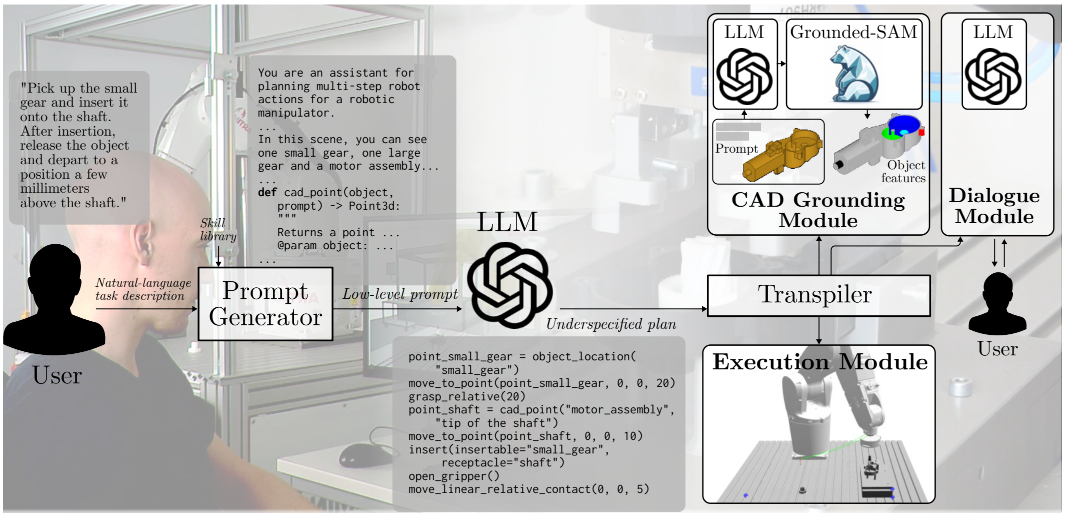
Sections 4.1 and 4.2 introduce MetaWizard and MetaWizard2, two members of a family of program synthesis systems that permit human users to specify tasks in an intuitive manner and automatically infer executable robot programs to solve them. MetaWizard and MetaWizard2 use KR&R techniques with explicit, semisymbolic representations of task and domain knowledge. Both approaches ease the burden of programming for the user by imposing the modeling of task and domain knowledge on domain experts.

The generalization abilities of large-scale multimodal foundation models suggest that they represent vast amounts of latent knowledge. In the context of robotics, VLA models trained on web-scale datasets have been used for motion (O'Neill et al., 2024; Ghosh et al., 2024; Zitkovich et al., 2023) and task planning (Liang et al., 2023; Singh et al., 2023; Vemprala et al., 2023) in a zero-shot manner, i.e. without requiring task- or domain-specific finetuning. This section investigates the use of LLMs and multimodal foundation models for interactive robot program synthesis. It introduces MetaWizardLLM, a MetaWizard variant that combines the modular architecture and dialog-based principle of MetaWizard2 with a novel task grounding mechanism based on LLMs and multimodal RAG. It explores to what extent LLMs and large-scale multimodal models represent generalizable, latent knowledge, and to what extent they can implicitly reason about that knowledge to fulfil user queries without dedicated finetuning.

The work presented in this section is the subject of ongoing research. In Experiment 4.3.2, a functional prototype of MetaWizardLLM is validated in the context of industrial gear assembly. Moreover, Experiment 4.3.2 showcases the integration of the MetaWizardLLM with SPI into a framework for AI-enabled robot programming that affords both robot program synthesis and optimization.

4.3.1 MetaWizardLLM

Modern LLMs trained on web-scale datasets have been shown to be capable of generating program code for executing tasks described in natural language (Poldrack et al., 2023; Wang and Chen, 2023; Yang et al., 2024). The required understanding of natural-language syntax and semantics, task and domain knowledge as well as programming language syntax and semantics are implicitly represented in the weights of the networks. In the domain of robotics, Liang et al. (2023) introduce CaP, an LLM-based system for robot program synthesis that fulfills natural-language user queries by generating executable python code. CaP generates and parame-



"Pick up the small gear and insert it onto the shaft. After insertion, release the object and depart to a position a few millimeters above the shaft."

```
You are an assistant for
planning multi-step robot
actions for a robotic
manipulator.
...
In this scene, you can see
one small gear, one large
gear and a motor assembly...
...
def cad_point(object,
   prompt) -> Point3d:
   """
   Returns a point ...
   @param object: ...
...
```

```
point_small_gear = object_location(
    "small_gear")
move_to_point(point_small_gear, 0, 0, 20)
grasp_relative(20)
point_shaft = cad_point("motor_assembly",
    "tip of the shaft")
move_to_point(point_shaft, 0, 0, 10)
insert(insertable="small_gear",
     receptacle="shaft")
open_gripper()
move_linear_relative_contact(0, 0, 5)
```

Figure 4.24: The MetaWizardLLM robot program synthesis system. MetaWizardLLM synthesizes executable robot programs from natural-language task descriptions. Given a task description provided by a human programmer, underspecified plan is generated by an LLM, which is grounded by a combination of RAG-based spatial reasoning over a computer-aided design (CAD) model of the environment, as well as natural-language dialogue with the programmer.

terizes function calls to a Python API for controlling the robot, a principle since realized by several related program synthesis systems (Wu et al., 2023a; Singh et al., 2023; Wang et al., 2024b; Luo et al., 2024). Based on the intuition that LLMs can generate executable high-level program code without requiring task- or domain-specific finetuning, MetaWizardLLM uses an LLM to bootstrap an underspecified plan given a natural-language task description, and then grounds the plan using natural-language dialog as well as a novel grounding module for reasoning over CAD representations of the environment.

The architecture of MetaWizardLLM is shown in Figure 4.24. The two processing steps realized by MetaWizardLLM – synthesis of underspecified plans and symbol grounding via RAG over CAD models – are described in Sections 4.3.1.1 and 4.3.1.2.

4.3.1.1 LLM-based Generation of Underspecified Plans

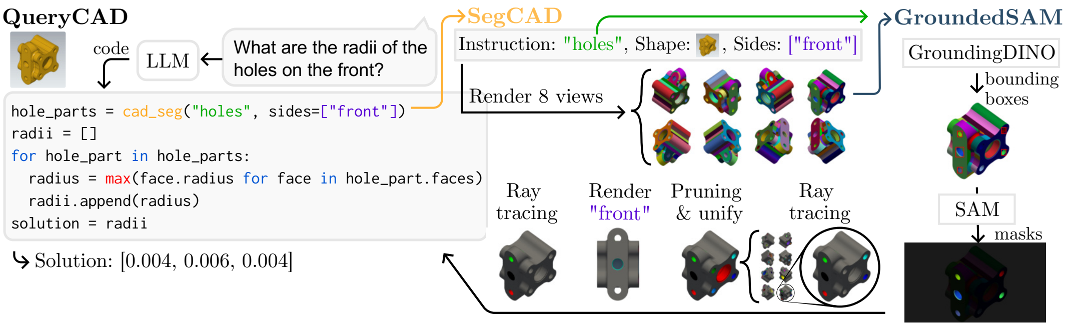
To generate executable robot programs for high-level tasks, MetaWizardLLM converts a high-level instruction into an underspecified plan, which is then grounded through a combination of LLM-based inference over a CAD representation of the environment as well as interactive user dialogue. For natural-language user interaction, such as for obtaining the initial instruction, MetaWizardLLM uses the same dialog module as MetaWizard2, except that speech-to-text conversion is performed via the Whisper speech processing network (Radford et al., 2023). For symbol grounding, the embedding distance-based approach is replaced by a prompt-based

```python
# Ask user for high-level task description, e.g. "Pick up the small gear and insert it onto the spindle tip."
question = dialog_system.q_and_a("What task do you want me to help with?")

# Generate a prompt containing API description, description of the environment and user query
prompt = prompt_generator.get_task_planning_prompt(question)

# Use an LLM to bootstrap an underspecified plan
llm = LLM(host="api.openai.com", port=443)
plan = llm(prompt)

# Parse the generated Python code and traverse the tree to ground the underspecified plan
import ast
tree = ast.parse(plan)
grounding_module.visit(tree)
```

Listing 4.12: High-level pseudocode for the MetaWizardLLM program synthesis and task grounding steps.

approach using the 8B variant of the LLama3 LLM (Dubey et al., 2024). The prompts used for parsing concepts chosen by the user from a list of available options as well as for extracting the values of numerical parameters are provided in Listings C.2 and C.1. The initial instruction is parsed as freeform natural-language text to be processed by downstream LLMs (see below). The task to be solved is described at a very high level of abstraction, such as “put the glass into the sink” or “plug the cable into the connector”.

Prompt generation Given the instruction, a prompt is assembled to generate an underspecified plan that performs the requested high-level task. Beyond the instruction, the prompt contains a natural-language description of the environment, additional natural-language information about the task domain and any constraints that may be relevant to the task, as well as the API available to the plan-generating LLM. The prompt structure follows the structure used by the OpenAI LLM API (OpenAI, 2024b). It is divided into a *system prompt*, that provides the LLM with a natural-language description of the role it is to perform, the expected output format (e.g. Python code), a natural-language description of the scene, a list of environment objects, as well as all available API functions including their signatures and docstrings, but omitting function bodies. An exemplary prompt for the gear assembly task considered in Experiment 4.3.2 is shown in Listing C.3. Beside the system prompt, the prompt contains an *assistant* section, that provides the LLM with one or more exemplary interactions to illustrate the usage of concepts introduced in the system prompt, such as the semantics of functions. For the gear assembly task, the assistant section contains an example of how some of the provided API functions can be used to transfer an object from one point to another. The third section of the prompt contains the user instruction. Note that the system prompt is the primary



```
hole_parts = cad_seg("holes", sides=["front"])
radii = []
for hole_part in hole_parts:
  radius = max(face.radius for face in hole_part.faces)
  radii.append(radius)
solution = radii
```

Figure 4.25: Overview of the CAD grounding module. Given a textual prompt, an LLM generates code, which in turn parameterizes and calls a DNN-based segmentation pipeline (SegCAD) (Kienle et al., 2025).

source of concrete contextual knowledge of the task at hand, and any potential specialized domain knowledge, for the LLM. Providing contextual knowledge via a prompt alleviates the need for both finetuning and explicit, structured knowledge representations, relying instead on prior knowledge implicitly represented in the weights of the LLM to “understand” and interpret the system prompt. The overall prompt is generated automatically from the user instruction, natural-language domain and environment descriptions, as well as the source code of the Python robot API. A generated prompt for a gear assembly task (see Experiment 4.3.2) is shown in Listing C.3 of the appendix.

Plan generation The plan itself is generated by querying a GPT-4 omni (GPT-4o) LLM (OpenAI, 2024a). Like other models from the GPT-4 model family, GPT-4o directly outputs Python code with a very low likelihood of syntax errors (Poldrack et al., 2023). The generated plan for a gear assembly task is shown in Figure 4.26 (right). As the robot API provided to the LLM via the system prompt contains grounding functions such as `cad_point` or `object_location`, objects in the plan are referred to by string identifiers, and poses as local variables, to which the return values of the grounding functions are assigned. By parsing and traversing the generated code with Python’s built-in abstract syntax tree (AST) parser, grounding functions are mapped to their corresponding implementations in the respective grounding module (for Experiment 4.3.2, the CAD grounding module). During AST traversal, an executable ARTM robot program is incrementally constructed, as generated API function calls are mapped to instantiations of the corresponding ARTM primitives (e.g. `Grasp Relative`, `Move to Point` etc.)

4.3.1.2 Symbol Grounding via Retrieval-Augmented Generation

Object locations and features such as holes or pins are grounded by RAG over a CAD representation of the environment (Kienle et al., 2025). An overview of QueryCAD, the CAD grounding module, is shown in Figure 4.25.

QueryCAD follows a similar paradigm as MetaWizardLLM itself, in that it uses an LLM to generate Python code given a natural-language query, and interprets the generated code to answer the query. Code generation is performed by prompting GPT-4o (OpenAI, 2024a) with a prompt of similar structure as that used by MetaWizardLLM. The generated code has access to an API of CAD processing functions, which in turn parameterize and prompt a neural CAD processing pipeline to perform low-level perception and reasoning tasks such as finding all instances of a feature, such as holes,pegs or pins, on an object, or extracting numerical information such as the poses of features relative to the object origin. The CAD segmentation pipeline performs the following steps:

1. **2D rendering:** Given a viewing angle, the CAD object is rendered to a 2D image using orthographic projection. To facilitate segmentation and feature recognition, the faces of the object are colored in contrasting colors.

2. **Instance segmentation:** The GroundedSAM image segmentation framework (Ren et al., 2024), which combines the GroundingDINO (Liu et al., 2024a) and Segment Anything (Kirillov et al., 2023) networks, is used to segment the rendered image into regions that correspond to the prompt.

3. **Image-to-CAD registration:** To obtain the CAD faces corresponding to the segmented regions in the 2D image, raytracing is performed from each pixel of the segmentation mask to the 3D CAD object.

4. **Multi-view rendering:** As not all object features are visible from all viewing angles, steps 1-3 are performed for each of 6 different viewing angles (top, bottom, right, left, front and back), and the union of all segmented faces is returned. If the prompt for the segmentation network specifies a particular viewing angle, only the faces visible from that viewing angle are returned.

Kienle et al. (2025) explain the segmentation pipeline in additional detail. Note that, following the RAG paradigm (Lewis et al., 2020), the prompt for the segmentation pipeline is generated by evaluating the Python code in turn generated by GPT-4o, and the segmentation pipeline accesses structured ground-truth data, notably by exploiting the structure of CAD objects as graphs of faces and edges. MetaWizardLLM uses QueryCAD the same way, by generating code that, when interpreted, parameterizes a prompt for QueryCAD.

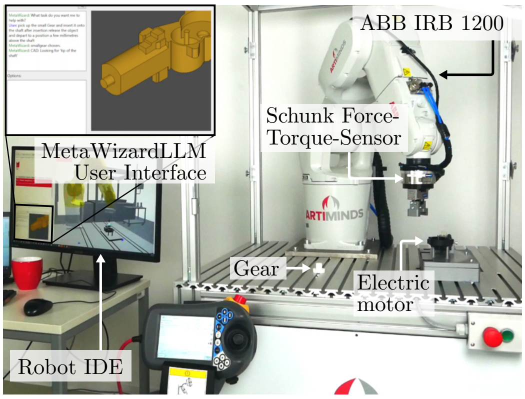
4.3.2 Experiments

MetaWizardLLM was validated on an industrial gear assembly scenario.

Experiment setup The experiment setup is shown in Figure 4.26. An ABB IRB 1200 industrial manipulator[26] is equipped with a Schunk FT-AXIA80 force-torque

---

[26] ABB Ltd., Västerås, Sweden



```python
point_small_gear = object_location("small_gear")
move_to_point(point_small_gear, 0, 0, 20)
grasp_relative(20)
point_shaft = cad_point("motor_assembly", "tip of the shaft")
move_to_point(point_shaft, 0, 0, 10)
insert(insertable="small_gear", receptacle="shaft")
open_gripper()
move_linear_relative_contact(0, 0, 5)
```

Figure 4.26: Experiment setup (left) and generated underspecified plan (right) for a gear assembly task.

sensor and a pneumatic gripper.[27] The task consists of assembling the gearing mechanism for an electric window motor,[28] which is clamped to the table.[29] The robot is to pick up a gear at a specified position and insert in onto a shaft in the motor housing.

For simulation, motion planning and execution, the ArtiMinds RPS industrial robot IDE is used in a manner identical to Experiment 4.2.4. The robot API function signatures and docstrings that form the main part of the system prompt for underspecified plan generation are extracted directly from the Python source code of the execution module, that instantiates ARTM skills and subprograms in the robot IDE. This avoids explicit knowledge engineering and uses programming knowledge already implicit in the code base.

The programmer initially specifies the task in natural language. MetaWizardLLM echoes its current belief state – what it understood from the user, and what grounding task it is currently performing – back to the programer both in natural language and graphically (see Figure 4.26 (left)).

Results A transcript of the user interaction is shown in Listing B.3 of the appendix. The prompt generated from the robot API and the user query is shown in Listing C.3. The underspecified plan generated by the LLM is shown in Figure 4.26 (right). Underspecified plan generation is fully hidden from the user and does not require user interaction. MetaWizardLLM grounded the approach, grasp and depart sequence to fully autonomously via the CAD grounding module, as the description "the small gear" matched exactly one object in the scene (`small_gear`), whose location was known. Note that the underspecified plan contains appropriate offsets (here, 20 mm in the Z direction for `move_to_point` and `grasp_relative`), due

---

[27] Schunk SE & Co KG, Lauffen am Neckar, Germany

[28] Robert Bosch GmbH, Stuttgart, Germany

[29] The motor and clamping mechanism are provided by the Learning Factory for Global Production at the Institute of Production Science (wbk) at Karlsruhe Institute of Technology.

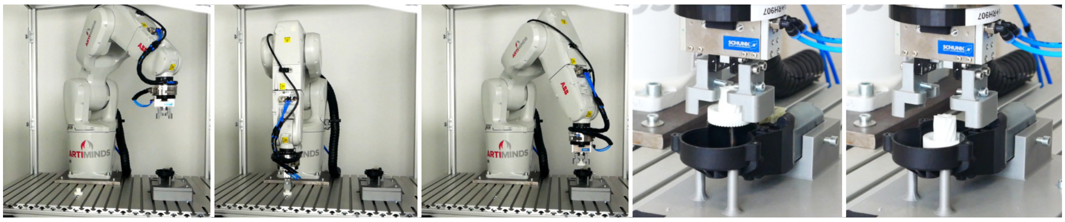


Figure 4.27: Execution of the robot program generated by MetaWizardLLM.

to the requirement of approaching "above the target" specified in the docstring of `grasp_relative`.

The insertion sequence (approach of the peg with the grasped gear, force-controlled establishment of contact, force-controlled insertion, and departure with an open gripper) was generated fully autonomously except for one ambiguity concerning the "shaft" feature of the electric motor. QueryCAD identified two shafts on the object, and asked the user which shaft they meant, both in natural language and by highlighting the two shafts with different colors in a 3D rendering of the electric motor. The user specified the intended shaft in natural language ("the magenta part"). After interactive CAD-based grounding of the underspecified plan, the resulting fully parameterized ARTM program was simulated and executed on the robot, solving the task.

Discussion MetaWizardLLM permitted the synthesis of a fully grounded, executable robot program for a gear insertion task given a minimal, natural-language task description as well as a natural-language scene description. User interaction was limited to the initial specification of the task, as well as the resolution of an ambiguous object feature. The combination of natural-language interaction with a GUI for displaying the belief state of the system (here, highlighting detected object parts) shows promise, as it raises the level of abstraction of programming to the point at which task-, domain-, robotics- or AI-specific knowledge are no longer required on the part of the programmer. To fully ascertain the benefits and limitations of MetaWizardLLM, larger-scale experiments on a range of different, more complex use cases as well as multi-participant user studies are the subject of future work.

4.3.3 Related Work

With CaP, Liang et al. (2023) established several of the core operating principles of MetaWizardLLM. CaP leverage an LLM to generate Python code representing nested policies at several levels of abstraction. They observe that due to the large amount of Python code in web-scale text datasets, LLMs trained on such datasets are highly proficient at composing and chaining abstractions used and expressed in code (Chen et al., 2021), such as function calls or control flow structures. Based on this observation, they propose a prompt structure that encodes existing knowledge about robot skills, their semantics and the associated constraints as API functions, their function signatures and docstrings as well as usage examples. This approach enables LLMs not specifically finetuned for robot program synthesis to generate Python code that combines and parameterizes calls to a robot control API to solve tabletop and mobile manipulation tasks. Arenas et al. (2024) propose a framework for prompt engineering based on CaP, that helps robot programmers use state-of-the-art prompting techniques such as chain-of-thought, example- or instruction-based prompts. ProgPrompt (Singh et al., 2023) follows a similar approach of generating Python code that uses and parameterizes a robot API, with a prompt structure similar to CaP. SayCan (Ahn et al., 2022) is the intellectual predecessor of CaP and ProgPrompt, in that it generates natural-language plans given natural-language descriptions. Generating Python code, however, has the additional advantage that the generated outputs follow a predefined, easily parseable syntax with well-defined semantics, and can be directly executed.

The paradigm of leveraging the particularly high robustness of LLMs for code generation to solve non-coding tasks has been explored in prior work. Wang et al. (2024b) show that LLM-based AI agents that represent policies as executable code tend to perform better across a wide range of tasks than agents using other representations. Yang et al. (2024) propose that code generation is powerful means to enable LLMs to perform a wide variety of tasks at a high level of performance by granting them access to (software) tools or (hardware) sensors and actuators. They provide an overview of numerous works relying on code for tool use, and further argue that code generation provides LLMs with several abilities required by intelligent agents, such as advanced environment perception, planning, action grounding or memory organization. Tool use via code generation provides a flexible, general-purpose alternative to traditional RAG approaches, that perform retrieval through predefined operations on structured data representations (Lewis et al., 2020; Yu, 2022; Yu et al., 2022; Salemi and Zamani, 2024), or text-to-SQL approaches, that generate sequences of database operations in a database DSL (Baig et al., 2022; Fu et al., 2023; Gao et al., 2024). MetaWizardLLM explores the paradigm of code-based tool use for robot program synthesis by generating an underspecified plan that can ground itself by making function calls to a grounding module: the grounding module, in turn, follows the same paradigm to flexibly access information encoded in CAD data.

4.3.4 Discussion

MetaWizardLLM realizes robot program synthesis via LLM-based generation of an underspecified plan, that is grounded via RAG over a CAD representation of the environment. From a programmer's perspective, the primary difference between MetaWizardLLM and other MetaWizard variants is that the task is specified by a short natural-language description. From a metaprogramming perspective, the difference is more substantial. MetaWizardLLM realizes metaprogramming, the specification of the behavior of the program synthesis system itself, via prompt engineering, relying on semi-structured representations in natural language or (pseudo-) code to represent task and domain knowledge. This makes it considerably easier for domain experts to make highly specialized knowledge accessible to the program synthesis system, facilitating the extension of the system to new use cases or the adaptation of the system to related task variances. Via prompt engineering, human metaprogrammers can influence the behavior of the system via an intuitive modality. While this is alleviates the need for metaprogrammers to be proficient in logic programming, it also means eschewing mechanistic interpretability – the prompt engineer will generally not understand the mathematical computations by which the LLM arrives at the generated plan by way of the prompt, and will not be able to make hard a priori guarantees about the behavior of the program synthesis system. Hybrid systems that combine mechanistically interpretable algorithms for critical parts of the metatask, such as program generation, with opaque, LLM-based modules for less critical aspects, such as text-to-speech conversion, may constitute an acceptable tradeoff for many applications that do not require full interpretability.

In Experiment 4.3.2, MetaWizardLLM shows a surprising ability to *plan*, and respect the pre- and postconditions of the used robot skills, despite the lack of explicit mechanisms enforcing these conditions. Consider the implicit definition of a precondition for the insertion skill in Listing C.3. In the docstring for `insert`, three preconditions are specified in natural language: The caller must ensure that the object and receptacle are aligned, that the object is grasped by the robot, and that the robot is positioned above the receptacle. These preconditions are fulfilled by the generated code, in that the object is first grasped before being inserted, and that the approach movement is parameterized with the location of the shaft onto which the gear is to be inserted, with an additional offset along the Z axis to ensure that the gear is above the shaft before insertion. The planning mechanism by which the preconditions are taken into account is implicit and arises during training of the LLM. It is elicited at query time by the particular prompt structure, likely because the preconditions for functions are often mentioned in docstrings, and the training dataset contains large amounts of source code. In preliminary experiments, larger models such as GPT-4o or Llama3 405B respected such planning constraints with much greater reliability than smaller models such as Llama3 7B (Dubey et al., 2024). The robustness of the implicit planning capabilities of MetaWizardLLM and the influences of model sizes and other factors will be studied empirically on larger-scale benchmarks.

Experiment 4.3.2 validates MetaWizardLLM on a simple industrial usecase. Future work will qualitatively and quantitatively evaluate MetaWizardLLM on a diverse range of tasks and environments. Consideration of more complex applications requires the integration of real-world perception modules, such as the RoboSherlock-based perception module of MetaWizard. Multimodal vision-language foundation models such as GPT-4o promise generalizable cross-modality reasoning abilities that permit the implementation of general-purpose visual perception modules able to ground near-arbitrary symbols in natural language.

Additional experimentation is required to determine the extent to which MetaWizardLLM is limited by the lack of causal understanding of LLMs, and the resulting "hallucinations" – erroneous model outputs that may be difficult to discern from correct outputs – observed in a wide range of LLM-based applications (Gunjal et al., 2024). Theory and early empirical results point to RAG as realized by the CAD grounding module to reduce hallucinations and improve overall robustness (Lewis et al., 2020; Chen et al., 2024). As model errors will always be present in ML models, additional symbolic processing and validation may be required to ensure that critical parts of the program synthesis pipeline, such as program execution, are safe, particularly in human-robot collaborative applications.

4.4 Discussion

The family of program synthesis systems presented in this chapter bootstrap executable robot programs by generating and manipulating programs in an explicit, graphical, symbolic source program representation. MetaWizard and MetaWizard2 perform reasoning over a structured, semi-symbolic knowledge base to interpret actions demonstrated in VR or to interactively bootstrap a task sequence through dialogue with a human programmer. MetaWizardLLM combines LLMs for natural-language understanding and code generation with a CAD backend to generate and ground plans that satisfy a natural-language task description. All three are general-purpose program synthesis systems that are not restricted to particular tasks or domains; in fact, by virtue of being *metaprogramming* systems, they afford task and domain transfer purely by adding additional task or domain knowledge. This combination of universality and applicability – as the synthesized programs concisely represent highly complex behavior, and are directly executable on the robot – is achieved by virtue of the fact that the source program representation of NRPs is a skill-based representation with known semantics: For each robot skill, the (intended) robot behavior and possible failure modes are known, as are the semantics of skill input parameters. Robot skills are combined to complex tasks via symbolic composition, again with known and defined semantics. The symbolic nature of NRP source programs facilitates the expression of task abstraction and task grounding rules (Sections 4.1.4 and 4.1.5), metatasks (Section 4.2.2), and structured prompts (Section 4.3.1.1), and allows for human programmers to read, understand and modify the synthesized programs.

The development and study of the presented robot program synthesis systems provides answers to the research questions posed in the beginning of the chapter.

1. **How can humans express their intent in a way that permits precision and detail where required, while avoiding redundant or unneeded specification?**

MetaWizard, MetaWizard2 and MetaWizardLLM emphasize the use of intuitive, high-level interfaces such as VR demonstrations or natural-language dialogue. To the programmer, programming occurs at a raised level of abstraction, and is reduced to demonstrating the intended behavior, conversationally answering questions, or describing the task at hand in natural language. Both VR demonstrations as well as natural language are inherently ambiguous and require *interpretation* to serve as inputs to a technical program synthesis system. MetaWizard proposes a Prolog-based reasoning mechanism for action interpretation, MetaWizard2 uses token embeddings to interpret human inputs, and MetaWizardLLM relegates action interpretation to an LLM. Each method has limitations: Rule-based action interpretation requires possibly extensive specification of task knowledge; symbol grounding by nearest-neighbor search in embedding space may disregard cross-token context; and LLM-based action interpretation is subject to hallucinations. All three approaches achieve sufficient performance for validation on real-world use cases, and constitute promising avenues for further inquiry.

2. **How can executable robot programs be synthesized from underspecified task descriptions provided by human programmers?**

In this chapter, two paradigms for program synthesis were explored: MetaWizard proposes an end-to-end paradigm, which generates executable robot programs from human VR demonstrations by reasoning over a semisymbolic knowledge base, bootstrapping a task sequence by searching the space of known tasks and then grounding the resulting underspecified plan in sensory data and background knowledge. MetaWizard2 and MetaWizardLLM propose an interactive paradigm, in which robot programs are created online through interaction with a human programmer, and programming, planning, execution and dialog are orchestrated by a metatask or prompt that is, again, structured knowledge. Both approaches emphasize the need to ground the symbols that make up programs in real-world entities, such as objects or robot actions. The development of universal, multimodal mechanisms that robustly and precisely identify symbols in real-world percepts is a promising direction for future work, particularly given the generalization abilities promised by large-scale, pretrained multimodal neural networks.

3. **How can human expert knowledge be represented, stored and leveraged for the purpose of automatic or interactive program synthesis?**

The MetaWizard family of program synthesis systems shifts some of the burden of programming – the specification of *how* the robot ought to behave to solve a task – to a prior *metaprogramming* phase, in which the knowledge base is populated. Metaprogramming, however, must only be performed once, and then greatly simplifies any subsequent programming. The meta-level question of how expert domain and task knowledge can be codified as knowledge for use by AI systems is a highly relevant research question in its own right. For most real-world applications, domain experts are not ontology designers, logic programmers or prompt engineers, and user-facing systems must be designed to facilitate knowledge transfer from humans to machines. Future iterations of MetaWizard, MetaWizard2 and MetaWizardLLM will be given the ability to *learn* domain knowledge as they are used, from interactions with human programmers or directly from sensory experience.

4. **To what extent can explicit robot program representations support the synthesis of robot programs that can solve complex tasks while being interpretable by human programmers?**

Like the program optimization algorithms proposed in Chapter 3, the MetaWizard program synthesis systems leverage the source program representation of NRPs for program execution. This means that the synthesized robot behavior is interpretable through a skill-based robot program with well-defined semantics: Each robot action executes a robot skill, with a semantically meaningful parameterization and known expected robot behavior. The synthesized programs can be read, modified and audited by human programmers; depending on the used representation, they may be certified to comply with safety requirements or company standards. MetaWizard is exclusively based on logic programming, traditional path planning and code generation, and can be considered mechanistically interpretable, save for the perception pipeline. MetaWizard2 relies on neural networks for NLP, which reduces its mechanistic interpretability to an extent, but its main computational mechanisms remain interpretable. MetaWizardLLM relies largely on black-box LLMs for reasoning, but uses structured CAD data of the environment for symbol grounding. In all three cases, tracing an inserted skill or synthesized program back to individual entries in the knowledge base, real-world percepts or individual CAD features may be challenging. The generation of user-facing explanations for generated robot programs is an open research question that merits further inquiry.

4.5 Conclusion

4.5.1 Summary

This chapter introduced a family of program synthesis systems for NRPs. Section 4.1 introduced MetaWizard, a system for the end-to-end synthesis of NRP source programs given human VR demonstrations. MetaWizard relies on an explicit, semisymbolic representation of commonsense, domain and task knowledge as well as reasoning algorithms for task abstraction and task grounding. Experiment 4.1.7 validates MetaWizard in the context of manipulation tasks in a retail context.

Section 4.2 introduced MetaWizard2, a dialog-based MetaWizard alternative that synthesizes NRP source programs through interactive natural-language dialogue with a human programmer. Both the robot behavior to solve the task at hand as well as the behavior of MetaWizard2 are encoded in a structured metatask representation, whose underspecified parameters are grounded by MetaWizard2 through proactive asking of questions. Experiment 4.2.4 validates MetaWizard2 on a remanufacturing task as well as two related surface treatment tasks.

Section 4.3 presented MetaWizardLLM, a program synthesis system that leverages a chain of LLMs for generating NRP source program given natural-language task descriptions and interactive user dialogue. MetaWizardLLM bootstraps underspecified plans by prompting a code-generating LLM with a description of available robot skills as well as natural-language descriptions of the task and scene. The underspecified plan grounds itself by RAG over a CAD representation of the environment. MetaWizardLLM is validated on an industrial gear insertion task.

4.5.2 Outlook

The MetaWizard family of program synthesis systems remains under active development, with the objective of integration into a commercial robot IDE (Schmidt-Rohr et al., 2013). Future iterations of MetaWizard will place an emphasis on zero-shot task and domain generalization as well as flexible grounding. The use of multimodal foundation models that integrate vision, language and other modalities such as tactile information (Kienle et al., 2024), CAD models (Kienle et al., 2025) or scene graphs (Kenghagho Kenfack et al., 2020) promises to achieve both objectives with the same underlying technology. Trained on a broad range of tasks and domains, foundation models may endow AI systems with general-purpose reasoning abilities and can solve many queries without requiring task- or domain-specific finetuning. With the use of large-scale foundation models, the burden of metaprogramming will be shifted away from knowledge engineering and toward prompt engineering, providing the model with useful, natural-language descriptions of knowledge (Marvin et al., 2024). This perspective gives rise to the vision of MetaWizard being taught new knowledge by domain experts via natural-language descriptions, facilitating application in domains that require highly specific knowledge, such as artisan crafts or high-precision industrial manufacturing.

The symbol grounding problem is another core challenge in program synthesis for which multimodal foundation models may present an answer. Web-scale multimodal datasets such as Open X-Embodiment (O'Neill et al., 2024) contain language-annotated scenes, comprising low-level joint or end-effector data, for a diverse range of robots, environments and application domains. Large VLA models can solve a wide range of cognitive tasks, such as object recognition, localization, navigation, or task planning through querying in natural language (Li et al., 2024). Using techniques such as RAG, multiple multimodal models can be combined to sophisticated mixture-of-expert systems via the shared medium of natural language. Large-scale multimodal networks have become state-of-the-art for visual question answering tasks (Liu et al., 2024b), indicating that they may constitute a powerful, general-purpose technology for grounding symbols in real-world percepts.

CHAPTER 5

A Framework for Neurosymbolic Robot Programming

The nature of robots as universal manipulators motivates an increasing number of organizations to employ robots for the production of goods or the provision of services. Many applications, such as the electronics assembly or shopping assistance scenarios discussed in Section 1.1.1, require the solution of complex real-world programming challenges, such as ensuring that process requirements are met over the lifetime of the production cell, or that the robot reliably reacts to changes in dynamic environments. Chapter 1 motivates the use of AI to automate the programming of complex robot behavior through learning and data-driven optimization, and motivates neurosymbolic programming as a solution to afford such functionality without sacrificing control of robot behavior.

Chapters 2, 3 and 4 introduce representations and algorithms for neurosymbolic robot programming. While providing solutions to programming problems such as program creation and optimization, organizations or individuals seeking to use them may not have the technical or organizational AI expertise or infrastructure for implementing them into technical solutions they can productively use. To enable robot programmers to apply the proposed algorithms and representations to real-world programming problems, this chapter integrates them into a coherent framework for neurosymbolic robot programming. Taking a lifecycle perspective, Section 5.1.1 proposes a neurosymbolic robot programming workflow comprising the initial program creation, optimization and deployment as well as program maintenance and lifelong re-optimization that integrates seamlessly into the robot program lifecycle common to most industrial and many service robotics applications. Section 5.1.2 outlines an XUI that aims at enabling both AI novices and experts to use neurosymbolic AI for robot program optimization. Section 5.1.4 introduces a technical framework that embeds the algorithms and data structures presented in this work in an industrial robot software ecosystem. Section 5.3 discusses the overall framework with particular focus on real-world service and industry applications. Section 5.4 charts avenues for further research.

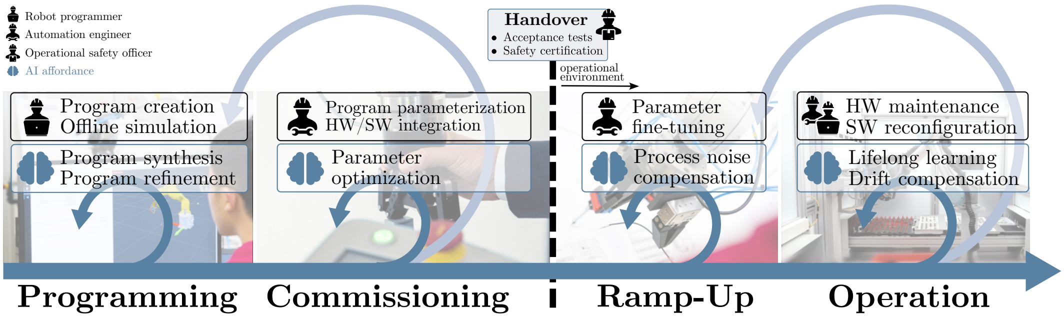


Figure 5.1: The five-step lifecycle of an industrial robot program, comprising human-program interactions (black) and corresponding affordances for AI assistance (blue) (Alt et al., 2024a).

5.1 A Framework for Neurosymbolic Robot Programming

NRPs bridge the representational divide between robot programs that explicitly and symbolically represent tasks for the purpose of interpretation and modification by human programmers, and predictive models of robot programs that afford learning and first-order optimization. Building on it, the SPI and MetaWizard families of algorithms provide AI assistance to robot programmers. To enable programmers to make use of them in real-world robot programming applications, they are integrated into a framework for neurosymbolic robot programming that prescribes both a workflow as well as a set of software tools comprising GUIs and APIs.

5.1.1 A Neurosymbolic Robot Programming Workflow

5.1.1.1 A Lifecycle Perspective on Robot Programming

To correspond to the reality of real-world robot programming applications, robot programming as bidirectional communication between programmer and robot introduced in Chapter 1 must not be conceptualized as a one-time interaction that begins with the programmer specifying the intended behavior and ends with the robot performing the programmed task. Rather, robot programming is a *repeated* interaction in which the program is created, executed, read and modified many times. The lifecycle of a robot program begins with its creation, by a human programmer or by a robot program synthesis system, and ends when it cedes being useful. In the interim, it may be executed many times, and changing environments or changing requirements may mean frequent, sometimes even continuous, changes to the program structure or parameterization.

Robot program lifecycle ▷

In Alt et al. (2024a), I propose a five-stage robot program lifecycle for industrial robot applications (see Figure 5.1):

1. **Programming:** The robot program is initially created. A suitable program structure, such as a sequence of hierarchically composed skills, is instantiated and parameterized to approximately solve the task at hand. In industrial practice, this is typically performed by a skilled robot programmer using a mixture of textual, teach-in and other software-assisted programming methods (see Section 2.6).

2. **Commissioning:** The robot program is deployed on the robot hardware, typically at a systems integrator or in a pre-production environment. This involves integration of additional sensors and actuators as well as external process control infrastructure such as programmable logic controllers (PLCs). The program structure may be adjusted by skilled engineers in response to preliminary tests. The program parameterization is adjusted in a process of trial and error until given requirements for process metrics such as cycle time, robustness or quality are met.

3. **Handover:** The programmed robot workcell undergoes safety certification and acceptance testing, typically by human auditors and engineers. Besides the fulfilment of process metrics and machinery safety regulations, adherence to company-wide or industry-specific robot programming standards is ensured. Upon passing of all required tests and inspections, the robot workcell is installed in a production environment.

4. **Ramp-Up:** Production begins, typically at a reduced pace, lower production quality or robustness. Due to differences between the pre-production and production environments, robot program parameters are incrementally adjusted to finetune the program to the operational environment and achieve the required level of performance. In current industrial practice, parameter adjustment is performed by skilled machine operators.

5. **Operation:** During operation, the program structure or, more typically, program parameters are subject to change in response to changing process requirements, workpiece variants, or changes to production hardware, e.g. when worn gripper fingers are replaced during routine maintenance. Program adjustments are typically made by skilled maintenance workers or machine operators.

Industrial robot programs undergo changes to their parameterization or program structure at every step of their lifecycle, implying a high potential for AI assistance. However, due to the high safety and performance requirements of industrial applications, robot programs must be readable and interpretable by humans, particularly in the handover and operation stages, when the fundamental safety of the program is first ascertained by human decisionmakers, and then must be guaranteed and potentially re-ascertained during the operation of the robot workcell,

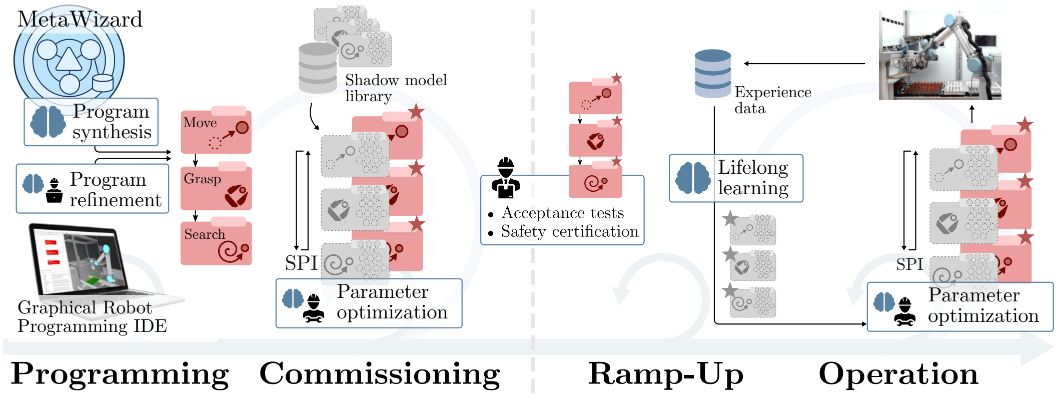


Figure 5.2: Bridging the AI Adoption Gap with Neurosymbolic AI (BANSAI) proposes a conceptual framework for AI-enabled robot programming that leverages neurosymbolic robot program synthesis and optimization algorithms for initial program creation and continuous re-optimization, without eschewing acceptance testing and auditing by human experts (Alt et al., 2024a).

even as changes to the robot hardware, program structure and parameterization are made.

The lifecycle of a robot program for a service application is very similar to that of an industrial robot program. As service robotics applications typically involve dynamic, possibly unstructured environments, the initial robot program must contain mechanisms for failure detection and recovery, giving greater importance to the initial programming stage. The other stages are similar to their industrial counterparts, except that the objective is not achievement of process metric objectives and production targets, but rather safe operation and robust task achievement in semi-structured or unstructured, dynamic environments.

5.1.1.2 AI-Assistance for the Complete Robot Programming Lifecycle

BANSAI ▷ In Alt et al. (2024a), my colleagues and I propose BANSAI (Bridging the AI Adoption Gap with Neurosymbolic AI), a conceptual framework that integrates the neurosymbolic representations and algorithms presented in the previous chapters into the robot program lifecycle. We note that the programming, commissioning, ramp-up and operation of robot programs involve human-program interactions that, in turn, correspond to affordances for AI assistance (see Figure 5.1 (blue)). Figure 5.2 provides an overview of the proposed workflow.

1. **Programming:** The initial robot source program is co-created by a human programmer and one of the MetaWizard program synthesis systems proposed in Chapter 4. Initial robot program parameters are inferred from structured knowledge (see Section 4.1), human-AI dialogue (see Section 4.2) or a generative DL model (see Section 4.3).

2. **Commissioning:** During commissioning, an initial dataset is created as the robot program is executed on the hardware with small perturbation to the program inputs (see Section 2.7.1). Given a pretrained library of shadow skills for commonly used robot skills such as searching, grasping or insertion, as well as synthetic pretraining data, the need for real-world data can be considerably reduced (see Section 3.2.1). An NRP shadow program for the source program is trained and robot program parameters are optimized via SPI (see Section 3.1.1) with respect to the relevant process metrics. If required, the robot source program can be modified by human programmers, as the source program remains interpretable and human-modifiable. If the program structure is changed significantly, training data may have to be collected again and parameters re-optimized using the updated shadow program.

3. **Handover:** One of the core advantages of the NRP representation is that handover does not change – the same auditing and acceptance testing mechanisms can remain in place. This is crucial for industrial applications, in which robot workcells (and the programs controlling them) must be certified to comply with safety regulations, industry and company-wide standards.

4. **Ramp-Up:** During ramp-up, additional training data is collected as production begins, and the shadow model is finetuned on the production environment (see Section 3.2.1). Robot program parameters are optimized to achieve the target process metrics in production. As during commissioning, the robot program can be modified by human programmers if required.

5. **Operation:** During operation, data can be passively collected as the robot program is continuously executed. Robot program parameters can be continuously re-optimized, using an updated shadow program (“lifelong learning”, see Section 3.2.2), to compensate e.g. workpiece variances (Alt et al., 2022b; Kienle et al., 2024), wear and tear (Raible et al., 2023b) or other nonstationary sources of noise, such as sudden shifts due to hardware replacements or repairs.

The algorithms and representations introduced in this work realize an AI-enabled robot programming workflow that seamlessly integrates into the process of creating, deploying and maintaining industrial robot-based production systems (Alt et al., 2024a). As the robot program lifecycle for service robotics applications is similar, the same workflow can be applied in service domains. BANSAI leverages the benefits of neurosymbolic programming – tractable learning, synthesis and optimization, while ensuring human interpretability and modifiability – to provide AI assistance functions that solve practical bottlenecks in robot programming. As a conceptual framework, BANSAI proposes a general-purpose process model for AI-enabled robot programming, similar in nature (albeit smaller in scope) to process models for AI engineering such as PAISE® (Hasterok and Stompe, 2022).

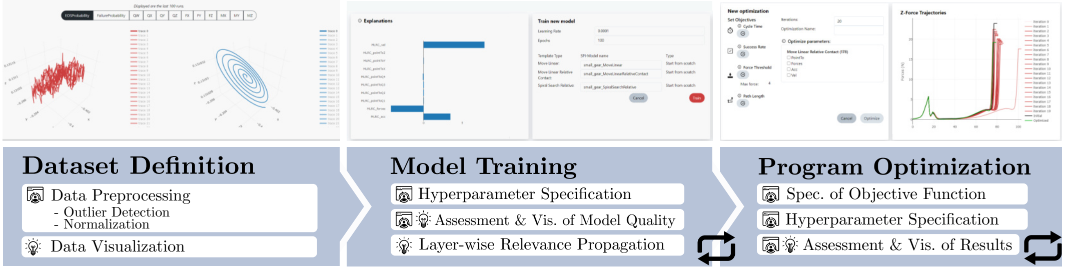

Dataset Definition
Data Preprocessing
- Outlier Detection
- Normalization
Data Visualization
Model Training
Hyperparameter Specification
Assessment & Vis. of Model Quality
Layer-wise Relevance Propagation
Program Optimization
Spec. of Objective Function
Hyperparameter Specification
Assessment & Vis. of Results

Figure 5.3: The SPI program optimization workflow, realized by a wizard-like XUI. At each step of the workflow, the XUI offers user-adaptive (🖼) or explainability (💡) features (Alt et al., 2024d).

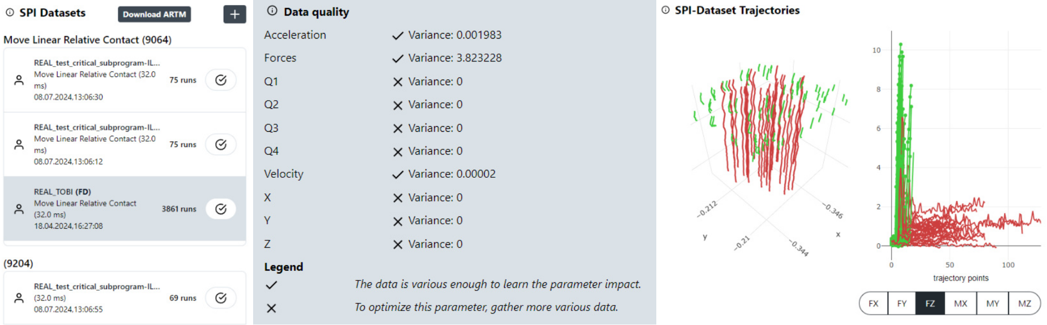
5.1.2 User Interfaces for Neurosymbolic Robot Programming

BANSAI ties the algorithms and representations proposed in this work into a coherent conceptual framework and workflow for user interaction. The application of BANSAI for practical programming problems hinges, like most AI solutions, on the ability of humans to interact with the AI in an intuitive manner (FakhrHosseini et al., 2024). Moreover, particularly in safety-critical domains such as robotics, AI adoption is contingent on fostering trust of human operators into the AI system (Theis et al., 2023). Explainability of AI methods, such as the “explainability of outcomes” afforded by the dual NRP representation and algorithms based on surrogate models such as SPI, is a central contributing factor to trust in AI (Theis et al., 2023; Chen, 2023; Agostinho et al., 2023). Another crucial factor is the availability of XUIs for humans to use and interact with explainable AI systems (Chromik and Butz, 2021; Bove et al., 2023; Füßl et al., 2024).

In Alt et al. (2024d), my colleagues and I present an XUI for SPI. It proposes a wizard-like user experience, that guides users step-by-step through the data collection, shadow program training and parameter optimization procedures, offering visualizations or explicit XAI functionalities at every step of the workflow. Figure 5.3 overview of the XUI from a user workflow perspective. The XUI organizes the parameter optimization process into three steps:

Dataset definition The collection and curation of high-quality datasets is crucial for successful deployment of any data-driven AI system (Peres et al., 2020). In application domains such as industrial robotics, data collection and curation is typically done by domain experts with little prior AI expertise, making it a crucial bottleneck and risk factor in AI deployment (Siaterlis et al., 2022). The proposed XUI addresses this issue by providing two dedicated sub-workflows for data collection and curation:

1. **Dataset collection:** In follow-up work to Alt et al. (2024d), we developed an automatic data collection framework that permits the human programmer to select the robot skill(s) or subprogram(s) in the source program whose



Move Linear Relative Contact (9064)

REAL_test_critical_subprogram-IL... Move Linear Relative Contact (32.0 ms) 08.07.2024,13:06:30 — 75 runs

REAL_test_critical_subprogram-IL... Move Linear Relative Contact (32.0 ms) 08.07.2024,13:06:12 — 75 runs

REAL_TOBI **(FD)** Move Linear Relative Contact (32.0 ms) 18.04.2024,16:27:08 — 3861 runs

(9204)

REAL_test_critical_subprogram-IL... (32.0 ms) 08.07.2024,13:06:55 — 69 runs

Acceleration	✓ Variance: 0.001983
Forces	✓ Variance: 3.823228
Q1	✕ Variance: 0
Q2	✕ Variance: 0
Q3	✕ Variance: 0
Q4	✕ Variance: 0
Velocity	✓ Variance: 0.00002
X	✕ Variance: 0
Y	✕ Variance: 0
Z	✕ Variance: 0

Legend

✓	*The data is various enough to learn the parameter impact.*
✕	*To optimize this parameter, gather more various data.*

Figure 5.4: User interface for dataset selection, estimates of data quality, and visualization of position (left) and force (right) trajectories in the dataset.

parameters are to be optimized (the *critical skill*).[1] Based on this selection, the *critical subprogram* is automatically determined via graph search on the source program DAG. The critical subprogram contains all skills or subprograms upstream of the critical skill for which shadow skills must be learned. This corresponds to the subgraph of all non-deterministic skills or subprograms preceding the critical skill, and is identified by traversing the topologically sorted program DAG in reverse order ("backward") from the critical skill, stopping at the first encountered deterministic skill (with a fixed end state). This permits the system to automatically identify all source skills for which shadow skills are to be trained with one single click by the user. The system then suggests intervals for input perturbation for each of these source skills, allows the user to manually adapt them e.g. to ensure safety limits, and automatically executes the source program a given amount of times on the robot, storing the resulting robot data.

2. **Dataset curation:** The XUI provides GUI elements for displaying the collected robot data as 2D or 3D trajectories (see Figure 5.3), as well as aggregated metrics for dataset quality (see Figure 5.4). As SPI iteratively optimizes program parameters, the shadow program is evaluated with different inputs over the course of an optimization, requiring the trained shadow skills to exhibit strong generalization. Their respective training datasets must, consequently, cover a sufficiently broad range of the input space. In the XUI, the user is warned if the variances along one or multiple input dimensions is very small.

Model training The training of ML models poses significant challenges to AI novices, as the choice of appropriate hyperparameters for training depends significantly on the application, the network architecture and the structure and quality of the training data (Bischl et al., 2023). Moreover, the task of evaluating the quality of a trained model is particularly challenging for AI novices (Yang et al., 2018; Yang

---

[1] Unpublished work. Credit to Cristian Gorun for the implementation.

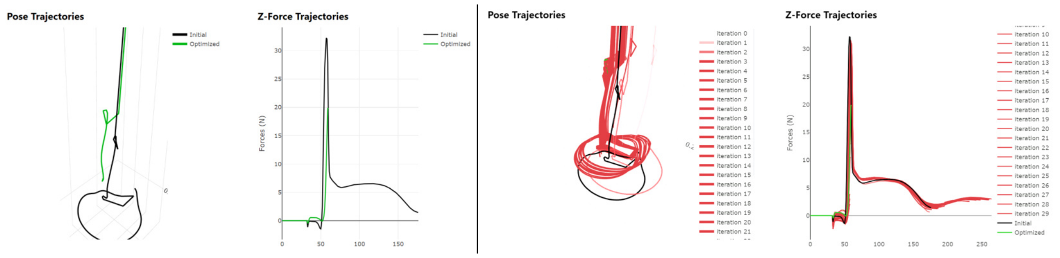


Figure 5.5: Visualization of predicted trajectories over the course of SPI prediction in guided (left) and expert modes (right).

et al., 2020a). The proposed XUI offers user-adaptive mechanisms for hyperparameter selection, leaving detailed hyperparameter selection to expert users and using sensible defaults for novices (see Section 5.1.2.1). After model training, model performance is visualized to the user by joint plotting of ground-truth test data, held out from the training dataset, and model predictions, as well as by displaying automatically computed, human-interpretable accuracy metrics (e.g. Cartesian MAE between predicted and ground-truth end-effector poses) (see Section 5.1.2.2). The user is warned if an error metric is below sensible defaults.

Program optimization In interviews with expert robot programmers conducted during the development and evaluation of SPI, it has been observed that the specification of the task objective Φ – the expression of domain-specific objectives, such as “the cycle time must be below 5s, and insertion forces may not exceed 5N”, in terms of a weighted sum of differentiable error terms – constitutes the largest challenge for practitioners using SPI. To that end, similar user adaptivity and plotting mechanisms are realized as for model training. One core innovation is the use of the learned shadow program as a surrogate of the real robot for evaluating optimized parameter sets. For a given optimization result, the user is instantly provided with a visualization of the corresponding posterior trajectory output by the shadow program (see Figure 5.5). By avoiding the delay incurred by executing the source program with optimized parameters on the real robot, parameter optimization becomes interactive, and users arrives at suitable parameter sets much more quickly than through trial and error on the real robot hardware.

5.1.2.1 User Assistance for AI Novices via User-Adaptive UI

Lack of AI expertise or data literacy (Leon-Urrutia et al., 2022) among industry practitioners is one of the most significant challenges in the deployment of AI systems (Li et al., 2021; Siaterlis et al., 2022; Arinez et al., 2020). To address this “skills gap” (Jaiswal et al., 2024) and enable AI novices to use AI-based robot program optimization, the proposed XUI offers two modes, a “guided” and an “expert” mode, that add or remove user interface elements, configuration options, visualizations and e.g. explanatory tooltips. One example is the specification of hyperparameters and objective functions for parameter optimization. In "expert" mode, users are able to specify the update rate, whether the GPU ought to be used, and custom weights for each selected component of the objective function. In "guided" mode, the optimizer uses predefined weights for the objective components, GPU-accelerated optimization is enabled and the update rate is set to a default value and subject to learning rate scheduling. I refer to Alt et al. (2024d) for additional details on user adaptivity.

5.1.2.2 Explainability and Visualization of Outcomes

Explainability is one of the central factors in fostering trust in AI systems among industry practitioners (Agostinho et al., 2023; Ferrario and Loi, 2022). The proposed XUI contains two types of explainability features, designed to provide users with an understanding of the data, model and optimizer behavior, as well as the resulting robot behavior. First, Layer-wise Relevance Propagation (LRP) is applied to trained models, computing the relative importance of model inputs (program parameters) to model predictions. The results are visualized as a bar chart and serve primarily as a "sanity check" on model performance, allowing domain experts to gauge whether model behavior roughly corresponds to their expectation of robot behavior over the training dataset. On a dataset containing motions with varying dynamics, for example, velocity or acceleration parameters should have comparatively higher LRP scores. Second, data, model predictions and predicted trajectories before and after optimization are plotted. The GUIs for model and optimization evaluation feature elements for interactive modification of input parameters and nearly-real-time visualization, allowing the user to explore and validate model or program behavior for different inputs or hyperparameters (see Figure 5.5). I refer to Alt et al. (2024d) for details.

5.1.3 Validation and User Study

The XUI, and the parts of BANSAI it covers, have been demonstrated on the gear assembly scenario described in Section 4.3.2.[2] The XUI has been validated in a preliminary user study with 12 experienced robot programmers with different levels of prior AI expertise. Given a predefined robot program and a given set of robot data, participants were asked to use the XUI to perform the complete SPI workflow, including model training. After performing the task, participants completed a questionnaire containing questions on the perceived utility of the XUI for each step of the workflow (e.g. "How useful was the interface to train models and evaluate their quality?"), about specific user-adaptive features (e.g. "How often did you feel assisted through provided textual guidance?") as well as the NASA TLX questionnaire (Hart, 2006) to assess cognitive load during the experiment.

---

[2] A video of the user interaction, as well as of the robot performing the task before and after program optimization, can be found at https://www.youtube.com/watch?v=NCnJKBTKYh4.

In addition, the questionnaire contained several questions specifically focusing on whether the user had a sense of understanding the system (e.g. "How well can you assess the system's capabilities?').

Participants who self-selected as AI novices completed the task in an average of 32 minutes, including the time required for model training. Survey results across both participant groups indicate high levels of perceived usefulness across all steps of the process, with the highest utility perceived at the parameter optimization step. The perceived cognitive load was low, and the responses indicate that both expert and non-expert users gained a sufficient level of understanding of the system. Both groups found it comparatively difficult to determine whether or not the system behaved correctly, and indicated that the transparency of the model can be further improved. One particularly interesting result of the preliminary study is that experts took longer to complete the task (43 minutes on average) than AI novices, and reported lower confidence in their own performance. Qualitative interviews after the experiment indicate that experts took additional time to explore the XUI out of curiosity, explaining the longer task completion time. I refer to Alt et al. (2024d) for a detailed account and analysis of the user study. Future studies of SPI and the proposed overall neurosymbolic robot programming framework will be designed to avoid such exploration effects, e.g. by imposing a time limit or rewarding participants for efficient task performance. Moreover, they will increase the number of participants to achieve statistical significance, and follow a 2x2 between-subject design with two independent variables: The level of explainability (high or low) and the level of control (high or low) given to the user. This study design would help disentangle the individual effects of explanations and user control on task performance, cognitive load, and user satisfaction.

5.1.4 A Software Framework for Neurosymbolic Robot Programming

Beyond the workflow and user experience (UX) aspects of AI-enabled robot programming, the real-world deployment and use of AI for programming robots requires technical infrastructure for data collection, storage and management, the training and management of ML models, as well as integration with the underlying robot hardware (Heimberger et al., 2023; Horvat and Heimberger, 2023; Uren and Edwards, 2023). The algorithms and data structures proposed in Chapters 2, 3 and 4, as well as the XUI described in this chapter are integrated into a coherent software framework for neurosymbolic robot programming, that combines commercial, established solutions for robot programming and data management with software modules and services encapsulating the algorithms and data structures developed in this work.

Software framework ▷ Figure 5.6 illustrates the proposed software framework. It consists of the following components:

1. **Robot IDE:** The ARTM industrial robot program representation is used as a NRP source program representation. ARTM programs can be manually composed, edited, simulated and executed in the ArtiMinds RPS industrial robot IDE. Beyond manual programming, the RPS provides Python and Hypertext Transfer Protocol (HTTP) APIs, allowing software systems to create, modify, simulate and execute ARTM programs and retrieve e.g. simulation results or program metadata.

2. **Robot experience database:** Robot experience data is crucial for training NRP shadow programs. As programs are executed, raw robot data such as timestamped joint angles, end-effector poses or force-torque readings are streamed to the ArtiMinds LAR robot data platform, which semantically annotates it with the corresponding metadata (e.g. which skill a given trajectory corresponds to) and stores it in a MariaDB relational database.

3. **Data management and curation platform:** ArtiMinds LAR provides user interfaces and HTTP APIs for data access and management. It provides a dataset abstraction as well as GUI elements for displaying data to human programmers, which are used by the program optimization frontend.

4. **AI backends:** The AI technologies proposed in the previous chapter have been implemented in a modular fashion. NRPs are implemented as PyTorch DCGs with additional tooling for automatic shadow program creation for the ARTM source program representation. SPI (see Section 3.1) as well as the DGPMP2-ND differentiable motion planner (see Sections 2.5, 3.3) are implemented as Python packages. SPI is deployed as a container on GPU-accelerated ML hardware and exposes a HTTP API to enable clients, such as the program optimization frontend, to train and optimize NRPs. The modular structure of MetaWizardLLM (see Section 4.3 permits the deployment of the required LLM and CAD grounding modules either on on-premise ML hardware or in a cloud environment. They are loosely coupled with the program synthesis frontend via a remote procedure call interface.

5. **Program synthesis frontend:** MetaWizardLLM provides a frontend for robot program synthesis, allowing user interaction via spoken natural language, written text and CAD visualization. It instantiates, parameterizes and triggers the simulation and execution of robot programs via the HTTP interface of the robot IDE.

6. **Program optimization frontend:** The frontend for program optimization (see Section 5.1.2) uses user interface components and HTTP API endpoints of the data management and curation platform, and integrates seamlessly into its GUI and user workflow.

The framework forms the technical equivalent to BANSAI, by seamlessly integrating neurosymbolic AI into a data-centric technology stack for robot program-

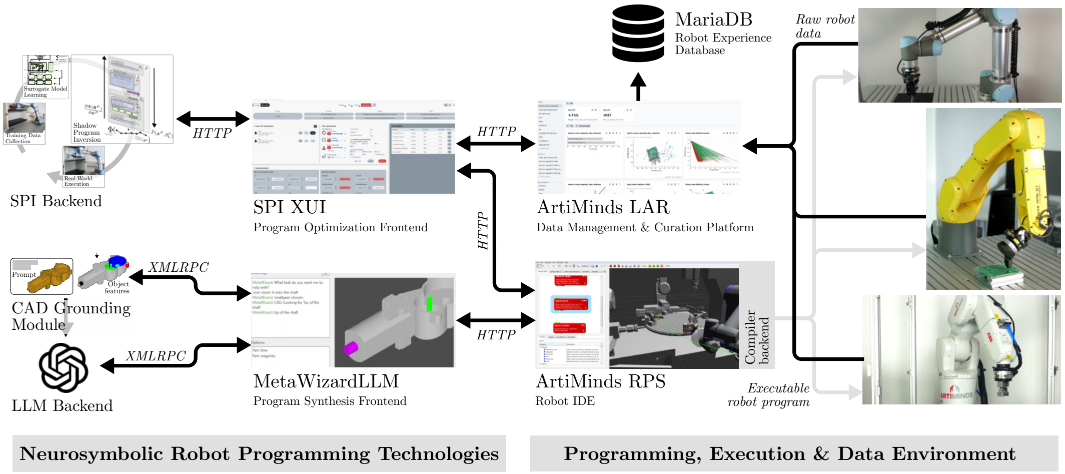


Figure 5.6: A software framework for AI-enabled robot programming, that integrates frontend and backend technologies for neurosymbolic robot programming with an industrial robot programming and data environment via loosely coupled interfaces.

ming. As BANSAI enables robot programmers to integrate AI assistance functions into their existing robot programming workflow, the software framework permits them to do so while continuing to use familiar software tools, communicating in industry-standard manners. AI-specific implementation details are hidden behind container-based deployment systems and well-defined, loosely coupled interfaces.

The framework has been used for the experiments in Chapters 3 and 4. While primarily developed and tested for use cases in industrial contexts, the framework can also be used in service contexts and has been evaluated on retail and household assistance applications (see Experiments 3.1.3.3, 3.3.2.1 and 4.1.7.2).

5.2 Related Work

5.2.0.1 Conceptual Frameworks for AI-Enabled Robot Programming

The complexity of creating, deploying and using AI methods in real-world application domains can be alleviated by conceptual frameworks that combine a range of technologies, data structures or algorithms into a coherent workflow to solve practical problems, such as robot programming.

Process-level frameworks In response to the complexity of real-world AI systems, and the organizational and infrastructural demands they place on organizations seeking to implement them, the Industry 4.0 (Plattform Industrie 4.0, 2019) and Industry 5.0 (Xu et al., 2021) initiatives aim to establish AI as an engineering discipline with standardized, general-purpose workflows, abstractions, roles and artifacts (Staron et al., 2024). At the time of writing, PAISE® (Hasterok and Stompe,

2022) constitutes the most comprehensive process model for AI systems engineering. It defines a workflow as well as corresponding roles and artifacts for the design, development and deployment of AI systems with a focus on ML methods. PAISE® relates elements from the ISO/IEC 15288 standard for system lifecycle processes (IEEE Standards Association, 2023) by technical details and specializations for AI systems engineering. Several related process models for AI systems engineering have since been proposed (Martínez-Fernández et al., 2022; Wu, 2024). Like BANSAI, PAISE® and related models consider the complete lifecycle of an AI system from conception to maintenance. Unlike BANSAI, they do not consider the use of AI systems to solve domain-specific problems such as robot programming, but focus on the development of the AI systems themselves. In surgical robotics, several conceptual frameworks and process models have been proposed, with the dual aims of making AI-assisted robotic surgery tractable, and to ensure that the resulting robot behavior correctly performs the intended surgical procedure at the required standard of care (Oleari et al., 2019; O'Sullivan et al., 2020; Prokhorenko et al., 2020; Marcus et al., 2024). Comparatively fewer conceptual frameworks with sufficient generality have been proposed for robot programming for general-purpose robotic manipulation. Hoebert et al. (2023) propose a framework for AI-assisted robot programming that is similar to BANSAI in scope. They propose an ontological representation of products, from which program synthesis systems can generate plans, which are grounded and executed by perception and execution modules. Along a similar vein, iRoPro (Liang et al., 2022) combines an interactive symbolic program synthesis system similar to MetaWizard with a lifecycle-centric workflow that permits users to modify or re-teach synthesized programs after they have been executed. On the subsymbolic side of the representational spectrum, ChatGPT for Robotics (Vemprala et al., 2023) proposes a robot programming workflow centering around the use of LLMs for high-level planning and robot code generation. They likewise consider programming as an iterative activity, that refines and modifies an explicitly represented robot program, albeit by using exclusively subsymbolic AI methods for planning and reasoning. BANSAI strikes a balance between the symbolic and subsymbolic paradigms of robot programming, proposing to use neurosymbolic AI methods and representations, and incorporating data-driven approaches for program optimization and lifelong learning.

Cognitive architectures Cognitive architectures are conceptual frameworks for creating, operating and understanding artificially intelligent systems. A cognitive architecture "mirrors the system architecture, using the power of abstraction to render the modeling, specification, and design of a complete complex system tractable" (Vernon, 2022). CRAM (Beetz et al., 2010; Beetz et al., 2023) is a cognitive architecture specifically designed to endow robots with the cognitive mechanisms required to understand human intents, generate actionable plans, execute them and reason about their own plans in light of acquired experience. The NEEM abstraction for robot experience data (Beetz et al., 2018; Olivares-

Alarcos et al., 2023) and the CPL (Beetz et al., 2023) provide general-purpose concepts for representing episodic knowledge and robot plans, which serve as a basis for downstream applications such as metacognition (Koralewski et al., 2019; Kazhoyan et al., 2020) or program synthesis (Ramirez-Amaro et al., 2014; Beßler et al., 2018; Nyga et al., 2018; Alt et al., 2023; Alt et al., 2024c). The ArmarX (Burghart et al., 2006; Peller-Konrad et al., 2023), ISAC (Kawamura et al., 2008) and CORTEX (Bustos et al., 2019) cognitive architectures provide conceptual frameworks and software support for high-level cognition and low-level control of humanoid robots. Trafton et al. (2013) propose ACT-R/E, an extension of the ACT-R cognitive architecture (Ritter et al., 2019), with the aim of realizing robot programming as bidirectional communication. ACT-R/E is not used as a framework for controlling robot behavior, but as a framework used by robots to understand, learn and accurately model human behavior and human cognition. BANSAI provides a dedicated model for the robot programming process, and integrates abstractions and representations from CRAM as the foundations of the MetaWizard and MetaWizard2 program synthesis systems. In contrast to cognitive architectures, which realize general-purpose mechanisms for robot cognition and often incorporate low-level mechanisms for perception or motor control, BANSAI places a dedicated focus on robot programming, and provides a workflow centered around the creation, optimization and maintenance of robot programs.

5.2.0.2 Software Frameworks for AI-Enabled Robot Programming

BANSAI is accompanied by a software framework for AI-enabled robot programming, that integrates the algorithms and representations proposed in this work into a software ecosystem for industrial robot programming. The software framework can be contextualized in light of related frameworks. The CRAM software ecosystem, comprising the KnowRob KR&R system (Tenorth and Beetz, 2013; Beetz et al., 2018), the SOMA ontology and extensions (Diab et al., 2019; Beßler et al., 2020; Beßler et al., 2021), the RoboSherlock (Beetz et al., 2015) and RoboKudo (Mania et al., 2024) perception systems, as well as a VR-based framework for simulation and human demonstration (Bozcuoğlu and Beetz, 2017; Haidu et al., 2018; Haidu and Beetz, 2021; Mania et al., 2021), provides a technical realization of the CRAM cognitive architecture. Similar software frameworks have been developed for other cognitive architectures (Laird et al., 2012; Wei and Hindriks, 2013; Vahrenkamp et al., 2015), providing tools for perception, planning, execution and other tasks. While realizing the model of robot programming prescribed by the respective cognitive architectures, robot programming frameworks built on top of cognitive architectures are not designed for real-world deployment and require considerable software engineering effort to use in production environments. Robot-specific software frameworks such as iCub-HRI (Fischer et al., 2018), or commercial manufacturer-specific robot programming frameworks such as the Franka AI Reference Platform (Haddadin et al., 2022) or the NEURA AI API and AI hub (NEURA

Robotics, 2024), provide AI technologies for PbD, reinforcement learning, or other AI-enabled robot programming methods. Manufacturer-independent commercial frameworks for AI-assisted robot programming such as the Micropsi MIRAI suite for programming vision-guided manipulation tasks (Micropsi Industries, 2024) or the Covariant Brain system for industrial bin picking (Covariant, 2024a) achieve high degrees of integration and ease of deployment, but eschew interpretability by directly controlling the robot. BANSAI and the software framework implementing it maintain the notion of programming as bidirectional communication with an explicitly represented program as the central artifact, ensuring the interpretability of robot behavior throughout the program's lifecycle.

5.3 Discussion

The framework for neurosymbolic robot programming presented in this work enables robot programmers to leverage neurosymbolic AI methods for creating, optimizing and maintaining robot programs. It addresses the crucial question of how powerful, data-driven capabilities for automatic program synthesis and optimization afforded by AI can be harnessed while retaining human control over robot behavior. Its contribution to the resolution of this research question also charts a path toward resolving the practical, open question of bringing state-of-the-art AI methods to industrial practice. Particularly in the manufacturing industries, there is an often-noted "gap" between the state of AI research and the degree to which AI is adopted in practical applications (Alt et al., 2024a; Jaiswal et al., 2024; Heimberger et al., 2024; FakhrHosseini et al., 2024; Uren and Edwards, 2023; Horvat and Heimberger, 2023; Siaterlis et al., 2022). Several reasons for this gap have been proposed. Three of the most commonly cited reasons are a lack of "AI readiness" at an organizational level, expressed as a lack of established processes and infrastructure for deployment and use of AI systems (Horvat and Heimberger, 2023; Heimberger et al., 2023), a lack of AI expertise at the level of individual users ("skills gap", Jaiswal et al. (2024) and Azmat et al. (2020)), as well as lack of trust in AI among users and decisionmakers (Theis et al., 2023; Schepman and Rodway, 2023). BANSAI contributes to bridging the AI adoption gap by addressing each of these reasons in turn.

Predefined workflows and software infrastructure BANSAI proposes a workflow for AI-enabled robot programming that extends the robot programming workflow currently prevalent in industry. It prescribes a concrete series of steps to create, parameterize and optimize robot programs using neurosymbolic methods and provides industry practitioners with a process template and a set of technologies to solve challenging robot programming problems. Like other engineering process models, BANSAI helps alleviate the uncertainty faced by organizations planning to use AI to program robots (Hasterok and Stompe, 2022). By extending and supplementing the conventional robot programming workflow, BANSAI imposes little additional overhead on organizations, and suggests concrete algorithms and data structures that implement it. The degree to which BANSAI facilitates AI adoption for robot programming at an organizational level is the subject of future research. Several components of BANSAI, such as SPI, DGPMP2-ND and parts of MetaWizard, are available open-source. Other components will be made available as part of a commercial robot programming suite (see Section 6.3). The use of a software framework at a comparatively high technology readiness level considerably reduces implementation and deployment costs, lowering the economic barriers to AI adoption (Cubric, 2020).

Intuitive user interfaces BANSAI embraces the role of the human programmer in the robot programming process, and proposes neurosymbolic technologies to *assist* humans in the programming of robots. BANSAI emphasizes the use of intuitive interaction modalities – an XUI for SPI, and natural-language dialogue or human demonstrations for MetaWizard. Intuitive human-machine interaction is an important prerequisite for the acceptance of AI by users (Sohn and Kwon, 2020). Consequently, the focus of BANSAI on intuitive human-machine interaction may facilitate AI adoption at the level of individual users, particularly for users with limited AI expertise (Long and Magerko, 2020).

Interpretability The interpretability of AI methods is a crucial factor for AI acceptance, partly by fostering trust in AI systems. BANSAI emphasizes interpretability in two ways. First, the use of NRPs as underlying robot program representation ensures that only interpretable source programs are executed by robots, and that neural representations are used only for synthesizing or optimizing interpretable programs. Ensuring the interpretability of the representation that ultimately encodes robot behavior is a prerequisite for widespread use in industry (Siaterlis et al., 2022; Agostinho et al., 2023; Theis et al., 2023; Heimberger et al., 2024) and removes a crucial barrier to AI adoption. Second, the XUI for SPI as well as the dialog-based user interaction principle of MetaWizard2 and MetaWizardLLM serve to make the behavior of AI-enabled programming systems more transparent to the user, by visualizing or verbalizing the behavior of the learned shadow model or the belief state of the program synthesis system. Increasing the transparency of AI methods has been identified as an important component of fostering trust (Edmonds et al., 2019; Kok and Soh, 2020; Ferrario and Loi, 2022; Leichtmann et al., 2023). Deriving causal explanations of model behavior, as well as computing and visualizing model uncertainty, is the subject of ongoing and future research.

5.4 Conclusion

This section presented a framework for neurosymbolic programming, comprising the BANSAI conceptual framework as well as a software framework for the AI-enabled creation, optimization, maintenance and deployment of robot programs. The framework comprises user interfaces and software infrastructure for the training and use of the algorithms and data structures presented in Chapters 3 and 4. BANSAI proposes a workflow for neurosymbolic programming that reflects the way robots are programmed in industry, while the software framework integrates neurosymbolic technologies for program synthesis and optimization into an industrial robot software ecosystem. Neurosymbolic robot programming is, however, not limited to industrial applications. The application of neurosymbolic robot programming in service robotics promises to address several practical challenges. Many use cases for service robots, such as everyday household assistance, are situated in contexts in which neither prior AI nor robotics infrastructure exists, and in which the "robot programmers" have neither robotics nor AI expertise. In such contexts, a framework that provides both a comprehensive workflow covering the complete robot program lifecycle, as well as comprehensive software solutions and infrastructure to realize it, is a crucial prerequisite to enable non-roboticists to deploy and use robotic assistants in homes, supermarkets or hospitals. The evaluation of neurosymbolic AI-enabled robot programming in service contexts is an important avenue for future work.

CHAPTER 6

Conclusion

6.1 Summary

This work investigates neurosymbolic programming as a paradigm that enables robots to capably solve complex manipulation tasks, while ensuring that human programmers remain in control of robot behavior. It proposes that the combination of subsymbolic, implicit representations such as neural networks with symbolic, explicit representations of programs or knowledge permits the realization of robot programming as bidirectional communication between robot programmer and robot, allowing the robot to automatically generate or optimize its behavior according to the programmer's intent through AI-enabled program synthesis or optimization, while at the same time allowing the robot programmer to read, understand and interact with the robot program. Neurosymbolic robot programming promises to enable AI assisted robot programming in domains in which training data is limited by enabling generalization through explicit, structured representations of knowledge, and the modularity afforded by symbolically composable program representations. Neurosymbolic robot programming also affords the use of AI methods in domains with high safety requirements, in which the ability for human programmers to understand the robot behavior resulting from a program is crucial both for physical safety as well as perceived trust.

Neurosymbolic robot programming hinges on bridging the representational divide between programs that afford learning and optimization, and programs that afford human interpretation and modification. To that end, a novel neurosymbolic robot program representation was introduced (see Chapter 2). NRPs are a dual robot program representation that associate a symbolic, skill-based robot program (the *source program*) with a differentiable surrogate (*shadow program*). The shadow program combines deep neural networks with differentiable planners to form a predictive model of the associated source program, which can be trained to reflect real-world robot behavior. The shadow DCG and the source program are related by structural and semantic equivalence, permitting the automatic construction of shadow programs for given source programs as well as the transfer of e.g. optimization results computed over the shadow program back into the source program representation. The DGPMP2-ND differentiable collision-free motion planner was introduced, which enables the creation of NRPs for complex robot programs involving both Cartesian- and configuration-space constraints. The dual nature of NRPs bridges the representational divide between symbolic and subsymbolic programs, permitting algorithms to leverage the benefits of either representation for e.g. program synthesis or optimization.

Chapter 3 introduced SPI, a first-order optimizer for NRPs. SPI optimizes robot program parameters by gradient descent over NRP shadow programs. Optimization is performed using the learned model of robot behavior, reflecting the real-world environment and robot dynamics. SPI optimizes robot program parameters with respect to differentiable, user-provided, task-specific objective functions. In combination with sequential transfer learning on simulated or real-world data, SPI has been shown to be highly data efficient and enables the lifelong optimization of robot program parameters in the face of nonstationary noise processes such as wear and tear, workpiece changes or sensor drift. By integrating the DGPMP2-ND differentiable motion planner, SPI-DP extends SPI to jointly optimize motion trajectories and robot program parameters.

Chapter 4 introduced the MetaWizard family of program synthesis systems, that leverage structured representations of knowledge in combination with subsymbolic methods to bootstrap NRPs via intuitive human-AI interaction. It explores three approaches, offering different user interaction paradigms or leveraging different internal representations and algorithms for reasoning and symbol grounding. MetaWizard accepts a human demonstration of a task in VR and generates an executable NRP source program by symbolic reasoning over a knowledge base, using semisymbolic or neural modules for grounding symbolic entities in physical reality. MetaWizard2 models tasks as realizations of a metatask model and dynamically grounds missing knowledge via natural-language dialogue with the human programmer. MetaWizardLLM retains the dialogue-based paradigm of interactive programming and leverages LLMs to bootstrap underspecified plans, which are grounded via RAG over CAD representations of the environment.

The proposed representations and algorithms have been designed with the aim of solving practical robot programming problems. To facilitate the application of neurosymbolic robot programming in real-world applications, Chapter 5 introduces a framework for AI-enabled robot programming. The BANSAI conceptual framework outlines how neurosymbolic data structures and algorithms can be applied to assist programmers in the creation, optimization, deployment and maintenance of robot programs in industrial and service applications. To enable users and organizations with limited AI experience and expertise to use neurosymbolic AI for the optimization of robot programs, an XUI is proposed, that guides users through the program learning and optimization workflow, while offering user-adaptive assistance functions and visual explanations. It is integrated into a software framework for neurosymbolic programming, that integrates the algorithms and data structures introduced in this work into an industrial robot programming ecosystem.

6.2 Discussion

This work investigated the central research question introduced in Chapter 1: **How can robots be programmed to tractably solve complex tasks in real-world environments, while leaving humans in control of robot behavior?** The proposed neurosymbolic robot programming framework proposes a technical answer to this question. The following paragraphs examine the research question from the perspectives of robot programming as an activity, an engineering discipline and a field of scientific inquiry.

6.2.0.1 Robot Programming as an Activity

Neurosymbolic programming affirms programming as bidirectional communication between robot and human programmer, while augmenting the capabilities of both the robot and the programmer via AI. It frees the human programmer from specifying the initial robot program at the motion level; instead, they can state their intent in natural language at the task level, and a robot program is inferred automatically to perform the task. They are freed from manually specifying and tweaking robot program parameters during ramp-up, deployment or maintenance, which are instead optimized by a data-driven, first-order optimizer. The robot's capabilities, on the other hand, are optimized by AI methods at the structural, parameter and motion level to better achieve the task through semantic understanding of user intent, AI-based grounding of underspecified plans, or iterative optimization over a learned model of the dynamics of the task, the environment, or itself. For the programmer, neurosymbolic programming shifts the activity of programming from the explicit manipulation of programs to the instruction and supervision of program-generating and program-optimizing AI algorithms, or, if AI and the robot are conceptualized as a unitary technical system, to the instruction and supervision of an *intelligent robot*.

The unique property of neurosymbolic robot programming as proposed in this work is that this activity remains bidirectional and interactive: The use of an explicitly represented, symbolic robot program allows human programmers to *understand* what the robot will do in response to an instruction, and to *modify* the robot's behavior by modifying the instructions or by directly modifying the robot program. From this perspective, neurosymbolic robot programming resembles traditional, symbolic programming in robotics and general-purpose computing. It allows for ancillary activities such as *debugging*, which involves repeated interaction with programs to gain a causal understanding of the system's behavior, enabling the detection of programmatic reasons for observed errors. Unlike implicitly represented programs, such as neural policies, NRP source programs can be debugged in a *white-box* manner, by reading and understanding the code that drives robot behavior. It also allows for established workflows and processes related to programming, such as auditing or certification, to remain unchanged, accelerating industry adoption. Interpretability is only ensured for the generated or modified robot program, however. Neither the program optimizers nor the program synthesis systems presented in this work are mechanistically interpretable. This satisfies the safety requirements of many applications, particularly in industrial robotics, where the robot program as an artefact is subject to scrutiny. In service applications, where program synthesis and optimization may be performed online to react to changes in the environment, interpretable metaprograms (program synthesizers and optimizers) would both increase trust by users and facilitate debugging and metaprogramming to correct and adapt the behavior of the neurosymbolic programming systems themselves. Increasing the interpretability of SPI and the MetaWizard program synthesis systems is a crucial avenue of further research.

6.2.0.2 Robot Programming as an Engineering Discipline

Programming is an *engineering discipline*, involving the “application of scientific knowledge to resolve conflicting constraints and requirements for problems of immediate, practical significance” (Shaw, 1990). In robotics, programming is concerned with applying knowledge about the domain, robot kinematics and dynamics, planning, perception, commonsense and other types of knowledge to solve practical, real-world problems by eliciting robot behavior – here, motions manipulating objects in the environment. Programming frameworks support engineers by providing representations such as programming languages, task models, and GUIs, as well as tools, such as compilers, planners, and simulators, that encapsulate general-purpose methodologies or knowledge, reducing the cognitive burden on the programmer: Effective innovations in programming raise the level of abstraction at which systems are programmed (Kramer, 2007). Neurosymbolic robot programming raises the level of abstraction for programming robots in three central ways.

First, it raises the representational level of abstraction: Programs are created via interactive dialogue in natural language or via PbD, and optimized in a largely automatic process controlled through a GUI. Natural-language interaction and high-level user input enable programmers to program more complex tasks at a given level of effort, or program tasks of a given complexity with a considerable reduction in effort.

Second, it raises the semantic level of abstraction, as program synthesis and optimization only require the programmer to state their intent in terms of task objectives or demonstrations, rather than prescribe a solution to the task at hand. Programming at a high semantic level alleviates the need for the programmer to solve highly domain-specific optimization problem of finding and specifying good robot behavior to solve the task, and permits them to focus instead on meaningfully describing the task. This permits domain experts to program robots without requiring robotics-specific knowledge and contributes to the democratization of robot programming.

Third, neurosymbolic robot programming shifts much of the engineering effort from programming to metaprogramming. Instead of parameterizing programs to solve real-world tasks, programmers parameterize program optimizers, and instead of encoding task knowledge in robot programs, programmers encode task knowledge explicitly (in KBs or prompts) or implicitly (in trained shadow programs) in the data structures that enable program synthesis and optimization. The central advantage of metaprogramming is that it must only be done once to cover a wide range of programming applications: Trained shadow programs afford optimization with respect to near-arbitrary task objectives, and a sufficiently large KB of tasks affords synthesis of robot programs for a wide range of user queries or environments. Neurosymbolic robot programming raises the level of abstraction of programming by encapsulating lower-level aspects of programming, such as code generation or iterative parameter optimization, to AI algorithms operating behind a shared, neurosymbolic representation. Programmers reap the full benefits of abstraction when this encapsulation is complete, i.e. both general, covering all tasks and use cases, and robust, always yielding expected behavior. Achieving the required degree of generality and robustness for seamless use in productive or everyday applications remains the focus of ongoing research and development.

6.2.0.3 Robot Programming as a Field of Inquiry

Programming has long been considered a scientific discipline (Dijkstra, 1977), and the development and study of programming systems has been a dynamic field of scientific inquiry. The central scientific contribution of this work is an exploration of the research question posed in Chapter 1 and again at the beginning of this section: How can robots be programmed to tractably solve complex tasks in real-world environments, while leaving humans in control of robot behavior?

One central thesis of this work is that bridging the representational divide between symbolic and subsymbolic representations facilitates the design and deployment of algorithms that afford powerful AI assistance, while representing robot programs in a way that is interpretable and modifiable by human programmers. The NRP representation, as well as the proposed algorithms for program optimization and program synthesis, propose an avenue toward resolving the seeming dichotomy between capability and control. At the same time, the investigation of neurosymbolic robot programming methods poses several new open research questions.

First, it remains to be investigated whether and to what degree the use of a symbolic intermediary between human and robot constrains the breadth of robot behavior that can be represented. Such a “safety tax” has been observed in explainable AI systems, which often trade representational power for mechanistic interpretability (Yampolskiy, 2022; Jensen et al., 2023; Dalrymple et al., 2024).

This work has shown the ARTM to be a suitable representation, particularly for industrial applications. Applications involving online control in highly dynamic environments may require different source program representations.

Second, there may be application contexts in which the interpretability of the source program – the program that is executed on the robot – may not suffice for safety. One such use case is lifelong learning (see Section 3.2.2), where programs are updated frequently, up to each program execution. The manual re-certification or re-auditing of the source program is impractical. Future research will investigate methods for automatically and provably ensuring the adherence of optimized or synthesized source programs to predefined specifications.

Third, the degree to which neurosymbolic robot programming scales to very large programs and highly complex, long-horizon tasks merits further, particularly empirical, investigation. While the deep GRUs used in shadow programs are, in principle, universal function approximators (Petrov et al., 2024), the performance of deep neural networks has empirically been shown to scale with the training duration, dataset size and network size following a power law with diminishing returns (Kaplan et al., 2020). This indicates that there may be a degree of task complexity at which the accurate prediction of robot behavior may become intractable, particularly with limited amounts of real-world training data. Ongoing research is investigating the use of multimodal Transformer architectures and pretrained foundation models for the prediction of complex visuotactile robot behavior (Kienle et al., 2024). Likewise, the use of pretrained LLMs for zero-shot generalization, data-efficient finetuning and RAG is studied in the context of improving the tractability of program synthesis for increasingly complex tasks (Alt et al., 2024b; Kienle et al., 2025).

The other central thesis of this work is that a conceptualization of AI-enabled robot programming as bidirectional interaction between robot and programmer, via the intermediary of an explicitly represented *program*, enables leveraging of neurosymbolic AI to solve real-world programming problems. It is given credence by the real-world experiments conducted in the context of this work. The experiments validating MetaWizard2 and MetaWizardLLM, in particular, provide qualitative evidence of the value of the role of the human programmer. In both experiments, the programmer not only specifies the task, but also validates the generated robot program before it is executed on the real robot. Particularly for MetaWizardLLM, the symbolic nature of the generated source program, and its resulting interpretability, acts as a crucial safeguard against possibly unsafe LLM outputs, fostering trust in the program synthesis system (see Section 4.3.2). Likewise, strengthening the bidirectional nature of programming by providing an XUI for SPI allowed AI novices to use sophisticated AI technologies productively to solve a practical program optimization task (see Section 5.1.2). The results obtained in this work echo related work on human-AI collaboration and human-in-the-loop systems (Grønsund and Aanestad, 2020; Wiethof and Bittner, 2021; Memmert and Bittner, 2022; Mosqueira-Rey et al., 2023), indicating that a central role for the human programmer in the robot programming process may improve both the tractability and safety of AI-enabled robot programming. BANSAI illustrates that neurosymbolic AI methods permit the design of AI-enabled robot programming workflows that seamlessly integrate into existing, human-centric workflows and infrastructure, facilitating the use of AI in production contexts. The empirical quantification of the benefit of using neurosymbolic AI in real-world field trials is a crucial area of future investigation. Another set of research questions raised by this work concern the degree to which neurosymbolic AI benefits human programmers, particularly in contrast to other AI methods. The preliminary user study conducted in the context of the XUI for SPI (Alt et al. (2024d), see Section 5.1.2) shows promising results. A larger, double-blind study with multiple user groups is required to discern the concrete benefits of AI assistance, interpretability, and user control on task performance, user satisfaction and trust in the AI system. Likewise, a large-scale study of the complete neurosymbolic AI framework on industrial and service robotics applications is required to quantify the extent to which the neurosymbolic workflow proposed by BANSAI and realized by the algorithms and data structures presented in this work tractably solves complex robot programming problems in real-world settings.

6.3 Outlook

The presented framework for neurosymbolic robot programming charts a path toward AI-enabled programming of robot manipulation tasks. The results presented in this work inspire future research and development to increase the breadth and complexity of the tasks that can be addressed, as well as to increase the level of technological maturity to enable its deployment and use in industry, household, retail and other real-world settings.

At the time of writing, NRPs and SPI are undergoing integration into a commercial robot programming environment. NRPs are integrated into the ArtiMinds RPS and LAR software suites as a "neural digital twin" of the ARTM robot program representation. It permits the use of available robot data on the LAR platform to train a physically accurate, predictive model of a robot program in execution. SPI has been patented (Alt et al., 2022a) and a version of SPI has been developed that runs on GPU-equipped cloud or on-premise servers as a service, and can be accessed by downstream applications via an API. In the research project EASY[1], SPI is being deployed in an edge-cloud context, in which model training is performed on cloud servers, while parameter and trajectory optimization are performed in the industrial edge or directly on the robot controller (Schultheis et al., 2024). In the research project VADER[2], SPI is integrated into the CatenaX industrial software

---

[1] Research project EASY (2022-2025), funded by the German Federal Ministry for Economic Affairs and Climate Action under grant #01MD22002B.

[2] Research project VADER (2023-2025), funded by the German Federal Ministry for Economic Affairs and Climate Action under grant #13IK026A.

ecosystem (Schöppenthau et al., 2023) as an on-demand service for data-driven program optimization.

A version of the XUI for SPI outlined in Section 5.1.2 has been integrated into the LAR web frontend, and permits users to optimize ARTM robot programs via SPI. Taken together, these developments form a commercial implementation of the program optimization aspects of BANSAI (see Section 5.1.1). It has been successfully validated in a first commercial project on an assembly use case in the automotive industry.

Beyond commercial development, the NRP program representation is the subject of ongoing research and future publications. To enable NRPs to represent a wider variety of complex tasks, multimodality is explored as a highly promising avenue of research. The incorporation of 2D or 3D vision information, video imagery or sound promise to enable NRPs to represent visuotactile manipulation tasks. The Multimodal Trajectory Transformer (MuTT) architecture extends the state representation by 2D images of the scene, and permits the prediction of robot trajectories in changing scenes without requiring finetuning (Kienle et al., 2024). Beside multimodality, a core focus of research on NRPs is placed on the use and finetuning of pretrained foundation models, that afford zero- or few-shot generalization for novel environments or task variants. The use of large models pretrained on very large datasets promises to allow NRPs shadow skills to represent skill executions in arbitrary environments, avoiding the need to finetune them for the concrete environment at hand. Beyond the use of pretrained models such as RT-2-X (O'Neill et al., 2024), the use of physics simulators such as Isaac Sim (NVIDIA, 2024a) for large-scale simulation of environments and task variants is explored. In addition, a promising line of research investigates the extension of NRPs to probabilistic models, such as Bayesian neural networks or variational autoencoders.

Like NRPs, SPI and SPI-DP are undergoing active research and development. A focus is placed on increasing the stability of the optimizer on the borders of the NRP's training data domain, to avoid oscillation or divergence as in the inputs are optimized beyond the range included in the training dataset. One promising approach involves regularizing the optimizer by injecting gradients that constrain the optimizer to the parameter region seen in the training data. The stable optimization of the branching conditions of NRPs with conditional branching ("`if`") remains an open research question. Future research will investigate the representation of task objectives that cannot be expressed as differentiable functions over the posterior trajectory in a straightforward manner, such as quality criteria defined in terms of the manipulated objects (e.g. a button having been pushed, or two objects having been stacked on top of each other). Applying the principle of neural surrogates to task objectives appears promising: Training a differentiable model, such as a neural network, to represent the task objectives and score the posterior trajectory, may allow SPI to optimize robot program parameters with respect to a wider range of task objectives. In a different line of work, SPI-DP is evaluated on a large array of complex tasks from the COLOSSEUM manipulation benchmark (Pumacay et al., 2024) to evaluate its performance in a wide variety of tasks and environments.

The MetaWizard family of robot program synthesis systems is likewise undergoing active research. The primary subject of inquiry is the use of LLMs for general-purpose reasoning as well as symbol grounding via RAG and other techniques (Kienle et al., 2025). To provide useful AI assistance for programming in complex service or industrial scenarios, future research must ascertain the degree to which pretrained LLMs can zero-shot generalize to novel tasks and environments via prompt engineering, and to what extent external knowledge sources or domain-specific finetuning are required. Future work will extend MetaWizardLLM to generate error-handling routines for robust, fault-tolerant manipulation. Likewise, alternative modes of user interaction beyond natural-language dialogue are explored, such as “autocompletion” of partially implemented program skeletons. Ultimately, the program synthesis systems proposed in this work may be integrated with parameter optimizers such as SPI to form a fully-fledged TAMP system, that jointly synthesizes and parameterizes robot programs for a given task. Using NRPs as a neural digital twin, such a system could evaluate candidate program structures and parameterizations via prospection (Vernon et al., 2015; Neumann et al., 2020; Picklum, 2024), without requiring real-world executions.

6.4 Conclusion

The central insight of this work is that the way programs are represented profoundly influences what “programming” is. Textual programs invite an immediate, interactive programming pattern in which the human programmer encodes the desired robot behavior by means of a programming language. Skill-based robot programming adds a layer of abstraction, transforming “programming” into the combination and parameterization of motion planners and perception routines. AI-enabled programming constitutes the next step on the ladder of abstraction, delegating complex *cognitive* programming tasks to AI systems. The vision of AI-enabled robot programming is that human programmers express *what* they want a robot to do, and the robot figures out *how* to fulfil the programmer’s intents on its own. It is tempting, therefore, to stop thinking about programs at all, and rather view robot behavior as an emergent phenomenon engendered by an AI algorithm. The search for explainable AI methods (Burkart and Huber, 2021) and the often-cited need for explainability particularly in safety-critical systems (Ferrario and Loi, 2022; Agostinho et al., 2023; Leichtmann et al., 2023), is evidence for the belief that there is value in having a mechanistic, if not causal, understanding of system behavior. As noted by Edsger Dijkstra, “it should be a trivial matter keep the happening evoked by one’s program firmly within one’s intellectual grip” (Dijkstra, 1971). As evidenced in this work, the choice and design of program representations has great bearing on the degree to which human programmers can understand the robot behavior evoked by artificially intelligent programming assistants. Given the complexity of real-world robotic manipulation, and the promise of AI to learn, reason about and solve complex manipulation problems, the search for better robot program representations is becoming an increasingly important and scientifically rewarding field of study.

Bibliography

Abelson, H., G. J. Sussman, and J. Sussman (July 1996). *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass.: The MIT Press. ISBN: 978-0-262-51087-5.

Abnar, S., M. Dehghani, B. Neyshabur, and H. Sedghi (Oct. 2021). “Exploring the Limits of Large Scale Pre-Training”. In: *International Conference on Learning Representations*.

Abu-Dakka, F. J., B. Nemec, J. A. Jørgensen, T. R. Savarimuthu, N. Krüger, and A. Ude (Aug. 2015). “Adaptation of Manipulation Skills in Physical Contact with the Environment to Reference Force Profiles”. In: *Autonomous Robots* 39.2, pp. 199–217. ISSN: 1573-7527. DOI: 10.1007/s10514-015-9435-2.

Adams, S., T. Cody, and P. A. Beling (Aug. 2022). “A Survey of Inverse Reinforcement Learning”. In: *Artificial Intelligence Review* 55.6, pp. 4307–4346. ISSN: 1573-7462. DOI: 10.1007/s10462-021-10108-x.

Agostinho, C., Z. Dikopoulou, E. Lavasa, K. Perakis, S. Pitsios, R. Branco, S. Reji, J. Hetterich, E. Biliri, F. Lampathaki, S. Rodríguez Del Rey, and V. Gkolemis (Dec. 2023). “Explainability as the Key Ingredient for AI Adoption in Industry 5.0 Settings”. In: *Frontiers in Artificial Intelligence* 6, p. 1264372. ISSN: 2624-8212. DOI: 10.3389/frai.2023.1264372.

Agrawal, P., R. Girshick, and J. Malik (2014). “Analyzing the Performance of Multilayer Neural Networks for Object Recognition”. In: *Computer Vision – ECCV 2014*. Ed. by D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars. Cham: Springer International Publishing, pp. 329–344. ISBN: 978-3-319-10584-0. DOI: 10.1007/978-3-319-10584-0_22.

Ahn, M., A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, A. Herzog, D. Ho, J. Hsu, J. Ibarz, B. Ichter, A. Irpan, E. Jang, R. J. Ruano, K. Jeffrey, S. Jesmonth, N. J. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, K.-H. Lee, S. Levine, Y. Lu, L. Luu, C. Parada, P. Pastor, J. Quiambao, K. Rao, J. Rettinghouse, D. Reyes, P. Sermanet, N. Sievers, C. Tan, A. Toshev, V. Vanhoucke, F. Xia, T. Xiao, P. Xu, S. Xu, M. Yan, and A. Zeng (Aug. 2022). “Do As I Can, Not As I Say: Grounding Language in Robotic Affordances”. In: *6th Annual Conference on Robot Learning*. arXiv. DOI: 10.48550/arXiv.2204.01691. arXiv: 2204.01691 [cs].

Ajaykumar, G., M. Steele, and C.-M. Huang (Oct. 2021). “A Survey on End-User Robot Programming”. In: *ACM Computing Surveys* 54.8, 164:1–164:36. ISSN: 0360-0300. DOI: 10.1145/3466819.

Akbulut, M. T., U. Bozdogan, A. Tekden, and E. Ugur (May 2021a). “Reward Conditioned Neural Movement Primitives for Population-Based Variational Policy Optimization”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10808–10814. DOI: 10.1109/ICRA48506.2021.9560897.

Akbulut, M., E. Oztop, M. Y. Seker, H. X, A. Tekden, and E. Ugur (Oct. 2021b). “ACNMP: Skill Transfer and Task Extrapolation through Learning from Demonstration and Reinforcement Learning via Representation Sharing”. In: *Proceedings of the 2020 Conference on Robot Learning*. PMLR, pp. 1896–1907.

Akcay, A. (Nov. 2016). *Daimler Integra Standards for Robotics*. Birmingham, UK.

Akrour, R., D. Sorokin, J. Peters, and G. Neumann (July 2017). “Local Bayesian Optimization of Motor Skills”. In: *International Conference on Machine Learning*. PMLR, pp. 41–50.

Alt, B., J. Dvorak, D. Katic, R. Jäkel, M. Beetz, and G. Lanza (Jan. 2024a). “BANSAI: Towards Bridging the AI Adoption Gap in Industrial Robotics with Neurosymbolic Programming”. In: *Procedia CIRP*. Vol. 130. Póvoa de Varzim, Portugal: Elsevier B.V., pp. 532–537. DOI: 10.1016/j.procir.2024.10.125. arXiv: 2404.13652 `[cs]`.

Alt, B., R. Jäkel, and D. Katic (Feb. 2022a). “Method and System for Determining Optimized Program Parameters for a Robot Program”. WO2022022784A1 (Karlsruhe).

Alt, B., D. Katic, R. Jäkel, and M. Beetz (Oct. 2022b). “Heuristic-Free Optimization of Force-Controlled Robot Search Strategies in Stochastic Environments”. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Kyoto, Japan: IEEE, pp. 8887–8893. ISBN: 978-1-6654-7927-1. DOI: 10.1109/IROS47612.2022.9982093.

Alt, B., D. Katic, R. Jäkel, A. K. Bozcuoglu, and M. Beetz (May 2021). “Robot Program Parameter Inference via Differentiable Shadow Program Inversion”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. Xi’an, China: IEEE, pp. 4672–4678. ISBN: 978-1-7281-9077-8. DOI: 10.1109/ICRA48506.2021.9561206.

Alt, B., F. K. Kenfack, A. Haidu, D. Katic, R. Jäkel, and M. Beetz (Sept. 2023). “Knowledge-Driven Robot Program Synthesis from Human VR Demonstrations”. In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*. Rhodes, Greece: IJCAI, pp. 34–43. ISBN: 978-1-956792-02-7. DOI: 10.24963/kr.2023/4.

Alt, B., U. Keßner, A. Taranovic, D. Katic, A. Hermann, R. Jäkel, and G. Neumann (Mar. 2024b). “Domain-Specific Fine-Tuning of Large Language Models for Interactive Robot Programming”. In: *European Robotics Forum 2024*. Ed. by C. Secchi and L. Marconi. Vol. 32. Springer Proceedings in Advanced Robotics. Rimini, Italy: Springer Nature Switzerland, pp. 274–279. ISBN: 978-3-031-76424-0. DOI: 10.1007/978-3-031-76424-0_49. arXiv: 2312.13905 [cs].

Alt, B., C. Kienle, D. Katic, R. Jäkel, and M. Beetz (May 2025). "Shadow Program Inversion with Differentiable Planning: A Framework for Unified Robot Program Parameter and Trajectory Optimization". In: *2025 IEEE International Conference on Robotics and Automation (ICRA)*. Atlanta, USA: IEEE. DOI: 10.48550/arXiv.2409.08678. arXiv: 2409.08678 [cs].

Alt, B., F. Stöckl, S. Müller, C. Braun, J. Raible, S. Alhasan, O. Rettig, L. Ringle, D. Katic, R. Jäkel, M. Beetz, M. Strand, and M. F. Huber (May 2024c). "RoboGrind: Intuitive and Interactive Surface Treatment with Industrial Robots". In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. Yokohama, Japan: IEEE, pp. 1–8. ISBN: 979-8-3503-8457-4. DOI: 10.1109/ICRA57147.2024.10611143. arXiv: 2402.16542 [cs].

Alt, B., J. Zahn, C. Kienle, J. Dvorak, M. May, D. Katic, R. Jäkel, T. Kopp, M. Beetz, and G. Lanza (Apr. 2024d). "Human-AI Interaction in Industrial Robotics: Design and Empirical Evaluation of a User Interface for Explainable AI-Based Robot Program Optimization". In: *Procedia CIRP*. Vol. 130. Póvoa de Varzim, Portugal: Elsevier B.V., pp. 591–596. DOI: 10.1016/j.procir.2024.10.134. arXiv: 2404.19349 [cs].

Alur, R., A. Radhakrishna, and A. Udupa (2017). "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by A. Legay and T. Margaria. Berlin, Heidelberg: Springer, pp. 319–336. ISBN: 978-3-662-54577-5. DOI: 10.1007/978-3-662-54577-5_18.

Alur, R., R. Singh, D. Fisman, and A. Solar-Lezama (Nov. 2018). "Search-Based Program Synthesis". In: *Communications of the ACM* 61.12, pp. 84–93. ISSN: 0001-0782. DOI: 10.1145/3208071.

Amos, B., I. Jimenez, J. Sacks, B. Boots, and J. Z. Kolter (2018). "Differentiable MPC for End-to-end Planning and Control". In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc.

Andrychowicz, M., M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. de Freitas (Dec. 2016). "Learning to Learn by Gradient Descent by Gradient Descent". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., pp. 3988–3996. ISBN: 978-1-5108-3881-9.

Ardizzone, L., J. Kruse, C. Rother, and U. Köthe (Sept. 2018). "Analyzing Inverse Problems with Invertible Neural Networks". In: *International Conference on Learning Representations*.

Arenas, M. G., T. Xiao, S. Singh, V. Jain, A. Ren, Q. Vuong, J. Varley, A. Herzog, I. Leal, S. Kirmani, M. Prats, D. Sadigh, V. Sindhwani, K. Rao, J. Liang, and A. Zeng (May 2024). "How to Prompt Your Robot: A PromptBook for Manipulation Skills with Code as Policies". In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4340–4348. DOI: 10.1109/ICRA57147.2024.10610784.

Arinez, J. F., Q. Chang, R. X. Gao, C. Xu, and J. Zhang (Aug. 2020). “Artificial Intelligence in Advanced Manufacturing: Current Status and Future Outlook”. In: *Journal of Manufacturing Science and Engineering* 142.11. ISSN: 1087-1357. DOI: 10.1115/1.4047855.

Asfour, T., M. Waechter, L. Kaul, S. Rader, P. Weiner, S. Ottenhaus, R. Grimm, Y. Zhou, M. Grotz, and F. Paus (Dec. 2019). “ARMAR-6: A High-Performance Humanoid for Human-Robot Collaboration in Real-World Scenarios”. In: *IEEE Robotics & Automation Magazine* 26.4, pp. 108–121. ISSN: 1558-223X. DOI: 10.1109/MRA.2019.2941246.

Ashwani, S., K. Hegde, N. R. Mannuru, M. Jindal, D. S. Sengar, K. C. R. Kathala, D. Banga, V. Jain, and A. Chadha (Apr. 2024). *Cause and Effect: Can Large Language Models Truly Understand Causality?* DOI: 10.48550/arXiv.2402.18139. arXiv: 2402.18139 [cs].

Azmat, F., B. Ahmed, W. Colombo, and R. Harrison (June 2020). “Closing the Skills Gap in the Era of Industrial Digitalisation”. In: *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. Vol. 1, pp. 365–370. DOI: 10.1109/ICPS48405.2020.9274788.

Bagnell, J. A., F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, M. Pivtoraiko, J.-S. Valois, and R. Zhu (Oct. 2012). “An Integrated System for Autonomous Robotics Manipulation”. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2955–2962. DOI: 10.1109/IROS.2012.6385888.

Bahl, S., M. Mukadam, A. Gupta, and D. Pathak (2020). “Neural Dynamic Policies for End-to-End Sensorimotor Learning”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 5058–5069.

Baig, M. S., A. Imran, A. Yasin, A. H. Butt, and M. I. Khan (Feb. 2022). “Natural Language to SQL Queries: A Review”. In: *International Journal of Innovations in Science & Technology* 4.1, pp. 147–162. ISSN: 2709-6130.

Bansak, K. C., E. Paulson, and D. Rothenhaeusler (Apr. 2024). “Learning Under Random Distributional Shifts”. In: *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 3943–3951.

Barraquand, J. and J.-C. Latombe (Dec. 1991). “Robot Motion Planning: A Distributed Representation Approach”. In: *The International Journal of Robotics Research* 10.6, pp. 628–649. ISSN: 0278-3649. DOI: 10.1177/027836499101000604.

Bates, T., K. Ramirez-Amaro, T. Inamura, and G. Cheng (Sept. 2017). “On-Line Simultaneous Learning and Recognition of Everyday Activities from Virtual Reality Performances”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3510–3515. DOI: 10.1109/IROS.2017.8206193.

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (2018). “Automatic Differentiation in Machine Learning: A Survey”. In: *Journal of Machine Learning Research* 18.153, pp. 1–43. ISSN: 1533-7928.

Beck, A. (Sept. 2017). *First-Order Methods in Optimization*. Philadelphia, PA, USA: SIAM-Society for Industrial and Applied Mathematics. ISBN: 978-1-61197-498-0.

Beetz, M., F. Bálint-Benczédi, N. Blodow, D. Nyga, T. Wiedemeyer, and Z.-C. Márton (May 2015). "RoboSherlock: Unstructured Information Processing for Robot Perception". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1549–1556. DOI: 10.1109/ICRA.2015.7139395.

Beetz, M., D. Bessler, A. Haidu, M. Pomarlan, A. K. Bozcuoglu, and G. Bartels (May 2018). "KnowRob 2.0 - A 2nd Generation Knowledge Processing Framework for Cognition-Enabled Robotic Agents". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 512–519. ISBN: 978-1-5386-3081-5. DOI: 10.1109/ICRA.2018.8460964.

Beetz, M., G. Kazhoyan, and D. Vernon (Apr. 2023). *The CRAM Cognitive Architecture for Robot Manipulation in Everyday Activities*. DOI: 10.48550/arXiv.2304.14119. arXiv: 2304.14119 [cs].

Beetz, M., L. Mösenlechner, and M. Tenorth (Oct. 2010). "CRAM — A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1012–1017. DOI: 10.1109/IROS.2010.5650146.

Behl, H. S., A. G. Baydin, R. Gal, P. H. S. Torr, and V. Vineet (Aug. 2020). "AutoSimulate: (Quickly) Learning Synthetic Data Generation". In: *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXII*. Berlin, Heidelberg: Springer-Verlag, pp. 255–271. ISBN: 978-3-030-58541-9. DOI: 10.1007/978-3-030-58542-6_16.

Bellini-Leite, S. C. (Aug. 2024). "Dual Process Theory for Large Language Models: An Overview of Using Psychology to Address Hallucination and Reliability Issues". In: *Adaptive Behavior* 32.4, pp. 329–343. ISSN: 1059-7123. DOI: 10.1177/10597123231206604.

Bereska, L. and S. Gavves (Apr. 2024). "Mechanistic Interpretability for AI Safety - A Review". In: *Transactions on Machine Learning Research*. ISSN: 2835-8856.

Berkenkamp, F., A. Krause, and A. P. Schoellig (Oct. 2023). "Bayesian Optimization with Safety Constraints: Safe and Automatic Parameter Tuning in Robotics". In: *Machine Learning* 112.10, pp. 3713–3747. ISSN: 1573-0565. DOI: 10.1007/s10994-021-06019-1.

Beschi, S., D. Fogli, and F. Tampalini (2019). "CAPIRCI: A Multi-modal System for Collaborative Robot Programming". In: *End-User Development*. Ed. by A. Malizia, S. Valtolina, A. Morch, A. Serrano, and A. Stratton. Cham: Springer International Publishing, pp. 51–66. ISBN: 978-3-030-24781-2. DOI: 10.1007/978-3-030-24781-2_4.

Beßler, D. (Sept. 2022). "Ontological Representation of Activity Context for Flexible Robot Task Execution". PhD thesis. Bremen, Germany: University of Bremen.

Beßler, D., M. Pomarlan, and M. Beetz (July 2018). "OWL-enabled Assembly Planning for Robotic Agents". In: *Proceedings of the 17th International Conference on*

Autonomous Agents and MultiAgent Systems. AAMAS ’18. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp. 1684–1692.

Beßler, D., R. Porzel, M. Pomarlan, M. Beetz, R. Malaka, and J. Bateman (2020). “A Formal Model of Affordances for Flexible Robotic Task Execution”. In: *ECAI 2020*, pp. 2425–2432. DOI: 10.3233/FAIA200374.

Beßler, D., R. Porzel, M. Pomarlan, A. Vyas, S. Höffner, M. Beetz, R. Malaka, and J. Bateman (2021). “Foundations of the Socio-Physical Model of Activities (SOMA) for Autonomous Robotic Agents&lt;Span Ref-Type=&quot;Fn&quot; Rid=&quot;FAIA210379_fn001&quot; Style=&quot;Display:None&quot;&gt; &lt;Sup&gt;1&lt;/Sup&gt; &lt;/Span&gt;” in: *Formal Ontology in Information Systems*. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 159–174. DOI: 10.3233/FAIA210379.

Betker, J., G. Goh, L. Jing, T. Brooks, J. Wang, L. Li, L. Ouyang, J. Zhuang, J. Lee, Y. Guo, W. Manassra, P. Dhariwal, C. Chu, Y. Jiao, and A. Ramesh (2023). “Improving Image Generation with Better Captions”. In.

Bhardwaj, M., B. Boots, and M. Mukadam (May 2020). “Differentiable Gaussian Process Motion Planning”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10598–10604. DOI: 10.1109/ICRA40945.2020.9197260.

Bhat, N. (June 2020). *Pyttsx3*.

Billard, A., S. Calinon, R. Dillmann, and S. Schaal (2008). “Robot Programming by Demonstration”. In: *Springer Handbook of Robotics*. Ed. by B. Siciliano and O. Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1371–1394. ISBN: 978-3-540-23957-4 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_60.

Biloš, M., J. Sommer, S. S. Rangapuram, T. Januschowski, and S. Günnemann (Nov. 2021). “Neural Flows: Efficient Alternative to Neural ODEs”. In: *Advances in Neural Information Processing Systems*.

Bischl, B., M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, D. Deng, and M. Lindauer (2023). “Hyperparameter Optimization: Foundations, Algorithms, Best Practices, and Open Challenges”. In: *WIREs Data Mining and Knowledge Discovery* 13.2, e1484. ISSN: 1942-4795. DOI: 10.1002/widm.1484.

Blank, D. S., L. A. Meeden, and J. B. Marshall (1992). “Exploring the Symbolic/Subsymbolic Continuum: A Case Study of RAAM”. In: *The Symbolic and Connectionist Paradigms: Closing the Gap*. The Cognitive Science Series: Technical Monographs and Edited Collection. Hillsdale, NJ, US: Lawrence Erlbaum Associates, Inc, pp. 113–148. ISBN: 978-0-8058-1079-0 978-0-8058-1080-6.

Blattmann, A., R. Rombach, H. Ling, T. Dockhorn, S. W. Kim, S. Fidler, and K. Kreis (2023). “Align Your Latents: High-Resolution Video Synthesis With Latent Diffusion Models”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 22563–22575.

Bodenstedt, S., M. Wagner, B. P. Müller-Stich, J. Weitz, and S. Speidel (Dec. 2020). "Artificial Intelligence-Assisted Surgery: Potential and Challenges". In: *Visceral Medicine* 36.6, pp. 450–455. ISSN: 2297-4725. DOI: 10.1159/000511351.

Bonatti, R., S. Vemprala, S. Ma, F. V. Frujeri, S. Chen, and A. Kapoor (Nov. 2022). "PACT: Perception-Action Causal Transformer for Autoregressive Robotics Pretraining". In: *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Bostrom, N. (2017). "Strategic Implications of Openness in AI Development". In: *Global Policy* 8.2, pp. 135–148. ISSN: 1758-5899. DOI: 10.1111/1758-5899.12403.

Bousmalis, K., G. Vezzani, D. Rao, C. M. Devin, A. X. Lee, M. B. Villalonga, T. Davchev, Y. Zhou, A. Gupta, A. Raju, A. Laurens, C. Fantacci, V. Dalibard, M. Zambelli, M. F. Martins, R. Pevceviciute, M. Blokzijl, M. Denil, N. Batchelor, T. Lampe, E. Parisotto, K. Zolna, S. Reed, S. G. Colmenarejo, J. Scholz, A. Abdolmaleki, O. Groth, J.-B. Regli, O. Sushkov, T. Rothörl, J. E. Chen, Y. Aytar, D. Barker, J. Ortiz, M. Riedmiller, J. T. Springenberg, R. Hadsell, F. Nori, and N. Heess (Sept. 2023). "RoboCat: A Self-Improving Generalist Agent for Robotic Manipulation". In: *Transactions on Machine Learning Research*. ISSN: 2835-8856.

Bove, C., M.-J. Lesot, C. A. Tijus, and M. Detyniecki (Mar. 2023). "Investigating the Intelligibility of Plural Counterfactual Examples for Non-Expert Users: An Explanation User Interface Proposition and User Study". In: *Proceedings of the 28th International Conference on Intelligent User Interfaces*. IUI '23. New York, NY, USA: Association for Computing Machinery, pp. 188–203. ISBN: 979-8-4007-0106-1. DOI: 10.1145/3581641.3584082.

Bowman, S. R., J. Hyun, E. Perez, E. Chen, C. Pettit, S. Heiner, K. Lukošiūtė, A. Askell, A. Jones, A. Chen, A. Goldie, A. Mirhoseini, C. McKinnon, C. Olah, D. Amodei, D. Amodei, D. Drain, D. Li, E. Tran-Johnson, J. Kernion, J. Kerr, J. Mueller, J. Ladish, J. Landau, K. Ndousse, L. Lovitt, N. Elhage, N. Schiefer, N. Joseph, N. Mercado, N. DasSarma, R. Larson, S. McCandlish, S. Kundu, S. Johnston, S. Kravec, S. E. Showk, S. Fort, T. Telleen-Lawton, T. Brown, T. Henighan, T. Hume, Y. Bai, Z. Hatfield-Dodds, B. Mann, and J. Kaplan (Nov. 2022). *Measuring Progress on Scalable Oversight for Large Language Models*. arXiv: 2211.03540.

Boyd, S., N. Parikh, E. Chu, B. Peleato, and J. Eckstein (Jan. 2011). "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends® in Machine Learning* 3.1, pp. 1–122. ISSN: 1935-8237. DOI: 10.1561/2200000016.

Bozcuoğlu, A. K. and M. Beetz (May 2017). "A Cloud Service for Robotic Mental Simulations". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2653–2658. DOI: 10.1109/ICRA.2017.7989309.

Brockett, R. W. (1984). "Robotic Manipulators and the Product of Exponentials Formula". In: *Mathematical Theory of Networks and Systems*. Ed. by P. A. Fuhrmann. Berlin, Heidelberg: Springer, pp. 120–129. ISBN: 978-3-540-38826-5. DOI: 10.1007/BFb0031048.

Brooks, R. A. (June 1990). “Elephants Don’t Play Chess”. In: *Robotics and Autonomous Systems*. Designing Autonomous Agents 6.1, pp. 3–15. ISSN: 0921-8890. DOI: 10.1016/S0921-8890(05)80025-9.

— (Mar. 2019). *A Better Lesson*.

Brosset, P., S. Patsko, A. Khadikar, A.-L. Thieullent, J. Buvat, Y. Khemka, and A. Jain (2019). *Scaling AI in Manufacturing: A Practitioner’s Perspective*. Tech. rep. Capgemini Research Institute.

Bruyninckx, H. and J. De Schutter (Aug. 1996). “Specification of Force-Controlled Actions in the "Task Frame Formalism" - A Synthesis”. In: *IEEE Transactions on Robotics and Automation* 12.4, pp. 581–589. ISSN: 2374-958X. DOI: 10.1109/70.508440.

Buchina, N., S. Kamel, and E. Barakova (Aug. 2016). “Design and Evaluation of an End-User Friendly Tool for Robot Programming”. In: *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pp. 185–191. DOI: 10.1109/ROMAN.2016.7745109.

Buchina, N. G., P. Sterkenburg, T. Lourens, and E. I. Barakova (Oct. 2019). “Natural Language Interface for Programming Sensory-Enabled Scenarios for Human-Robot Interaction”. In: *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pp. 1–8. DOI: 10.1109/RO-MAN46459.2019.8956248.

Bull, A. D. (2011). “Convergence Rates of Efficient Global Optimization Algorithms”. In: *Journal of Machine Learning Research* 12.88, pp. 2879–2904. ISSN: 1533-7928.

Burghart, C., R. Mikut, R. Stiefelhagen, T. Asfour, H. Holzapfel, P. Steinhaus, and R. Dillmann (Jan. 2006). “A Cognitive Architecture for a Humanoid Robot: A First Approach”. In: *5th IEEE-RAS International Conference on Humanoid Robots*. Vol. 2005. Tsukuba, Japan: IEEE, pp. 357–362. ISBN: 978-0-7803-9320-2. DOI: 10.1109/ICHR.2005.1573593.

Burkart, N. and M. F. Huber (May 2021). “A Survey on the Explainability of Supervised Machine Learning”. In: *Journal of Artificial Intelligence Research* 70, pp. 245–317. ISSN: 1076-9757. DOI: 10.1613/jair.1.12228.

Buss, S. (May 2004). “Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares Methods”. In: *IEEE Transactions in Robotics and Automation* 17.

Bustos, P., L. Manso, A. Bandera, J. Bandera, I. García-Varea, and J. Martínez-Gómez (June 2019). “The CORTEX Cognitive Robotics Architecture: Use Cases”. In: *Cogn. Syst. Res.* 55.C, pp. 107–123. ISSN: 1389-0417. DOI: 10.1016/j.cogsys.2019.01.003.

Cakmak, M. and A. L. Thomaz (Mar. 2012). “Designing Robot Learners That Ask Good Questions”. In: *2012 7th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 17–24. DOI: 10.1145/2157689.2157693.

Calandra, R., A. Seyfarth, J. Peters, and M. P. Deisenroth (May 2014). “An Experimental Comparison of Bayesian Optimization for Bipedal Locomotion”. In: *2014*

IEEE International Conference on Robotics and Automation (ICRA), pp. 1951–1958. DOI: 10.1109/ICRA.2014.6907117.

Chalkidis, I., M. Fergadiotis, P. Malakasiotis, N. Aletras, and I. Androutsopoulos (Nov. 2020). "LEGAL-BERT: The Muppets Straight out of Law School". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by T. Cohn, Y. He, and Y. Liu. Online: Association for Computational Linguistics, pp. 2898–2904. DOI: 10.18653/v1/2020.findings-emnlp.261.

Chang, W., D. Kwon, and J. Choi (Mar. 2024). "Understanding Distributed Representations of Concepts in Deep Neural Networks without Supervision". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.10, pp. 11212–11220. ISSN: 2374-3468. DOI: 10.1609/aaai.v38i10.28999.

Chao, Y. R. (Apr. 1968). *Language and Symbolic Systems*. Reissue edition. Cambridge: Cambridge University Press. ISBN: 978-0-521-09457-3.

Chaudhuri, S., K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, and Y. Yue (Dec. 2021). "Neurosymbolic Programming". In: *Foundations and Trends® in Programming Languages* 7.3, pp. 158–243. ISSN: 2325-1107, 2325-1131. DOI: 10.1561/2500000049.

Chen, J., H. Lin, X. Han, and L. Sun (Mar. 2024). "Benchmarking Large Language Models in Retrieval-Augmented Generation". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.16, pp. 17754–17762. ISSN: 2374-3468. DOI: 10.1609/aaai.v38i16.29728.

Chen, J.-T. and C.-M. Huang (2023). "Forgetful Large Language Models: Lessons Learned from Using LLMs in Robot Programming". In: *Proceedings of the AAAI Symposium Series* 2.1, pp. 508–513. ISSN: 2994-4317. DOI: 10.1609/aaaiss.v2i1.27721.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba (July 2021). *Evaluating Large Language Models Trained on Code*. DOI: 10.48550/arXiv.2107.03374. arXiv: 2107.03374 [cs].

Chen, R. T. Q., Y. Rubanova, J. Bettencourt, and D. K. Duvenaud (2018). "Neural Ordinary Differential Equations". In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc.

Chen, T., Q. Wang, Z. Dong, L. Shen, and X. Peng (Dec. 2023). "Enhancing Robot Program Synthesis Through Environmental Context". In: *Advances in Neural Information Processing Systems* 36, pp. 3881–3893.

Chen, T.-C. T. (Mar. 2023). *Explainable Artificial Intelligence (XAI) in Manufacturing: Methodology, Tools, and Applications*. Springer Nature. ISBN: 978-3-031-27961-4.

Chen, X., J. Ji, J. Jiang, G. Jin, F. Wang, and J. Xie (May 2010). “Developing High-Level Cognitive Functions for Service Robots”. In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*. AAMAS ’10. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp. 989–996. ISBN: 978-0-9826571-1-9.

Cheng, C.-A., M. Mukadam, J. Issac, S. Birchfield, D. Fox, B. Boots, and N. Ratliff (Apr. 2019). “RMPflow: A Computational Graph for Automatic Motion Policy Generation”. In: *arXiv:1811.07049 [cs]*. arXiv: `1811.07049 [cs]`.

Cheng, Y., L. Sun, and M. Tomizuka (Apr. 2021). “Human-Aware Robot Task Planning Based on a Hierarchical Task Model”. In: *IEEE Robotics and Automation Letters* 6.2, pp. 1136–1143. ISSN: 2377-3766. DOI: 10.1109/LRA.2021.3056370.

Chernova, S. and M. Veloso (Sept. 2004). “An Evolutionary Approach to Gait Learning for Four-Legged Robots”. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3, 2562–2567 vol.3. DOI: 10.1109/IROS.2004.1389794.

Chi, C., S. Feng, Y. Du, Z. Xu, E. Cousineau, B. C. Burchfiel, and S. Song (July 2023). “Diffusion Policy: Visuomotor Policy Learning via Action Diffusion”. In: *Robotics: Science and Systems XIX*. Vol. 19. ISBN: 978-0-9923747-9-2.

Cho, K., B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (Oct. 2014). “Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation”. In: *EMNLP*. Doha, Qatar, pp. 1724–1734. DOI: 10.3115/v1/D14-1179.

Christiano, P. (May 2021). *AI “Safety” vs “Control” vs “Alignment”*.

Christiano, P. F., J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei (2017). “Deep Reinforcement Learning from Human Preferences”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc.

Chromik, M. and A. Butz (2021). “Human-XAI Interaction: A Review and Design Principles for Explanation User Interfaces”. In: *Human-Computer Interaction – INTERACT 2021*. Ed. by C. Ardito, R. Lanzilotti, A. Malizia, H. Petrie, A. Piccinno, G. Desolda, and K. Inkpen. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 619–640. ISBN: 978-3-030-85616-8. DOI: 10.1007/978-3-030-85616-8_36.

Cieslak, C., A. Shah, B. Clark, and P. Childs (Sept. 2023). “Wind-Turbine Inspection, Maintenance and Repair Robotic System”. In: *ASME Turbo Expo 2023: Turbomachinery Technical Conference and Exposition*. American Society of Mechanical Engineers Digital Collection. DOI: 10.1115/GT2023-101713.

Colledanchise, M. and P. Ögren (July 2018). *Behavior Trees in Robotics and AI: An Introduction*. Boca Raton: CRC Press. ISBN: 978-0-429-48910-5. DOI: 10.1201/9780429489105.

Cooper, M. C. and J. Marques-Silva (Mar. 2023). “Tractability of Explaining Classifier Decisions”. In: *Artificial Intelligence* 316, p. 103841. ISSN: 0004-3702. DOI: 10.1016/j.artint.2022.103841.

Covariant (2024a). *Covariant Brain*. https://covariant.ai/covariant-brain/. Company Website.

— (Mar. 2024b). *RFM-1: A World Model That Understands Physics*.

Craig, J. J. (July 2004). *Introduction to Robotics: Mechanics and Control*. 3rd ed. Upper Saddle River, N.J: Pearson. ISBN: 978-0-201-54361-2.

Cubric, M. (Aug. 2020). “Drivers, Barriers and Social Considerations for AI Adoption in Business and Management: A Tertiary Study”. In: *Technology in Society* 62, p. 101257. ISSN: 0160-791X. DOI: 10.1016/j.techsoc.2020.101257.

Cuomo, S., V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli (July 2022). “Scientific Machine Learning Through Physics–Informed Neural Networks: Where We Are and What’s Next”. In: *Journal of Scientific Computing* 92.3, p. 88. ISSN: 1573-7691. DOI: 10.1007/s10915-022-01939-z.

Cutting-Decelle, A., R. Young, J. Michel, R. Grangel, J. Le Cardinal, and J. Bourey (June 2007). “ISO 15531 MANDATE: A Product-process-resource Based Approach for Managing Modularity in Production Management”. In: *Concurrent Engineering* 15.2, pp. 217–235. ISSN: 1063-293X. DOI: 10.1177/1063293X07079329.

D’Silva, V., D. Kroening, and G. Weissenbacher (July 2008). “A Survey of Automated Techniques for Formal Software Verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7, pp. 1165–1178. ISSN: 1937-4151. DOI: 10.1109/TCAD.2008.923410.

Dalrymple, D. "., J. Skalse, Y. Bengio, S. Russell, M. Tegmark, S. Seshia, S. Omohundro, C. Szegedy, B. Goldhaber, N. Ammann, A. Abate, J. Halpern, C. Barrett, D. Zhao, T. Zhi-Xuan, J. Wing, and J. Tenenbaum (July 2024). *Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems*. DOI: 10.48550/arXiv.2405.06624. arXiv: 2405.06624 [cs].

Dautenhahn, K. (Feb. 2007). “Socially Intelligent Robots: Dimensions of Human–Robot Interaction”. In: *Philosophical Transactions of the Royal Society B: Biological Sciences*. DOI: 10.1098/rstb.2006.2004.

De Schutter, J., T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx (May 2007). “Constraint-Based Task Specification and Estimation for Sensor-Based Robot Systems in the Presence of Geometric Uncertainty”. In: *The International Journal of Robotics Research* 26.5, pp. 433–455. ISSN: 0278-3649. DOI: 10.1177/027836490707809107.

Deacon, T. W. (Apr. 1998). *The Symbolic Species: The Co-evolution of Language and the Brain*. Illustrated Edition. New York, NY: W. W. Norton & Company. ISBN: 978-0-393-31754-1.

Deb, K., A. Pratap, S. Agarwal, and T. Meyarivan (Apr. 2002). “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2, pp. 182–197. ISSN: 1941-0026. DOI: 10.1109/4235.996017.

Degrave, J., M. Hermans, J. Dambre, and F. Wyffels (2019). “A Differentiable Physics Engine for Deep Learning in Robotics”. In: *Frontiers in Neurorobotics* 13. ISSN: 1662-5218. DOI: 10.3389/fnbot.2019.00006.

Deits, R., S. Tellex, P. Thaker, D. Simeonov, T. Kollar, and N. Roy (June 2013). “Clarifying Commands with Information-Theoretic Human-Robot Dialog”. In: *J. Hum.-Robot Interact.* 2.2, pp. 58–79. DOI: 10.5898/JHRI.2.2.Deits.

DeMers, D. and K. Kreutz-Delgado (Jan. 1997). “Inverse Kinematics of Dextrous Manipulators”. In: *Neural Systems for Robotics*. Ed. by O. Omidvar and P. van der Smagt. Boston: Academic Press, pp. 75–116. ISBN: 978-0-08-092509-7. DOI: 10.1016/B978-0-08-092509-7.50008-7.

Denavit, J. and R. S. Hartenberg (June 1955). “A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices”. In: *Journal of Applied Mechanics* 22.2, pp. 215–221. ISSN: 0021-8936. DOI: 10.1115/1.4011045.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (June 2009). “ImageNet: A Large-Scale Hierarchical Image Database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

Diab, M., M. Pomarlan, D. Be&szlig;ler, A. Abkari, J. Rossel, J. Bateman, and M. Beetz (2019). “An Ontology for Failure Interpretation in Automated Planning and Execution”. In: *Fourth Iberian Robotics Conference*.

Dijkstra, E. W. (1971). *EWD 316: A Short Introduction to the Art of Programming*.

— (Apr. 1977). “Programming: From Craft to Scientific Discipline”. In: *Proceedings of the International Computing Symposium 1977*. Ed. by E. Morlet and D. Ribbens. Liège, Belgium: North-Holland, pp. 23–30.

Do, Q. and M. E. Hasselmo (Dec. 2021). “Neural Circuits and Symbolic Processing”. In: *Neurobiology of learning and Memory* 186, p. 107552. ISSN: 1074-7427. DOI: 10.1016/j.nlm.2021.107552.

Dosovitskiy, A., L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby (Oct. 2020). “An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *International Conference on Learning Representations*.

Driess, D., J.-S. Ha, and M. Toussaint (July 2020). “Deep Visual Reasoning: Learning to Predict Action Sequences for Task and Motion Planning from an Initial Scene Image”. In: *Robotics: Science and Systems XVI*. DOI: 10.15607/RSS.2020.XVI.003.

Driess, D., F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence (July 2023). “PaLM-E: An Embodied Multimodal Language Model”. In: *Proceedings of the 40th International Conference on Machine Learning*. Vol. 202. ICML’23. Honolulu, Hawaii, USA: JMLR.org, pp. 8469–8488.

Dubey, A. et al. (Aug. 2024). *The Llama 3 Herd of Models*. DOI: 10.48550/arXiv.2407.21783. arXiv: 2407.21783 `[cs]`.

Dulęba, I. and M. Opałka (June 2013). “A Comparison of Jacobian-based Methods of Inverse Kinematics for Serial Robot Manipulators”. In: *International Journal of Applied Mathematics and Computer Science* 23.2, pp. 373–382.

Edmonds, M., F. Gao, H. Liu, X. Xie, S. Qi, B. Rothrock, Y. Zhu, Y. N. Wu, H. Lu, and S.-C. Zhu (Dec. 2019). “A Tale of Two Explanations: Enhancing Human Trust by Explaining Robot Behavior”. In: *Science Robotics* 4.37. ISSN: 2470-9476. DOI: 10.1126/scirobotics.aay4663.

Ekvall, S. and D. Kragic (Sept. 2006). “Learning Task Models from Multiple Human Demonstrations”. In: *ROMAN 2006 - The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pp. 358–363. DOI: 10.1109/ROMAN.2006.314460.

Elbanhawi, M. and M. Simic (2014). “Sampling-Based Robot Motion Planning: A Review”. In: *IEEE Access* 2, pp. 56–77. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2302442.

Elhage, N., T. Hume, C. Olsson, N. Schiefer, T. Henighan, S. Kravec, Z. Hatfield-Dodds, R. Lasenby, D. Drain, C. Chen, R. Grosse, S. McCandlish, J. Kaplan, D. Amodei, M. Wattenberg, and C. Olah (Sept. 2022). *Toy Models of Superposition*. DOI: 10.48550/arXiv.2209.10652.

ElMaraghy, H. A. and J. M. Rondeau (Mar. 1992). “Automatic Robot Program Synthesis for Assembly”. In: *Robotica* 10.2, pp. 113–123. ISSN: 1469-8668, 0263-5747. DOI: 10.1017/S0263574700007530.

Epic Games (Apr. 2019). *Unreal Engine*.

Ernst, H. A. (May 1962). “MH-1, a Computer-Operated Mechanical Hand”. In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE ’62 (Spring). New York, NY, USA: Association for Computing Machinery, pp. 39–51. ISBN: 978-1-4503-7875-8. DOI: 10.1145/1460833.1460839.

Espiau, B., K. Kapellos, and M. Jourdan (1996). “Formal Verification in Robotics: Why and How?” In: *Robotics Research*. Ed. by G. Giralt and G. Hirzinger. London: Springer, pp. 225–236. ISBN: 978-1-4471-1021-7. DOI: 10.1007/978-1-4471-1021-7_26.

FakhrHosseini, S., K. Chan, C. Lee, M. Jeon, H. Son, J. Rudnik, and J. Coughlin (Feb. 2024). “User Adoption of Intelligent Environments: A Review of Technology Adoption Models, Challenges, and Prospects”. In: *International Journal of Human–Computer Interaction* 40.4, pp. 986–998. ISSN: 1044-7318. DOI: 10.1080/10447318.2022.2118851.

Fan, A., S. Bhosale, H. Schwenk, Z. Ma, A. El-Kishky, S. Goyal, M. Baines, O. Celebi, G. Wenzek, V. Chaudhary, N. Goyal, T. Birch, V. Liptchinsky, S. Edunov, E. Grave, M. Auli, and A. Joulin (Jan. 2021). “Beyond English-Centric Multilingual Machine Translation”. In: *The Journal of Machine Learning Research* 22.1, 107:4839–107:4886. ISSN: 1532-4435.

Fan, H., X. Liu, J. Y. H. Fuh, W. F. Lu, and B. Li (Jan. 2024). “Embodied Intelligence in Manufacturing: Leveraging Large Language Models for Autonomous Industrial

Robotics”. In: *Journal of Intelligent Manufacturing*. ISSN: 1572-8145. DOI: 10.1007/s10845-023-02294-y.

FANUC America Corporation (2014). *FANUC America Corporation SYSTEM R-30iA and R-30iB Controller KAREL Reference Manual*. Tech. rep. MARRC75KR07091E Rev H. Rochester Hills, Michigan: FANUC America Corporation.

— (2023). *2023 Collaborative Robot Brochure*.

Ferrario, A. and M. Loi (June 2022). “How Explainability Contributes to Trust in AI”. In: *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*. FAccT ’22. New York, NY, USA: Association for Computing Machinery, pp. 1457–1466. ISBN: 978-1-4503-9352-2. DOI: 10.1145/3531146.3533202.

Fikes, R. E. and N. J. Nilsson (Dec. 1971). “Strips: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artificial Intelligence* 2.3-4, pp. 189–208. ISSN: 00043702. DOI: 10.1016/0004-3702(71)90010-5.

Finn, C., P. Abbeel, and S. Levine (July 2017a). “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks”. In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR, pp. 1126–1135.

Finn, C. and S. Levine (May 2017). “Deep Visual Foresight for Planning Robot Motion”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2786–2793. DOI: 10.1109/ICRA.2017.7989324.

Finn, C., K. Xu, and S. Levine (2018). “Probabilistic Model-Agnostic Meta-Learning”. In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc.

Finn, C., T. Yu, T. Zhang, P. Abbeel, and S. Levine (Oct. 2017b). “One-Shot Visual Imitation Learning via Meta-Learning”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. PMLR, pp. 357–368.

Fischer, T., J.-Y. Puigbò, D. Camilleri, P. D. H. Nguyen, C. Moulin-Frier, S. Lallée, G. Metta, T. J. Prescott, Y. Demiris, and P. F. M. J. Verschure (Mar. 2018). “iCub-HRI: A Software Framework for Complex Human–Robot Interaction Scenarios on the iCub Humanoid Robot”. In: *Frontiers in Robotics and AI* 5. ISSN: 2296-9144. DOI: 10.3389/frobt.2018.00022.

Flanagan, J. R., M. C. Bowman, and R. S. Johansson (Dec. 2006). “Control Strategies in Object Manipulation Tasks”. In: *Current Opinion in Neurobiology*. Motor Systems / Neurobiology of Behaviour 16.6, pp. 650–659. ISSN: 0959-4388. DOI: 10.1016/j.conb.2006.10.005.

Fodor, J. A. (Jan. 1980). *The Language of Thought*. Cambridge, Mass: Harvard University Press. ISBN: 978-0-674-51030-2.

Frank, M. and P. Wolfe (1956). “An Algorithm for Quadratic Programming”. In: *Naval Research Logistics Quarterly* 3.1-2, pp. 95–110. ISSN: 1931-9193. DOI: 10.1002/nav.3800030109.

Franko, J., S. Du, S. Kallweit, E. Duelberg, and H. Engemann (Jan. 2020). “Design of a Multi-Robot System for Wind Turbine Maintenance”. In: *Energies* 13.10, p. 2552. ISSN: 1996-1073. DOI: 10.3390/en13102552.

Frans, K., J. Ho, X. Chen, P. Abbeel, and J. Schulman (Feb. 2018). "Meta Learning Shared Hierarchies". In: *International Conference on Learning Representations*.

Fu, H., C. Liu, B. Wu, F. Li, J. Tan, and J. Sun (Feb. 2023). "CatSQL: Towards Real World Natural Language to SQL Applications". In: *Proc. VLDB Endow.* 16.6, pp. 1534–1547. ISSN: 2150-8097. DOI: 10.14778/3583140.3583165.

Füßl, A., V. Nissen, and S. H. Heringklee (2024). "An Explanation User Interface for a Knowledge Graph-Based XAI Approach to Process Analysis". In: *Advanced Information Systems Engineering Workshops*. Ed. by J. P. A. Almeida, C. Di Ciccio, and C. Kalloniatis. Cham: Springer Nature Switzerland, pp. 72–84. ISBN: 978-3-031-61003-5. DOI: 10.1007/978-3-031-61003-5_7.

Gabriel, I. (Sept. 2020). "Artificial Intelligence, Values, and Alignment". In: *Minds and Machines* 30.3, pp. 411–437. ISSN: 1572-8641. DOI: 10.1007/s11023-020-09539-2.

Gao, D., H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou (May 2024). "Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation". In: *Proc. VLDB Endow.* 17.5, pp. 1132–1145. ISSN: 2150-8097. DOI: 10.14778/3641204.3641221.

Garrett, C. R., R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez (Oct. 2020). "Integrated Task and Motion Planning". In: *arXiv:2010.01083 [cs]*. arXiv: 2010.01083 `[cs]`.

Garza, A., C. Challu, and M. Mergenthaler-Canseco (May 2024). *TimeGPT-1*. DOI: 10.48550/arXiv.2310.03589. arXiv: 2310.03589 `[cs, stat]`.

Georgievski, I. and M. Aiello (May 2015). "HTN Planning: Overview, Comparison, and Beyond". In: *Artificial Intelligence* 222, pp. 124–156. ISSN: 0004-3702. DOI: 10.1016/j.artint.2015.02.002.

Ghadirzadeh, A., X. Chen, P. Poklukar, C. Finn, M. Björkman, and D. Kragic (Sept. 2021). "Bayesian Meta-Learning for Few-Shot Policy Adaptation Across Robotic Platforms". In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1274–1280. DOI: 10.1109/IROS51168.2021.9636628.

Ghallab, M., D. S. Nau, P. Traverso, and G. Malik (May 2004). *Automated Planning: Theory and Practice*. Amsterdam ; Boston: Morgan Kaufmann Publishers In. ISBN: 978-1-55860-856-6.

Ghosh, D., H. Walke, K. Pertsch, K. Black, O. Mees, S. Dasari, J. Hejna, T. Kreiman, C. Xu, J. Luo, Y. Tan, L. Chen, Q. Vuong, T. Xiao, P. Sanketi, D. Sadigh, C. Finn, and S. Levine (July 2024). "Octo: An Open-Source Generalist Robot Policy". In: DOI: 10.15607/RSS.2024.XX.090.

Gilpin, L. H., D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal (Oct. 2018). "Explaining Explanations: An Overview of Interpretability of Machine Learning". In: *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 80–89. DOI: 10.1109/DSAA.2018.00018.

Goldenberg, A., B. Benhabib, and R. Fenton (Mar. 1985). "A Complete Generalized Solution to the Inverse Kinematics of Robots". In: *IEEE Journal on Robotics and Automation* 1.1, pp. 14–20. ISSN: 2374-8710. DOI: 10.1109/JRA.1985.1086995.

Google (July 2024). *Python Client Library*. Google LLC.

Gorostiza, J. F. and M. A. Salichs (Dec. 2011). “End-User Programming of a Social Robot by Dialog”. In: *Robotics and Autonomous Systems* 59.12, pp. 1102–1114. ISSN: 0921-8890. DOI: 10.1016/j.robot.2011.07.009.

Gray, J. and B. Rumpe (June 2022). “Explicit versus Implicit Models: What Are Good Languages for Modeling?” In: *Software and Systems Modeling* 21.3, pp. 839–841. ISSN: 1619-1374. DOI: 10.1007/s10270-022-01001-4.

Grønsund, T. and M. Aanestad (June 2020). “Augmenting the Algorithm: Emerging Human-in-the-Loop Work Configurations”. In: *The Journal of Strategic Information Systems*. Strategic Perspectives on Digital Work and Organizational Transformation 29.2, p. 101614. ISSN: 0963-8687. DOI: 10.1016/j.jsis.2020.101614.

Groth, O., C.-M. Hung, A. Vedaldi, and I. Posner (May 2021). “Goal-Conditioned End-to-End Visuomotor Control for Versatile Skill Primitives”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1319–1325. DOI: 10.1109/ICRA48506.2021.9560752.

Gugliermo, S., E. Schaffernicht, C. Koniaris, and F. Pecora (June 2023). “Learning Behavior Trees From Planning Experts Using Decision Tree and Logic Factorization”. In: *IEEE Robotics and Automation Letters* 8.6, pp. 3534–3541. ISSN: 2377-3766. DOI: 10.1109/LRA.2023.3268598.

Gunjal, A., J. Yin, and E. Bas (Mar. 2024). “Detecting and Preventing Hallucinations in Large Vision Language Models”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.16, pp. 18135–18143. ISSN: 2374-3468. DOI: 10.1609/aaai.v38i16.29771.

Guo, H., F. Wu, Y. Qin, R. Li, K. Li, and K. Li (July 2023). “Recent Trends in Task and Motion Planning for Robotics: A Survey”. In: *ACM Computing Surveys* 55.13s, 289:1–289:36. ISSN: 0360-0300. DOI: 10.1145/3583136.

Gupta, A., V. Kumar, C. Lynch, S. Levine, and K. Hausman (May 2020). “Relay Policy Learning: Solving Long-Horizon Tasks via Imitation and Reinforcement Learning”. In: *Proceedings of the Conference on Robot Learning*. PMLR, pp. 1025–1037.

Gurumurthy, S., S. Kumar, and K. Sycara (May 2020). “MAME : Model-Agnostic Meta-Exploration”. In: *Proceedings of the Conference on Robot Learning*. PMLR, pp. 910–922.

Ha, J.-S., Y.-J. Park, H.-J. Chae, S.-S. Park, and H.-L. Choi (May 2021). “Distilling a Hierarchical Policy for Planning and Control via Representation and Reinforcement Learning”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4459–4466. DOI: 10.1109/ICRA48506.2021.9561017.

Haddadin, S., S. Parusel, L. Johannsmeier, S. Golz, S. Gabl, F. Walch, M. Sabaghian, C. Jähne, L. Hausperger, and S. Haddadin (June 2022). “The Franka Emika Robot: A Reference Platform for Robotics Research and Education”. In: *IEEE Robotics & Automation Magazine* 29.2, pp. 46–64. ISSN: 1558-223X. DOI: 10.1109/MRA.2021.3138382.

Haidu, A., D. Beßler, A. K. Bozcuoğlu, and M. Beetz (Oct. 2018). "KnowRobSIM — Game Engine-Enabled Knowledge Processing Towards Cognition-Enabled Robot Control". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4491–4498. DOI: 10.1109/IROS.2018.8593935.

Haidu, A. and M. Beetz (Oct. 2016). "Action Recognition and Interpretation from Virtual Demonstrations". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2833–2838. DOI: 10.1109/IROS.2016.7759439.

— (May 2021). "Automated Acquisition of Structured, Semantic Models of Manipulation Activities from Human VR Demonstration". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9460–9466. DOI: 10.1109/ICRA48506.2021.9562016.

Hansen, J., F. Hogan, D. Rivkin, D. Meger, M. Jenkin, and G. Dudek (May 2022). "Visuotactile-RL: Learning Multimodal Manipulation Policies with Deep Reinforcement Learning". In: *2022 International Conference on Robotics and Automation (ICRA)*, pp. 8298–8304. DOI: 10.1109/ICRA46639.2022.9812019.

Hart, S. G. (Oct. 2006). "Nasa-Task Load Index (NASA-TLX); 20 Years Later". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50.9, pp. 904–908. ISSN: 1071-1813. DOI: 10.1177/154193120605000909.

Hasterok, C. and J. Stompe (Sept. 2022). "PAISE® – Process Model for AI Systems Engineering". In: *at - Automatisierungstechnik* 70.9, pp. 777–786. ISSN: 2196-677X. DOI: 10.1515/auto-2022-0020.

Haug, E. J. (Nov. 2021). "Manipulator Kinematics and Dynamics on Differentiable Manifolds: Part I Kinematics". In: *Journal of Computational and Nonlinear Dynamics* 17.021002. ISSN: 1555-1415. DOI: 10.1115/1.4052652.

Hayes, B. and B. Scassellati (May 2016). "Autonomously Constructing Hierarchical Task Networks for Planning and Human-Robot Collaboration". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5469–5476. DOI: 10.1109/ICRA.2016.7487760.

He, F., T. Liu, and D. Tao (2019a). "Control Batch Size and Learning Rate to Generalize Well: Theoretical and Empirical Evidence". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc.

He, K., R. Girshick, and P. Dollar (Oct. 2019b). "Rethinking ImageNet Pre-Training". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 4917–4926. DOI: 10.1109/ICCV.2019.00502.

He, K., X. Zhang, S. Ren, and J. Sun (June 2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

Heimann, O. and J. Guhl (Sept. 2020). "Industrial Robot Programming Methods: A Scoping Review". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1, pp. 696–703. DOI: 10.1109/ETFA46521.2020.9211997.

Heimberger, H., D. Horvat, and F. Schultmann (2023). "Assessing AI-Readiness in Production—A Conceptual Approach". In: *Intelligent and Transformative*

Production in Pandemic Times. Ed. by C.-Y. Huang, R. Dekkers, S. F. Chiu, D. Popescu, and L. Quezada. Cham: Springer International Publishing, pp. 249–257. ISBN: 978-3-031-18641-7. DOI: 10.1007/978-3-031-18641-7_24.

Heimberger, H., D. Horvat, and F. Schultmann (Aug. 2024). “Exploring the Factors Driving AI Adoption in Production: A Systematic Literature Review and Future Research Agenda”. In: *Information Technology and Management*. ISSN: 1573-7667. DOI: 10.1007/s10799-024-00436-z.

Higy, B., A. Mereta, G. Metta, and L. Badino (Feb. 2018). “Speech Recognition for the iCub Platform”. In: *Frontiers in Robotics and AI* 5, p. 10. ISSN: 2296-9144. DOI: 10.3389/frobt.2018.00010.

Himes, M. D., J. Harrington, A. D. Cobb, A. G. Baydin, F. Soboczenski, M. D. O’Beirne, S. Zorzan, D. Wright, Z. Scheffer, S. Domagal-Goldman, and G. Arney (Oct. 2020). “Accelerating Bayesian Inference via Neural Networks: Application to Exoplanet Retrievals”. In: *AAS Division of Planetary Science meeting* 52, p. 207.07.

Hinton, G. E. (1989). *Learning Distributed Representations of Concepts*. Parallel Distributed Processing: Implications for Psychology and Neurobiology. New York, NY, US: Clarendon Press/Oxford University Press, p. 61. ISBN: 978-0-19-852178-5.

Hirzle, A., A. Alonso Garcia, and A. Burkhardt (Jan. 2008). *Steuerungstechnische Standards als Fundament für die Leittechnik*. Tech. rep. DIV Deutscher Industrieverlag GmbH.

Hoebert, T., W. Lepuschitz, M. Vincze, and M. Merdan (Feb. 2023). “Knowledge-Driven Framework for Industrial Robotic Systems”. In: *Journal of Intelligent Manufacturing* 34.2, pp. 771–788. ISSN: 1572-8145. DOI: 10.1007/s10845-021-01826-8.

Hoffmann, J., S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre (Mar. 2022). *Training Compute-Optimal Large Language Models*. DOI: 10.48550/arXiv.2203.15556. arXiv: 2203.15556 [cs].

Hoffmann, J. (2011). “Everything You Always Wanted to Know about Planning: (But Were Afraid to Ask)”. In: *KI 2011: Advances in Artificial Intelligence*. Ed. by J. Bach and S. Edelkamp. Vol. 7006. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–13. ISBN: 978-3-642-24454-4 978-3-642-24455-1. DOI: 10.1007/978-3-642-24455-1_1.

Hogg, C., H. Muñoz-Avila, and U. Kuter (2016). “Learning Hierarchical Task Models from Input Traces”. In: *Computational Intelligence* 32.1, pp. 3–48. ISSN: 1467-8640. DOI: 10.1111/coin.12044.

Holeňa, M., D. Linke, U. Rodemerck, and L. Bajer (2010). “Neural Networks as Surrogate Models for Measurements in Optimization Algorithms”. In: *Analytical and Stochastic Modeling Techniques and Applications*. Ed. by K. Al-Begain, D. Fiems, and W. J. Knottenbelt. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 351–366. ISBN: 978-3-642-13568-2. DOI: 10.1007/978-3-642-13568-2_25.

Honnibal, M. and I. Montani (July 2024). *Spacy: Industrial-strength Natural Language Processing (NLP) in Python*.

Hoos, H. H. (Feb. 2012). “Programming by Optimization”. In: *Communications of the ACM* 55.2, pp. 70–80. ISSN: 0001-0782. DOI: 10.1145/2076450.2076469.

Horne, B., M. Jamshidi, and N. Vadiee (Mar. 1990). “Neural Networks in Robotics: A Survey”. In: *Journal of Intelligent and Robotic Systems* 3.1, pp. 51–66. ISSN: 1573-0409. DOI: 10.1007/BF00368972.

Hornik, K. (1991). “Approximation Capabilities of Multilayer Feedforward Networks”. In: *Neural Networks* 4.2, pp. 251–257. ISSN: 08936080. DOI: 10.1016/0893-6080(91)90009-T.

Hornik, K., M. Stinchcombe, and H. White (Jan. 1989). “Multilayer Feedforward Networks Are Universal Approximators”. In: *Neural Networks* 2.5, pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

Horvat, D. and H. Heimberger (2023). “AI Readiness: An Integrated Socio-technical Framework”. In: *Proceedings of the 11th International Conference on Production Research – Americas*. Ed. by F. Deschamps, E. Pinheiro de Lima, S. E. Gouvêa da Costa, and M. G. Trentin. Cham: Springer Nature Switzerland, pp. 548–557. ISBN: 978-3-031-36121-0. DOI: 10.1007/978-3-031-36121-0_69.

Hoskins, D. A., J. N. Hwang, and J. Vagners (Mar. 1992). “Iterative Inversion of Neural Networks and Its Application to Adaptive Control”. In: *IEEE Transactions on Neural Networks* 3.2, pp. 292–301. ISSN: 1045-9227. DOI: 10.1109/72.125870.

Hospedales, T. M., A. Antoniou, P. Micaelli, and A. J. Storkey (2021). “Meta-Learning in Neural Networks: A Survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2021.3079209.

Hsu, J., J. Mao, J. Tenenbaum, and J. Wu (Dec. 2023). “What’s Left? Concept Grounding with Logic-Enhanced Foundation Models”. In: *Advances in Neural Information Processing Systems* 36, pp. 38798–38814.

Hu, Y., Q. Xie, V. Jain, J. Francis, J. Patrikar, N. Keetha, S. Kim, Y. Xie, T. Zhang, S. Zhao, Y. Q. Chong, C. Wang, K. Sycara, M. Johnson-Roberson, D. Batra, X. Wang, S. Scherer, Z. Kira, F. Xia, and Y. Bisk (Dec. 2023). *Toward General-Purpose Robots via Foundation Models: A Survey and Meta-Analysis*. DOI: 10.48550/arXiv.2312.08782. arXiv: `2312.08782 [cs]`.

Hu, Y., L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand (Sept. 2019a). “DiffTaichi: Differentiable Programming for Physical Simulation”. In: *International Conference on Learning Representations*.

Hu, Y., J. Liu, A. Spielberg, J. B. Tenenbaum, W. T. Freeman, J. Wu, D. Rus, and W. Matusik (May 2019b). “ChainQueen: A Real-Time Differentiable Physical Simulator for Soft Robotics”. In: *2019 International Conference on Robotics and Automation (ICRA)*, pp. 6265–6271. DOI: 10.1109/ICRA.2019.8794333.

Hu, Z., Z. Gan, W. Li, W. Guo, X. Gao, and J. Zhu (Oct. 2022). "Learning From Demonstrations Via Multi-Level and Multi-Attention Domain-Adaptive Meta-Learning". In: *IEEE Robotics and Automation Letters* 7.4, pp. 11910–11917. ISSN: 2377-3766. DOI: 10.1109/LRA.2022.3207558.

Huang, C., O. Mees, A. Zeng, and W. Burgard (May 2023a). "Visual Language Maps for Robot Navigation". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10608–10615. DOI: 10.1109/ICRA48891.2023.10160969.

Huang, J. and M. Cakmak (Mar. 2017). "Code3: A System for End-to-End Programming of Mobile Manipulator Robots for Novices and Experts". In: *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. HRI '17. New York, NY, USA: Association for Computing Machinery, pp. 453–462. ISBN: 978-1-4503-4336-7. DOI: 10.1145/2909824.3020215.

Huang, L., W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu (Nov. 2023b). *A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions*. DOI: 10.48550/arXiv.2311.05232. arXiv: 2311.05232 `[cs]`.

Huang, Y., L. Rozo, J. Silvério, and D. G. Caldwell (June 2019). "Kernelized Movement Primitives". In: *The International Journal of Robotics Research* 38.7, pp. 833–852. ISSN: 0278-3649. DOI: 10.1177/0278364919846363.

Huben, R., H. Cunningham, L. R. Smith, A. Ewart, and L. Sharkey (Oct. 2023). "Sparse Autoencoders Find Highly Interpretable Features in Language Models". In: *The Twelfth International Conference on Learning Representations*.

Hurault, A. and J. Marques-Silva (2023). "Certified Logic-Based Explainable AI – The Case of Monotonic Classifiers". In: *Tests and Proofs*. Ed. by V. Prevosto and C. Seceleanu. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, pp. 51–67. ISBN: 978-3-031-38828-6. DOI: 10.1007/978-3-031-38828-6_4.

IEEE Standards Association (May 2023). *ISO/IEC/IEEE International Standard - Systems and Software Engineering–System Life Cycle Processes*. Standard. Piscataway, NJ.

Ijspeert, A. J., J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal (Feb. 2013). "Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors". In: *Neural Computation* 25.2, pp. 328–373. ISSN: 0899-7667. DOI: 10.1162/NECO_a_00393.

Ijspeert, A. J., J. Nakanishi, and S. Schaal (Sept. 2002). "Learning Rhythmic Movements by Demonstration Using Nonlinear Oscillators". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 1, pp. 958–963. DOI: 10.1109/IRDS.2002.1041514.

Ionescu, T. B. and S. Schlund (Jan. 2021). "Programming Cobots by Voice: A Human-Centered, Web-Based Approach". In: *Procedia CIRP*. 8th CIRP Conference of Assembly Technology and Systems 97, pp. 123–129. ISSN: 2212-8271. DOI: 10.1016/j.procir.2020.05.213.

Jaiswal, K., I. Kuzminykh, and S. Modgil (Sept. 2024). “Understanding the Skills Gap between Higher Education and Industry in the UK in Artificial Intelligence Sector”. In: *Industry and Higher Education*, p. 09504222241280441. ISSN: 0950-4222. DOI: 10.1177/09504222241280441.

Jäkel, R. (2013). “Learning of Generalized Manipulation Strategies in Service Robotics”. PhD thesis. Karlsruhe: Karlsruhe Institute of Technology.

Jäkel, R. and G. Dirschl (May 2016). “Verfahren und System zur Programmierung eines Roboters”. EP3013537A1.

Jäkel, R., S. W. Rühl, S. R. Schmidt-Rohr, M. Lösch, Z. Xue, and R. Dillmann (2012). “Layered Programming by Demonstration and Planning for Autonomous Robot Manipulation”. In: *Advanced Bimanual Manipulation: Results from the DEXMART Project*. Ed. by B. Siciliano. Springer Tracts in Advanced Robotics. Berlin, Heidelberg: Springer, pp. 1–57. ISBN: 978-3-642-29041-1. DOI: 10.1007/978-3-642-29041-1_1.

Janiak, M. and K. Tchoń (2008). “Extended Jacobian Inverse Kinematics and Approximation of Distributions”. In: *Advances in Robot Kinematics: Analysis and Design*. Ed. by J. Lenarčič and P. Wenger. Dordrecht: Springer Netherlands, pp. 138–146. ISBN: 978-1-4020-8600-7. DOI: 10.1007/978-1-4020-8600-7_15.

Jatavallabhula, K. M., M. Macklin, D. Fox, A. Garg, and F. Ramos (Mar. 2023). “Bayesian Object Models for Robotic Interaction with Differentiable Probabilistic Programming”. In: *Proceedings of The 6th Conference on Robot Learning*. PMLR, pp. 1563–1574.

Jensen, C., R. Reed, R. Marks, M. El-Sharkawi, Jae-Byung Jung, R. Miyamoto, G. Anderson, and C. Eggen (Sept. 1999). “Inversion of Feedforward Neural Networks: Algorithms and Applications”. In: *Proceedings of the IEEE* 87.9, pp. 1536–1549. ISSN: 00189219. DOI: 10.1109/5.784232.

Jensen, M., N. Emery-Xu, and R. Trager (Feb. 2023). *Industrial Policy for Advanced AI: Compute Pricing and the Safety Tax*. DOI: 10.48550/arXiv.2302.11436. arXiv: 2302.11436 `[econ, q-fin]`.

Jiang, Z., F. Jovan, P. Moradi, T. Richardson, S. Bernardini, S. Watson, A. Weightman, and D. Hine (2023). “A Multirobot System for Autonomous Deployment and Recovery of a Blade Crawler for Operations and Maintenance of Offshore Wind Turbine Blades”. In: *Journal of Field Robotics* 40.1, pp. 73–93. ISSN: 1556-4967. DOI: 10.1002/rob.22117.

Jin, W., Z. Wang, Z. Yang, and S. Mou (2020). “Pontryagin Differentiable Programming: An End-to-End Learning and Control Framework”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 7979–7992.

Jokinen, K. (May 2024). “The Need for Grounding in LLM-based Dialogue Systems”. In: *Proceedings of the Workshop: Bridging Neurons and Symbols for Natural Language Processing and Knowledge Graphs Reasoning (NeusymBridge) @ LREC-COLING-2024*. Ed. by T. Dong, E. Hinrichs, Z. Han, K. Liu, Y. Song, Y. Cao, C. F. Hempelmann, and R. Sifa. Torino, Italia: ELRA and ICCL, pp. 45–52.

Jonschkowski, R., D. Rastogi, and O. Brock (June 2018). "Differentiable Particle Filters: End-to-End Learning with Algorithmic Priors". In: *Robotics: Science and Systems XIV*. Vol. 14. ISBN: 978-0-9923747-4-7.

Joublin, F., A. Ceravola, P. Smirnov, F. Ocker, J. Deigmoeller, A. Belardinelli, C. Wang, S. Hasler, D. Tanneberg, and M. Gienger (Oct. 2023). *CoPAL: Corrective Planning of Robot Actions with Large Language Models*. DOI: 10.48550/arXiv.2310.07263. arXiv: 2310.07263 [cs].

Jumper, J., R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis (Aug. 2021). "Highly Accurate Protein Structure Prediction with AlphaFold". In: *Nature* 596.7873, pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

Kaelbling, L. (1993). "Learning to Achieve Goals". In: *International Joint Conference on Artificial Intelligence*.

Kaelbling, L. P. and T. Lozano-Perez (May 2011). "Hierarchical Task and Motion Planning in the Now". In: *2011 IEEE International Conference on Robotics and Automation*. Shanghai, China: IEEE, pp. 1470–1477. ISBN: 978-1-61284-386-5. DOI: 10.1109/ICRA.2011.5980391.

Kaelbling, L. P. and T. Lozano-Pérez (Aug. 2013). "Integrated Task and Motion Planning in Belief Space". In: *The International Journal of Robotics Research* 32.9-10, pp. 1194–1227. ISSN: 0278-3649. DOI: 10.1177/0278364913484072.

Kaplan, J., S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei (Jan. 2020). *Scaling Laws for Neural Language Models*. DOI: 10.48550/arXiv.2001.08361. arXiv: 2001.08361 [cs, stat].

Kaushik, R., T. Anne, and J.-B. Mouret (Oct. 2020). "Fast Online Adaptation in Robotics through Meta-Learning Embeddings of Simulated Priors". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5269–5276. DOI: 10.1109/IROS45743.2020.9341462.

Kavraki, L., P. Svestka, J.-C. Latombe, and M. Overmars (Aug. 1996). "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces". In: *IEEE Transactions on Robotics and Automation* 12.4, pp. 566–580. ISSN: 2374-958X. DOI: 10.1109/70.508439.

Kawamura, K., S. Gordon, P. Ratanaswasd, E. Erdemir, and J. Hall (Dec. 2008). "Implementation of Cognitive Control for a Humanoid Robot". In: *International Journal of Humanoid Robotics* 5, pp. 547–586. DOI: 10.1142/S0219843608001558.

Kazhoyan, G., A. Niedzwiecki, and M. Beetz (2020). "Towards Plan Transformations for Real-World Mobile Fetch and Place". In: *IEEE International Conference on Robotics and Automation (ICRA)*. DOI: 10.1109/ICRA40945.2020.9197446.

Kenghagho Kenfack, F., F. A. Siddiky, F. Balint-Benczedi, and M. Beetz (Oct. 2020). "RobotVQA — A Scene-Graph- and Deep-Learning-based Visual Question Answering System for Robot Manipulation". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 9667–9674. DOI: 10.1109/IROS45743.2020.9341186.

Kerley, C. I., L. Y. Cai, Y. Tang, L. L. Beason-Held, S. M. Resnick, L. E. Cutting, and B. A. Landman (Feb. 2023). "Batch Size: Go Big or Go Home? Counterintuitive Improvement in Medical Autoencoders with Smaller Batch Size". In: *Proceedings of SPIE–the International Society for Optical Engineering* 12464, 124640H. ISSN: 0277-786X. DOI: 10.1117/12.2653643.

Khadir, A. C., H. Aliane, and A. Guessoum (Feb. 2021). "Ontology Learning: Grand Tour and Challenges". In: *Computer Science Review* 39, p. 100339. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2020.100339.

Kienle, C., B. Alt, O. Celik, P. Becker, D. Katic, R. Jäkel, and G. Neumann (Aug. 2024). "MuTT: A Multimodal Trajectory Transformer for Robot Skills". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Abu Dhabi, United Arab Emirates: IEEE, pp. 9644–9651. ISBN: 979-8-3503-7770-5. DOI: 10.1109/IROS58592.2024.10802198. arXiv: 2407.15660 [cs].

Kienle, C., B. Alt, D. Katic, and R. Jäkel (May 2025). "QueryCAD: Grounded Question Answering for CAD Models". In: *2025 IEEE International Conference on Robotics and Automation (ICRA)*. Atlanta, USA: IEEE. DOI: 10.48550/arXiv.2409.08704. arXiv: 2409.08704 [cs].

Kim, M. J., K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. Sanketi, Q. Vuong, T. Kollar, B. Burchfiel, R. Tedrake, D. Sadigh, S. Levine, P. Liang, and C. Finn (June 2024). *OpenVLA: An Open-Source Vision-Language-Action Model*. DOI: 10.48550/arXiv.2406.09246. arXiv: 2406.09246 [cs].

Kingma, D. P. and J. Ba (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference for Learning Representations*. San Diego: arXiv. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980 [cs].

Kirillov, A., E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollar, and R. Girshick (2023). "Segment Anything". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4015–4026.

Kirkpatrick, J., R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell (Mar. 2017). "Overcoming Catastrophic Forgetting in Neural Networks". In: *Proceedings of the National Academy of Sciences* 114.13, pp. 3521–3526. DOI: 10.1073/pnas.1611835114.

Kirsh, D. (1990). "When Is Information Explicitly Represented?" In: *Information, Language and Cognition*. Ed. by P. P. Hanson. University of British Columbia Press.

Kirsh, D. (2006). “Implicit and Explicit Representation”. In: *Encyclopedia of Cognitive Science*. John Wiley & Sons, Ltd. ISBN: 978-0-470-01886-6. DOI: 10.1002/0470018860.s00166.

Klambauer, G., T. Unterthiner, A. Mayr, and S. Hochreiter (2017). “Self-Normalizing Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc.

Klee, S. D., G. Gemignani, D. Nardi, and M. Veloso (2015). “Graph-Based Task Libraries for Robots: Generalization and Autocompletion”. In: *AI*IA 2015 Advances in Artificial Intelligence*. Ed. by M. Gavanelli, E. Lamma, and F. Riguzzi. Vol. 9336. Cham: Springer International Publishing, pp. 397–409. ISBN: 978-3-319-24308-5 978-3-319-24309-2. DOI: 10.1007/978-3-319-24309-2_30.

Klimek, M., H. Michalewski, and P. Mi\łoś (Oct. 2017). “Hierarchical Reinforcement Learning with Parameters”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. PMLR, pp. 301–313.

Knuth, D. E. (Jan. 1984). “Literate Programming”. In: *The Computer Journal* 27.2, pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97.

Kober, J., J. A. Bagnell, and J. Peters (Sept. 2013). “Reinforcement Learning in Robotics: A Survey”. In: *The International Journal of Robotics Research* 32.11, pp. 1238–1274. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364913495721.

Koert, D., G. Maeda, R. Lioutikov, G. Neumann, and J. Peters (Nov. 2016). “Demonstration Based Trajectory Optimization for Generalizable Robot Motions”. In: *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, pp. 515–522. DOI: 10.1109/HUMANOIDS.2016.7803324.

Kojima, T., S. (Gu, M. Reid, Y. Matsuo, and Y. Iwasawa (Dec. 2022). “Large Language Models Are Zero-Shot Reasoners”. In: *Advances in Neural Information Processing Systems* 35, pp. 22199–22213.

Kok, B. C. and H. Soh (Dec. 2020). “Trust in Robots: Challenges and Opportunities”. In: *Current Robotics Reports* 1.4, pp. 297–309. ISSN: 2662-4087. DOI: 10.1007/s43154-020-00029-y.

Kolesnikov, A., L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby (Aug. 2020). “Big Transfer (BiT): General Visual Representation Learning”. In: *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V*. Berlin, Heidelberg: Springer-Verlag, pp. 491–507. ISBN: 978-3-030-58557-0. DOI: 10.1007/978-3-030-58558-7_29.

Koralewski, S., G. Kazhoyan, and M. Beetz (2019). “Self-Specialization of General Robot Plans Based on Experience”. In: *IEEE Robotics and Automation Letters*. DOI: 10.1109/LRA.2019.2928771.

Kornblith, S., J. Shlens, and Q. V. Le (2019). “Do Better ImageNet Models Transfer Better?” In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2661–2671.

Kortenkamp, D., R. Simmons, and D. Brugali (2016). “Robotic Systems Architectures and Programming”. In: *Springer Handbook of Robotics*. Ed. by B. Siciliano and O.

Khatib. Springer Handbooks. Cham: Springer International Publishing, pp. 283–306. ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_12.

Kotseruba, I. and J. K. Tsotsos (Jan. 2020). “40 Years of Cognitive Architectures: Core Cognitive Abilities and Practical Applications”. In: *Artificial Intelligence Review* 53.1, pp. 17–94. ISSN: 1573-7462. DOI: 10.1007/s10462-018-9646-y.

Kramer, J. (Apr. 2007). “Is Abstraction the Key to Computing?” In: *Communications of the ACM* 50.4, pp. 36–42. ISSN: 0001-0782. DOI: 10.1145/1232743.1232745.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (Dec. 2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 1. Lake Tahoe, Nevada, USA, pp. 1097–1105.

Kroemer, O., S. Niekum, and G. Konidaris (2021). “A Review of Robot Learning for Manipulation: Challenges, Representations, and Algorithms”. In: *Journal of Machine Learning Research* 22, pp. 1–82. arXiv: 1907.03146.

Krot, K. and V. Kutia (2019). “Intuitive Methods of Industrial Robot Programming in Advanced Manufacturing Systems”. In: *Intelligent Systems in Production Engineering and Maintenance*. Ed. by A. Burduk, E. Chlebus, T. Nowakowski, and A. Tubis. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, pp. 205–214. ISBN: 978-3-319-97490-3.

Kulk, J. and J. S. Welsh (Oct. 2011). “Evaluation of Walk Optimisation Techniques for the NAO Robot”. In: *2011 11th IEEE-RAS International Conference on Humanoid Robots*, pp. 306–311. DOI: 10.1109/Humanoids.2011.6100827.

Kumar, N., T. Silver, W. McClinton, L. Zhao, S. Proulx, T. Lozano-Pérez, L. Kaelbling, and J. Barry (July 2024). “Practice Makes Perfect: Planning to Learning Skill Parameter Policies”. In: *Robotics: Science and Systems 2024*. Delft, Netherlands. DOI: 10.15607/RSS.2024.XX.040.

Lai, P.-K. (Jan. 2022). “DeepSCM: An Efficient Convolutional Neural Network Surrogate Model for the Screening of Therapeutic Antibody Viscosity”. In: *Computational and Structural Biotechnology Journal* 20, pp. 2143–2152. ISSN: 2001-0370. DOI: 10.1016/j.csbj.2022.04.035.

Laird, J., K. R. Kinkade, S. Mohan, and J. Xu (2012). “Cognitive Robotics Using the Soar Cognitive Architecture”. In: *CogRob@AAAI*.

Laird, J. E., A. Newell, and P. S. Rosenbloom (Sept. 1987). “SOAR: An Architecture for General Intelligence”. In: *Artificial Intelligence* 33.1, pp. 1–64. ISSN: 0004-3702. DOI: 10.1016/0004-3702(87)90050-6.

Lallement, R., L. de Silva, and R. Alami (2014). “HATP: An HTN Planner for Robotics”. In: *2nd ICAPS Workshop on Planning and Robotics*. Portsmouth, USA.

Lan, G. (2020). *First-Order and Stochastic Optimization Methods for Machine Learning*. Springer Series in the Data Sciences. Cham: Springer International Publishing. ISBN: 978-3-030-39567-4 978-3-030-39568-1. DOI: 10.1007/978-3-030-39568-1.

Landin, P. J. (Jan. 1964). “The Mechanical Evaluation of Expressions”. In: *The Computer Journal* 6.4, pp. 308–320. ISSN: 0010-4620. DOI: 10.1093/comjnl/6.4.308.

Lauri, M., D. Hsu, and J. Pajarinen (Feb. 2023). “Partially Observable Markov Decision Processes in Robotics: A Survey”. In: *IEEE Transactions on Robotics* 39.1, pp. 21–40. ISSN: 1941-0468. DOI: 10.1109/TRO.2022.3200138.

LaValle, S. M. (1998). *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Technical Report TR 98-11. Ames, IA: Iowa State University.

Lee, D., C. Szegedy, M. Rabe, S. Loos, and K. Bansal (Sept. 2019). “Mathematical Reasoning in Latent Space”. In: *International Conference on Learning Representations*.

Leichtmann, B., C. Humer, A. Hinterreiter, M. Streit, and M. Mara (Feb. 2023). “Effects of Explainable Artificial Intelligence on Trust and Human Behavior in a High-Risk Decision Task”. In: *Computers in Human Behavior* 139, p. 107539. ISSN: 0747-5632. DOI: 10.1016/j.chb.2022.107539.

Lembono, T. S., A. Paolillo, E. Pignat, and S. Calinon (Apr. 2020). “Memory of Motion for Warm-starting Trajectory Optimization”. In: *IEEE Robotics and Automation Letters* 5.2, pp. 2594–2601. ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2020.2972893. arXiv: 1907.01474 [cs].

Leon-Urrutia, M., D. Taibi, V. Pospelova, S. Splendore, L. Urbsiene, and U. Marjanovic (2022). “Data Literacy: An Essential Skill for the Industry”. In: *Proceedings on 18th International Conference on Industrial Systems – IS’20*. Ed. by B. Lalic, D. Gracanin, N. Tasic, and N. Simeunović. Cham: Springer International Publishing, pp. 326–331. ISBN: 978-3-030-97947-8. DOI: 10.1007/978-3-030-97947-8_43.

Levenberg, K. (1944). “A Method for the Solution of Certain Non-Linear Problems in Least Squares”. In: *Quarterly of Applied Mathematics* 2.2, pp. 164–168. ISSN: 0033-569X, 1552-4485. DOI: 10.1090/qam/10666.

Levesque, H. J. and R. J. Brachman (1987). “Expressiveness and Tractability in Knowledge Representation and Reasoning”. In: *Computational Intelligence* 3.1, pp. 78–93. ISSN: 1467-8640. DOI: 10.1111/j.1467-8640.1987.tb00176.x.

Levine, S., C. Finn, T. Darrell, and P. Abbeel (Jan. 2016). “End-to-End Training of Deep Visuomotor Policies”. In: *The Journal of Machine Learning Research* 17.1, pp. 1334–1373. ISSN: 1532-4435.

Lewis, P., E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela (2020). “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 9459–9474.

Li, C., Z. Gan, Z. Yang, J. Yang, L. Li, L. Wang, and J. Gao (May 2024). “Multimodal Foundation Models: From Specialists to General-Purpose Assistants”. In: *Foundations and Trends® in Computer Graphics and Vision* 16.1-2, pp. 1–214. ISSN: 1572-2740, 1572-2759. DOI: 10.1561/0600000110.

Li, G., Z. Jin, M. Volpp, F. Otto, R. Lioutikov, and G. Neumann (Apr. 2023a). “ProDMP: A Unified Perspective on Dynamic and Probabilistic Movement Primitives”. In: *IEEE Robotics and Automation Letters* 8.4, pp. 2325–2332. ISSN: 2377-3766. DOI: 10.1109/LRA.2023.3248443.

Li, G., C. Yuan, S. Kamarthi, M. Moghaddam, and X. Jin (July 2021). “Data Science Skills and Domain Knowledge Requirements in the Manufacturing Industry: A Gap Analysis”. In: *Journal of Manufacturing Systems* 60, pp. 692–706. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2021.07.007.

Li, X., M. Liu, H. Zhang, C. Yu, J. Xu, H. Wu, C. Cheang, Y. Jing, W. Zhang, H. Liu, H. Li, and T. Kong (Oct. 2023b). “Vision-Language Foundation Models as Effective Robot Imitators”. In: *The Twelfth International Conference on Learning Representations*.

Liang, J., W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng (May 2023). “Code as Policies: Language Model Programs for Embodied Control”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. DOI: 10.1109/ICRA48891.2023.10160591.

Liang, Y., F. Hong, Q. Lin, S. Bi, and L. Feng (July 2017). “Optimization of Robot Path Planning Parameters Based on Genetic Algorithm”. In: *2017 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pp. 529–534. DOI: 10.1109/RCAR.2017.8311917.

Liang, Y. S., D. Pellier, H. Fiorino, and S. Pesty (Jan. 2022). “iRoPro: An Interactive Robot Programming Framework”. In: *International Journal of Social Robotics* 14.1, pp. 177–191. ISSN: 1875-4805. DOI: 10.1007/s12369-021-00775-9.

Liao, P., W. Song, P. Du, and H. Zhao (Dec. 2021). “Multi-Fidelity Convolutional Neural Network Surrogate Model for Aerodynamic Optimization Based on Transfer Learning”. In: *Physics of Fluids* 33.12, p. 127121. ISSN: 1070-6631. DOI: 10.1063/5.0076538.

Lin, T.-Y., M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick (2014). “Microsoft COCO: Common Objects in Context”. In: *Computer Vision – ECCV 2014*. Ed. by D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 740–755. ISBN: 978-3-319-10602-1. DOI: 10.1007/978-3-319-10602-1_48.

Linden and Kindermann (June 1989). “Inversion of Multilayer Nets”. In: *International 1989 Joint Conference on Neural Networks*, 425–430 vol.2. DOI: 10.1109/IJCNN.1989.118277.

Liu, D. C. and J. Nocedal (Aug. 1989). “On the Limited Memory BFGS Method for Large Scale Optimization”. In: *Mathematical Programming* 45.1, pp. 503–528. ISSN: 1436-4646. DOI: 10.1007/BF01589116.

Liu, P. and C. Y. Barlow (Apr. 2017). “Wind Turbine Blade Waste in 2050”. In: *Waste Management* 62, pp. 229–240. ISSN: 0956-053X. DOI: 10.1016/j.wasman.2017.02.007.

Liu, S., Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, Q. Jiang, C. Li, J. Yang, H. Su, J. Zhu, and L. Zhang (July 2024a). *Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection*. DOI: 10.48550/arXiv.2303.05499. arXiv: 2303.05499 `[cs]`.

Liu, X., T. Zhang, Y. Gu, I. L. Iong, Y. Xu, X. Song, S. Zhang, H. Lai, X. Liu, H. Zhao, J. Sun, X. Yang, Y. Yang, Z. Qi, S. Yao, X. Sun, S. Cheng, Q. Zheng, H. Yu, H.

Zhang, W. Hong, M. Ding, L. Pan, X. Gu, A. Zeng, Z. Du, C. H. Song, Y. Su, Y. Dong, and J. Tang (Aug. 2024b). *VisualAgentBench: Towards Large Multimodal Models as Visual Foundation Agents*. DOI: 10.48550/arXiv.2408.06327. arXiv: 2408.06327 [cs].

Liu, Y., C. Chen, T. Wang, L. Cheng, and J. Qin (Aug. 2023). "Model-Agnostic Meta-Learning for Fault Diagnosis of Industrial Robots". In: *2023 28th International Conference on Automation and Computing (ICAC)*, pp. 1–6. DOI: 10.1109/ICAC57885.2023.10275255.

Lizotte, D., T. Wang, M. Bowling, and D. Schuurmans (Jan. 2007). "Automatic Gait Optimization with Gaussian Process Regression". In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 944–949.

Long, D. and B. Magerko (Apr. 2020). "What Is AI Literacy? Competencies and Design Considerations". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. New York, NY, USA: Association for Computing Machinery, pp. 1–16. ISBN: 978-1-4503-6708-0. DOI: 10.1145/3313831.3376727.

Loshchilov, I. and F. Hutter (Sept. 2018). "Decoupled Weight Decay Regularization". In: *International Conference on Learning Representations*.

— (July 2022). "SGDR: Stochastic Gradient Descent with Warm Restarts". In: *International Conference on Learning Representations*.

Lozano-Perez, T. (July 1983). "Robot Programming". In: *Proceedings of the IEEE* 71.7, pp. 821–841. ISSN: 1558-2256. DOI: 10.1109/PROC.1983.12681.

Luckcuck, M., M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher (2019). "A Summary of Formal Specification and Verification of Autonomous Robotic Systems". In: *Integrated Formal Methods*. Ed. by W. Ahrendt and S. L. Tapia Tarifa. Cham: Springer International Publishing, pp. 538–541. ISBN: 978-3-030-34968-4. DOI: 10.1007/978-3-030-34968-4_33.

Lund, K. W. and E. S. Madsen (Mar. 2024). "State-of-the-Art Value Chain Roadmap for Sustainable End-of-Life Wind Turbine Blades". In: *Renewable and Sustainable Energy Reviews* 192, p. 114234. ISSN: 1364-0321. DOI: 10.1016/j.rser.2023.114234.

Luo, H., J. Wu, J. Liu, and M. F. Antwi-Afari (Oct. 2024). "Large Language Model-Based Code Generation for the Control of Construction Assembly Robots: A Hierarchical Generation Approach". In: *Developments in the Built Environment* 19, p. 100488. ISSN: 2666-1659. DOI: 10.1016/j.dibe.2024.100488.

Lutter, M., J. Silberbauer, J. Watson, and J. Peters (May 2021). "Differentiable Physics Models for Real-world Offline Model-based Reinforcement Learning". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4163–4170. DOI: 10.1109/ICRA48506.2021.9561805.

Lynch, C., M. Khansari, T. Xiao, V. Kumar, J. Tompson, S. Levine, and P. Sermanet (Mar. 2019). "Learning Latent Plans from Play". In: *Proceedings of Machine Learning Research*. arXiv: 1903.01973.

Lynch, C. and P. Sermanet (July 2021). “Language Conditioned Imitation Learning Over Unstructured Data”. In: *Robotics: Science and Systems XVII*. Vol. 17. ISBN: 978-0-9923747-7-8.

Ma, Y., Z. Song, Y. Zhuang, J. Hao, and I. King (May 2024). *A Survey on Vision-Language-Action Models for Embodied AI*. DOI: 10.48550/arXiv.2405.14093. arXiv: 2405.14093 [cs].

Magreñán, Á. A. and I. K. Argyros (Jan. 2018). “Gauss–Newton Method”. In: *A Contemporary Study of Iterative Methods*. Academic Press, pp. 61–67. ISBN: 978-0-12-809214-9. DOI: 10.1016/B978-0-12-809214-9.00005-X.

Mandi, Z., P. Abbeel, and S. James (Oct. 2022). “On the Effectiveness of Fine-tuning Versus Meta-reinforcement Learning”. In: *Advances in Neural Information Processing Systems*.

Mania, P., F. K. Kenfack, M. Neumann, and M. Beetz (Sept. 2021). “Imagination-Enabled Robot Perception”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 936–943. DOI: 10.1109/IROS51168.2021.9636359.

Mania, P., S. Stelter, G. Kazhoyan, and M. Beetz (May 2024). “An Open and Flexible Robot Perception Framework for Mobile Manipulation Tasks”. In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 17445–17451. DOI: 10.1109/ICRA57147.2024.10610743.

Manna, Z. and R. J. Waldinger (Mar. 1971). “Toward Automatic Program Synthesis”. In: *Communications of the ACM* 14.3, pp. 151–165. ISSN: 0001-0782. DOI: 10.1145/362566.362568.

Marco-Valle, A. (2020). “Bayesian Optimization in Robot Learning: Automatic Controller Tuning and Sample-efficient Methods”. PhD thesis. Eberhard Karls Universität Tübingen.

Marcus, H. J., P. T. Ramirez, D. Z. Khan, H. Layard Horsfall, J. G. Hanrahan, S. C. Williams, D. J. Beard, R. Bhat, K. Catchpole, A. Cook, K. Hutchison, J. Martin, T. Melvin, D. Stoyanov, M. Rovers, N. Raison, P. Dasgupta, D. Noonan, D. Stocken, G. Sturt, A. Vanhoestenberghe, B. Vasey, and P. McCulloch (Jan. 2024). “The IDEAL Framework for Surgical Robotics: Development, Comparative Evaluation and Long-Term Monitoring”. In: *Nature Medicine* 30.1, pp. 61–75. ISSN: 1546-170X. DOI: 10.1038/s41591-023-02732-7.

Martin, P. and J. d. R. Millán (1997). “Combining Reinforcement Learning and Differential Inverse Kinematics for Collision-Free Motion of Multilink Manipulators”. In: *Biological and Artificial Computation: From Neuroscience to Technology*. Ed. by J. Mira, R. Moreno-Díaz, J. Cabestany, G. Goos, J. Hartmanis, and J. van Leeuwen. Vol. 1240. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1324–1333. ISBN: 978-3-540-63047-0 978-3-540-69074-0. DOI: 10.1007/BFb0032593.

Martínez-Fernández, S., J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, and S. Wagner (Apr. 2022). “Software Engineering for AI-Based Systems: A Survey”. In: *ACM Transactions on Software Engineering and Methodology* 31.2, 37e:1–37e:59. ISSN: 1049-331X. DOI: 10.1145/3487043.

Marvel, J. A., W. S. Newman, D. P. Gravel, G. Zhang, Jianjun Wang, and T. Fuhlbrigge (Feb. 2009). "Automated Learning for Parameter Optimization of Robotic Assembly Tasks Utilizing Genetic Algorithms". In: *2008 IEEE International Conference on Robotics and Biomimetics*, pp. 179–184. DOI: 10.1109/ROBIO.2009.4913000.

Marvin, G., N. Hellen, D. Jjingo, and J. Nakatumba-Nabende (2024). "Prompt Engineering in Large Language Models". In: *Data Intelligence and Cognitive Informatics*. Ed. by I. J. Jacob, S. Piramuthu, and P. Falkowski-Gilski. Singapore: Springer Nature, pp. 387–402. ISBN: 978-981-99-7962-2. DOI: 10.1007/978-981-99-7962-2_30.

Masters, D. and C. Luschi (Apr. 2018). *Revisiting Small Batch Training for Deep Neural Networks*. DOI: 10.48550/arXiv.1804.07612. arXiv: 1804.07612 `[cs, stat]`.

Mateen, S. A., N. Malvia, S. A. Khader, D. Wang, D. Srinivasan, C.-F. J. Yang, L. Schumacher, and S. Manjanna (June 2024). "Thoracic Surgery Video Analysis for Surgical Phase Recognition". In: *2nd Robot-Assisted Medical Imaging Workshop*. Yokohama, Japan: arXiv. DOI: 10.48550/arXiv.2406.09185. arXiv: 2406.09185 `[cs]`.

McDermott, D. (1991). *A Reactive Plan Language*. Research Report YALEU/DCS/RR-864. Yale University: Yale University.

Meier, F., A. Wang, G. Sutanto, Y. Lin, and P. Shah (Feb. 2022). *Differentiable and Learnable Robot Models*. arXiv: 2202.11217 `[cs]`.

Memmert, L. and E. Bittner (Jan. 2022). "Complex Problem Solving through Human-AI Collaboration: Literature Review on Research Contexts". In: *Hawaii International Conference on System Sciences 2022 (HICSS-55)*.

Mericli, C., S. D. Klee, J. Paparian, and M. Veloso (May 2014). "An Interactive Approach for Situated Task Specification through Verbal Instructions". In: *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-Agent Systems*. AAMAS '14. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp. 1069–1076. ISBN: 978-1-4503-2738-1.

Meywerk, T., V. Herdt, and R. Drechsler (Nov. 2022). "Symbolic Fault Injection for Plan-based Robotics". In: *2022 22nd International Conference on Control, Automation and Systems (ICCAS)*, pp. 1710–1715. DOI: 10.23919/ICCAS55662.2022.10003719.

Micropsi Industries (2024). *Micropsi Industries | From Traditional Vision to AI Guidance*. https://www.micropsi-industries.com/blog/white-paper-from-traditional-vision-to-ai-guidance. Company Website.

Mockus, J. (1989). *Bayesian Approach to Global Optimization: Theory and Applications*. Ed. by M. Hazewinkel. Vol. 37. Mathematics and Its Applications. Dordrecht: Springer Netherlands. ISBN: 978-94-010-6898-7 978-94-009-0909-0. DOI: 10.1007/978-94-009-0909-0.

Mohseni-Kabir, A., S. Chernova, and C. Rich (2014). "Collaborative Learning of Hierarchical Task Networks from Demonstration and Instruction". In: *Artificial*

Intelligence for Human-Robot Interaction. Association for the Advancement of Artificial Intelligence, pp. 115–117.

Mölschl, L., J. J. Hollenstein, and J. Piater (Mar. 2023). *Differentiable Forward Kinematics for TensorFlow 2*. DOI: 10.48550/arXiv.2301.09954. arXiv: 2301.09954 [cs].

Monarch, R. (July 2021). *Human-in-the-Loop Machine Learning: Active Learning and Annotation for Human-Centered AI*. Sherlter Island, NY: Manning. ISBN: 978-1-61729-674-1.

MongoDB (2023). *MongoDB*. MongoDB, Inc.

Morrison, B. W., J. N. Kelson, N. M. V. Morrison, J. M. Innes, G. Zelic, Y. Al-Saggaf, and M. Paul (2023). “You’re Not the Boss of Me, Algorithm: Increased User Control and Positive Implicit Attitudes Are Related to Greater Adherence to an Algorithmic Aid”. In: *Interacting with Computers* 35.3, pp. 452–460. DOI: 10.1093/IWC/IWAD028.

Mosbach, M., M. Andriushchenko, and D. Klakow (Sept. 2020). “On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines”. In: *International Conference on Learning Representations*.

Mosqueira-Rey, E., E. Hernández-Pereira, D. Alonso-Ríos, J. Bobes-Bascarán, and Á. Fernández-Leal (Apr. 2023). “Human-in-the-Loop Machine Learning: A State of the Art”. In: *Artificial Intelligence Review* 56.4, pp. 3005–3054. ISSN: 1573-7462. DOI: 10.1007/s10462-022-10246-w.

Mühe, H., A. Angerer, A. Hoffmann, and W. Reif (Sept. 2010). *On Reverse-Engineering the KUKA Robot Language*. DOI: 10.48550/arXiv.1009.5004. arXiv: 1009.5004 [cs].

Mukadam, M., J. Dong, X. Yan, F. Dellaert, and B. Boots (Sept. 2018). “Continuous-Time Gaussian Process Motion Planning via Probabilistic Inference”. In: *The International Journal of Robotics Research* 37.11, pp. 1319–1340. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364918790369. arXiv: 1707.07383.

Mukadam, M., X. Yan, and B. Boots (May 2016). “Gaussian Process Motion Planning”. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9–15. DOI: 10.1109/ICRA.2016.7487091.

Munkhdalai, T. and H. Yu (July 2017). “Meta Networks”. In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR, pp. 2554–2563.

Myers, V., A. W. He, K. Fang, H. R. Walke, P. Hansen-Estruch, C.-A. Cheng, M. Jalobeanu, A. Kolobov, A. Dragan, and S. Levine (Aug. 2023). “Goal Representations for Instruction Following: A Semi-Supervised Language Interface to Control”. In: *7th Annual Conference on Robot Learning*.

Narodytska, N. (Oct. 2018). “Formal Verification of Deep Neural Networks”. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–1. DOI: 10.23919/FMCAD.2018.8603017.

Nau, D. S., Y. Cao, A. Lotem, and H. Muñoz-Avila (July 1999). “SHOP: Simple Hierarchical Ordered Planner”. In: *International Joint Conference on Artificial Intelligence*.

Naumann, M. (2017). “Flexibel Automatisieren Mit Drag&bot”. In: *Roboter in Der Intralogistik*. Stuttgart, Germany: Stuttgarter Produktionsakademie.

Neumann, M., S. Koralewski, and M. Beetz (Dec. 2020). *URoboSim – An Episodic Simulation Framework for Prospective Reasoning in Robotic Agents*. DOI: 10.48550/arXiv.2012.04442. arXiv: 2012.04442 [cs].

NEURA Robotics (2024). *Technologies*. Company Website.

Newell, A. (Apr. 1980). “Physical Symbol Systems”. In: *Cognitive Science* 4.2, pp. 135–183. ISSN: 0364-0213. DOI: 10.1016/S0364-0213(80)80015-2.

— (Jan. 1994). *Unified Theories of Cognition*. Reprint Edition. Cambridge, Mass: Harvard University Press. ISBN: 978-0-674-92101-6.

Ngo, R., L. Chan, and S. Mindermann (Oct. 2023). “The Alignment Problem from a Deep Learning Perspective”. In: *The Twelfth International Conference on Learning Representations*.

Ni, R. and A. H. Qureshi (Sept. 2022). “NTFields: Neural Time Fields for Physics-Informed Robot Motion Planning”. In: *The Eleventh International Conference on Learning Representations*.

— (Mar. 2024). “Physics-Informed Neural Motion Planning on Constraint Manifolds”. In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. Yokohama, Japan: IEEE. DOI: 0.1109/ICRA57147.2024.10610883.

Nichol, A., J. Achiam, and J. Schulman (Oct. 2018). “On First-Order Meta-Learning Algorithms”. In: *arXiv:1803.02999 [cs]*. arXiv: 1803.02999 [cs].

Nilsson, N. J. (2007). “The Physical Symbol System Hypothesis: Status and Prospects”. In: *50 Years of Artificial Intelligence: Essays Dedicated to the 50th Anniversary of Artificial Intelligence*. Ed. by M. Lungarella, F. Iida, J. Bongard, and R. Pfeifer. Berlin, Heidelberg: Springer, pp. 9–17. ISBN: 978-3-540-77296-5. DOI: 10.1007/978-3-540-77296-5_2.

Nofre, D., M. Priestley, and G. Alberts (2014). “When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960”. In: *Technology and Culture* 55.1, pp. 40–75. ISSN: 0040-165X. JSTOR: 24468397.

NVIDIA (2024a). *Isaac Sim*. https://developer.nvidia.com/isaac/sim. Company Website.

— (Mar. 2024b). *Project GR00T*. https://developer.nvidia.com/project-gr00t.

Nyga, D., S. Roy, R. Paul, D. Park, M. Pomarlan, M. Beetz, and N. Roy (Oct. 2018). “Grounding Robot Plans from Natural Language Instructions with Incomplete World Knowledge”. In: *Proceedings of The 2nd Conference on Robot Learning*. PMLR, pp. 714–723.

Nyholm, S. (July 2024). “AI, Robot Co-workers and Humans”. In: *The De Gruyter Handbook of Artificial Intelligence, Identity and Technology Studies*. De Gruyter, pp. 101–120. ISBN: 978-3-11-072175-1. DOI: 10.1515/9783110721751-006.

O’Neill, A. et al. (May 2024). “Open X-Embodiment: Robotic Learning Datasets and RT-X Models : Open X-Embodiment Collaboration0”. In: *2024 IEEE International*

Conference on Robotics and Automation (ICRA), pp. 6892–6903. DOI: 10.1109/ICRA57147.2024.10611477.

O'Sullivan, S., S. Leonard, A. Holzinger, C. Allen, F. Battaglia, N. Nevejans, F. W. B. van Leeuwen, M. I. Sajid, M. Friebe, H. Ashrafian, H. Heinsen, D. Wichmann, M. Hartnett, and A. G. Gallagher (2020). "Operational Framework and Training Standard Requirements for AI-empowered Robotic Surgery". In: *The International Journal of Medical Robotics and Computer Assisted Surgery* 16.5, e2020. ISSN: 1478-596X. DOI: 10.1002/rcs.2020.

Oistad, B. C., C. E. Sembroski, K. A. Gates, M. M. Krupp, M. R. Fraune, and S. Šabanović (2016). "Colleague or Tool? Interactivity Increases Positive Perceptions of and Willingness to Interact with a Robotic Co-worker". In: *Social Robotics*. Ed. by A. Agah, J.-J. Cabibihan, A. M. Howard, M. A. Salichs, and H. He. Cham: Springer International Publishing, pp. 774–785. ISBN: 978-3-319-47437-3. DOI: 10.1007/978-3-319-47437-3_76.

Okada, M., L. Rigazio, and T. Aoshima (June 2017). *Path Integral Networks: End-to-End Differentiable Optimal Control*. DOI: 10.48550/arXiv.1706.09597. arXiv: 1706.09597 [cs].

Olah, C. (Aug. 2015). *Understanding LSTM Networks*. Personal Blog.

Oleari, E., A. Leporini, D. Trojaniello, A. Sanna, U. Capitanio, F. Dehó, A. Larcher, F. Montorsi, A. Salonia, F. Setti, and R. Muradore (May 2019). "Enhancing Surgical Process Modeling for Artificial Intelligence Development in Robotics: The SARAS Case Study for Minimally Invasive Procedures". In: *2019 13th International Symposium on Medical Information and Communication Technology (ISMICT)*, pp. 1–6. DOI: 10.1109/ISMICT.2019.8743931.

Olivares-Alarcos, A., A. Andriella, S. Foix, and G. Alenyà (May 2023). "Robot Explanatory Narratives of Collaborative and Adaptive Experiences". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. London, United Kingdom: IEEE. DOI: 10.1109/ICRA48891.2023.10161359.

OpenAI (Sept. 2023). *ChatGPT*. https://chat.openai.com. Large Language Model.

— (May 2024a). *Hello GPT-4o*. https://openai.com/index/hello-gpt-4o/.

— (Aug. 2024b). *OpenAI API Reference*.

Oquab, M., T. Darcet, T. Moutakanni, H. V. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski (July 2023). "DINOv2: Learning Robust Visual Features without Supervision". In: *Transactions on Machine Learning Research*. ISSN: 2835-8856.

Ost, J., F. Mannan, N. Thuerey, J. Knodt, and F. Heide (June 2021). "Neural Scene Graphs for Dynamic Scenes". In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2855–2864. DOI: 10.1109/CVPR46437.2021.00288.

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens,

A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe (Apr. 2024). "Training Language Models to Follow Instructions with Human Feedback". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., pp. 27730–27744. ISBN: 978-1-7138-7108-8.

Pan, A., K. Bhatia, and J. Steinhardt (Oct. 2021). "The Effects of Reward Misspecification: Mapping and Mitigating Misaligned Models". In: *International Conference on Learning Representations*.

Pan, S. J. and Q. Yang (Oct. 2010). "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10, pp. 1345–1359. ISSN: 1041-4347. DOI: 10.1109/TKDE.2009.191.

Pandey, A. K. and R. Gelin (July 2018). "A Mass-Produced Sociable Humanoid Robot: Pepper: The First Machine of Its Kind". In: *IEEE Robotics & Automation Magazine* PP, pp. 1–1. DOI: 10.1109/MRA.2018.2833157.

Pangercic, D., B. Pitzer, M. Tenorth, and M. Beetz (Oct. 2012). "Semantic Object Maps for Robotic Housework - Representation, Acquisition and Use". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4644–4651. DOI: 10.1109/IROS.2012.6385603.

Pantano, M., T. Eiband, and D. Lee (Aug. 2022). "Capability-Based Frameworks for Industrial Robot Skills: A Survey". In: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*. Mexico City, Mexico: IEEE Press, pp. 2355–2362. DOI: 10.1109/CASE49997.2022.9926648.

Paraschos, A., C. Daniel, J. R. Peters, and G. Neumann (2013). "Probabilistic Movement Primitives". In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc.

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer (Oct. 2017). "Automatic Differentiation in PyTorch". In: *NIPS 2017 Workshop Autodiff*.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8026–8037.

Patacchiola, M., M. Sun, K. Hofmann, and R. E. Turner (Nov. 2023). "Comparing the Efficacy of Fine-Tuning and Meta-Learning for Few-Shot Policy Imitation". In: *Proceedings of The 2nd Conference on Lifelong Learning Agents*. PMLR, pp. 878–908.

Pateria, S., B. Subagdja, A.-h. Tan, and C. Quek (June 2021). "Hierarchical Reinforcement Learning: A Comprehensive Survey". In: *ACM Computing Surveys* 54.5, 109:1–109:35. ISSN: 0360-0300. DOI: 10.1145/3453160.

Patton, N., K. Rahmani, M. Missula, J. Biswas, and I. Dillig (Jan. 2024a). “Programming-by-Demonstration for Long-Horizon Robot Tasks”. In: *Proceedings of the ACM on Programming Languages* 8.POPL, 18:512–18:545. DOI: 10.1145/3632860.

— (Jan. 2024b). “Programming-by-Demonstration for Long-Horizon Robot Tasks”. In: *Programming-by-Demonstration for Long-Horizon Robot Tasks* 8.POPL, 18:512–18:545. DOI: 10.1145/3632860.

Paul, R. (Jan. 1977). “WAVE A Model Based Language for Manipulator Control”. In: *Industrial Robot: An International Journal* 4.1, pp. 10–17. ISSN: 0143-991X. DOI: 10.1108/eb004473.

Pavlick, E. (June 2023). “Symbols and Grounding in Large Language Models”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 381.2251, p. 20220041. DOI: 10.1098/rsta.2022.0041.

Paxton, C., V. Raman, G. D. Hager, and M. Kobilarov (Sept. 2017). “Combining Neural Networks and Tree Search for Task and Motion Planning in Challenging Environments”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6059–6066. DOI: 10.1109/IROS.2017.8206505.

Peller-Konrad, F., R. Kartmann, C. R. Dreher, A. Meixner, F. Reister, M. Grotz, and T. Asfour (June 2023). “A Memory System of a Robot Cognitive Architecture and Its Implementation in ArmarX”. In: *Robotics and Autonomous Systems* 164.C. ISSN: 0921-8890. DOI: 10.1016/j.robot.2023.104415.

Peres, R. S., X. Jia, J. Lee, K. Sun, A. W. Colombo, and J. Barata (2020). “Industrial Artificial Intelligence in Industry 4.0 - Systematic Review, Challenges and Outlook”. In: *IEEE Access* 8, pp. 220121–220139. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3042874.

Perlis, A. J. (July 1996). “Foreword”. In: *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass.: The MIT Press. ISBN: 978-0-262-51087-5.

Pervez, A. and D. Lee (Jan. 2018). “Learning Task-Parameterized Dynamic Movement Primitives Using Mixture of GMMs”. In: *Intelligent Service Robotics* 11.1, pp. 61–78. ISSN: 1861-2784. DOI: 10.1007/s11370-017-0235-8.

Petrov, A., T. A. Lamb, A. Paren, P. H. S. Torr, and A. Bibi (June 2024). *Universal In-Context Approximation By Prompting Fully Recurrent Models*. DOI: 10.48550/arXiv.2406.01424. arXiv: 2406.01424 [cs].

Petrucci, G., C. Ghidini, and M. Rospocher (2016). “Ontology Learning in the Deep”. In: *Knowledge Engineering and Knowledge Management*. Ed. by E. Blomqvist, P. Ciancarini, F. Poggi, and F. Vitali. Cham: Springer International Publishing, pp. 480–495. ISBN: 978-3-319-49004-5. DOI: 10.1007/978-3-319-49004-5_31.

Picklum, M. (2024). “Probabilistic Action Prospection Based on Experiences - Representation, Learning and Reasoning in Autonomous Robotic Agents”. PhD thesis. Universität Bremen. DOI: 10.26092/elib/2990.

Pierson, H. A. and M. S. Gashler (Aug. 2017). “Deep Learning in Robotics: A Review of Recent Research”. In: *Advanced Robotics* 31.16, pp. 821–835. ISSN: 0169-1864. DOI: 10.1080/01691864.2017.1365009.

Pitis, S. (Oct. 2023). “Failure Modes of Learning Reward Models for LLMs and Other Sequence Models”. In: *ICML 2023 Workshop The Many Facets of Preference-Based Learning*.

Pitt, D. (2022). “Mental Representation”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta and U. Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University.

Plattform Industrie 4.0 (2019). *2019 Progress Report: Shaping Industrie 4.0*. Tech. rep. Federal Ministry for Economic Affairs and Energy (BMWi).

Pogančić, M. V., A. Paulus, V. Musil, G. Martius, and M. Rolinek (Apr. 2020). “Differentiation of Blackbox Combinatorial Solvers”. In: *Eighth International Conference on Learning Representations*.

Pogodin, R., Y. Mehta, T. Lillicrap, and P. E. Latham (2021). “Towards Biologically Plausible Convolutional Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., pp. 13924–13936.

Poldrack, R. A., T. Lu, and G. Beguš (Apr. 2023). *AI-assisted Coding: Experiments with GPT-4*. DOI: 10.48550/arXiv.2304.13187. arXiv: 2304.13187 [cs].

Poole, H. H. (1989). “Robot Languages”. In: *Fundamentals of Robotics Engineering*. Ed. by H. H. Poole. Dordrecht: Springer Netherlands, pp. 249–270. ISBN: 978-94-011-7050-5. DOI: 10.1007/978-94-011-7050-5_10.

Prakash, U., A. Chollera, K. Khatwani, P. K. J., and T. Bodas (Jan. 2024). “Practical First-Order Bayesian Optimization Algorithms”. In: *Proceedings of the 7th Joint International Conference on Data Science & Management of Data (11th ACM IKDD CODS and 29th COMAD)*. CODS-COMAD ’24. New York, NY, USA: Association for Computing Machinery, pp. 173–181. ISBN: 979-8-4007-1634-8. DOI: 10.1145/3632410.3632418.

Presutti, V. and A. Gangemi (2016). “Dolce+D&S Ultralite and Its Main Ontology Design Patterns”. In: *Ontology Engineering with Ontology Design Patterns*. Ed. by P. Hitzler, A. Gangemi, K. Janowicz, A. Krisnadhi, and V. Presutti. Vol. 25. Studies on the Semantic Web. IOS Press, pp. 81–103. ISBN: 978-1-61499-676-7.

Prokhorenko, L., D. Klimov, D. Mishchenkov, and Y. Poduraev (May 2020). “Surgeon–Robot Interface Development Framework”. In: *Computers in Biology and Medicine* 120, p. 103717. ISSN: 0010-4825. DOI: 10.1016/j.compbiomed.2020.103717.

Pulvermüller, F., M. Garagnani, and T. Wennekers (2014). “Thinking in Circuits: Toward Neurobiological Explanation in Cognitive Neuroscience”. In: *Biological Cybernetics* 108.5, pp. 573–593. ISSN: 0340-1200. DOI: 10.1007/s00422-014-0603-9.

Pumacay, W., I. Singh, J. Duan, R. Krishna, J. Thomason, and D. Fox (July 2024). “THE COLOSSEUM: A Benchmark for Evaluating Generalization for Robotic Manipulation”. In: *Robotics: Science and Systems XX*. Delft, Netherlands. DOI: 10.15607/RSS.2024.XX.133.

Qiao, Y.-L., J. Liang, V. Koltun, and M. Lin (Nov. 2020). “Scalable Differentiable Physics for Learning and Control”. In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR, pp. 7847–7856.

Quintero, C. P., S. Li, M. K. Pan, W. P. Chan, H. Machiel Van der Loos, and E. Croft (Oct. 2018). “Robot Programming Through Augmented Trajectories in Augmented Reality”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1838–1844. DOI: 10.1109/IROS.2018.8593700.

Qureshi, A. H., J. Dong, A. Choe, and M. C. Yip (Oct. 2020). “Neural Manipulation Planning on Constraint Manifolds”. In: *IEEE Robotics and Automation Letters* 5.4, pp. 6089–6096. ISSN: 2377-3766. DOI: 10.1109/LRA.2020.3010220.

Racca, M., V. Kyrki, and M. Cakmak (Mar. 2020). “Interactive Tuning of Robot Program Parameters via Expected Divergence Maximization”. In: *Proceedings of the 2020 ACM/IEEE International Conference on Human-Robot Interaction*. HRI ’20. New York, NY, USA: Association for Computing Machinery, pp. 629–638. ISBN: 978-1-4503-6746-2. DOI: 10.1145/3319502.3374784.

Radford, A., J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever (July 2023). “Robust Speech Recognition via Large-Scale Weak Supervision”. In: *Proceedings of the 40th International Conference on Machine Learning*. Vol. 202. ICML’23. Honolulu, Hawaii, USA: JMLR.org, pp. 28492–28518.

Raible, J., C. Braun, and M. Huber (Sept. 2023a). “Automatic Path Planning for Robotic Grinding and Polishing Tasks Based on Point Cloud Slicing”. In: *ISR Europe 2023 - 56th International Symposium on Robotics*. Stuttgart, Germany: VDE Verlag.

Raible, J., O. Rettig, B. Alt, A. Yaman, I. Gauger, L. Biasi, S. Müller, D. Katic, M. Strand, and M. F. Huber (Aug. 2023b). “Artificial Neural Network Guided Compensation of Nonlinear Payload and Wear Effects for Industrial Robots”. In: *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*. Auckland, New Zealand: IEEE, pp. 1–8. ISBN: 979-8-3503-2069-5. DOI: 10.1109/CASE56687.2023.10260559.

Raji, I. D. and R. Dobbe (Dec. 2023). *Concrete Problems in AI Safety, Revisited*. DOI: 10.48550/arXiv.2401.10899. arXiv: 2401.10899 [cs].

Ramirez-Amaro, K., M. Beetz, and G. Cheng (June 2017). “Transferring Skills to Humanoid Robots by Extracting Semantic Representations from Observations of Human Activities”. In: *Artificial Intelligence*. Special Issue on AI and Robotics 247, pp. 95–118. ISSN: 0004-3702. DOI: 10.1016/j.artint.2015.08.009.

Ramirez-Amaro, K., T. Inamura, E. Dean-León, M. Beetz, and G. Cheng (Nov. 2014). “Bootstrapping Humanoid Robot Skills by Extracting Semantic Representations of Human-like Activities from Virtual Reality”. In: *2014 IEEE-RAS International Conference on Humanoid Robots*, pp. 438–443. DOI: 10.1109/HUMANOIDS.2014.7041398.

Rana, K., J. Haviland, S. Garg, J. Abou-Chakra, I. Reid, and N. Suenderhauf (Dec. 2023). “SayPlan: Grounding Large Language Models Using 3D Scene Graphs for Scalable Robot Task Planning”. In: *Proceedings of The 7th Conference on Robot Learning*. PMLR, pp. 23–72.

Ratliff, N., M. Zucker, J. A. Bagnell, and S. Srinivasa (May 2009). “CHOMP: Gradient Optimization Techniques for Efficient Motion Planning”. In: *2009 IEEE*

International Conference on Robotics and Automation (ICRA). Kobe, Japan: IEEE, pp. 489–494. ISBN: 978-1-4244-2788-8. DOI: 10.1109/ROBOT.2009.5152817.

Ratliff, N. D., J. Issac, D. Kappler, S. Birchfield, and D. Fox (July 2018). "Riemannian Motion Policies". In: *arXiv:1801.02854 [cs]*. arXiv: 1801.02854 [cs].

Reed, S., K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-maron, M. Giménez, Y. Sulsky, J. Kay, J. T. Springenberg, T. Eccles, J. Bruce, A. Razavi, A. Edwards, N. Heess, Y. Chen, R. Hadsell, O. Vinyals, M. Bordbar, and N. de Freitas (Aug. 2022). "A Generalist Agent". In: *Transactions on Machine Learning Research*. ISSN: 2835-8856.

Ren, T., S. Liu, A. Zeng, J. Lin, K. Li, H. Cao, J. Chen, X. Huang, Y. Chen, F. Yan, Z. Zeng, H. Zhang, F. Li, J. Yang, H. Li, Q. Jiang, and L. Zhang (Jan. 2024). *Grounded SAM: Assembling Open-World Models for Diverse Visual Tasks*. DOI: 10.48550/arXiv.2401.14159. arXiv: 2401.14159 [cs].

Revach, G., N. Shlezinger, X. Ni, A. L. Escoriza, R. J. G. van Sloun, and Y. C. Eldar (Jan. 2022). "KalmanNet: Neural Network Aided Kalman Filtering for Partially Known Dynamics". In: *IEEE Transactions on Signal Processing* 70, pp. 1532–1547. ISSN: 1053-587X. DOI: 10.1109/TSP.2022.3158588.

Ridnik, T., E. Ben-Baruch, A. Noy, and L. Zelnik-Manor (June 2021). "ImageNet-21K Pretraining for the Masses". In: *Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Ritter, F. E., F. Tehranchi, and J. D. Oury (May 2019). "ACT-R: A Cognitive Architecture for Modeling Cognition". In: *Wiley Interdisciplinary Reviews. Cognitive Science* 10.3, e1488. ISSN: 1939-5086. DOI: 10.1002/wcs.1488.

Rocktäschel, T. (Mar. 2018). "Combining Representation Learning with Logic for Language Processing". PhD thesis. UCL (University College London).

Rombach, R., A. Blattmann, D. Lorenz, P. Esser, and B. Ommer (2022). "High-Resolution Image Synthesis With Latent Diffusion Models". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695.

Rombouts, J. O., P. R. Roelfsema, and S. M. Bohte (July 2013). "Biologically Plausible Reinforcement Learning of Continuous Actions". In: *BMC Neuroscience* 14.1, P28. ISSN: 1471-2202. DOI: 10.1186/1471-2202-14-S1-P28.

Rosenbloom, P. S. (2023). "Rethinking the Physical Symbol Systems Hypothesis". In: *Artificial General Intelligence*. Ed. by P. Hammer, M. Alirezaie, and C. Strannegård. Cham: Springer Nature Switzerland, pp. 207–216. ISBN: 978-3-031-33469-6. DOI: 10.1007/978-3-031-33469-6_21.

Rosenthal, S. and M. Veloso (July 2011). "Modeling Humans as Observation Providers Using POMDPs". In: *2011 RO-MAN*, pp. 53–58. DOI: 10.1109/ROMAN.2011.6005272.

Ross, T., D. Zimmerer, A. Vemuri, F. Isensee, M. Wiesenfarth, S. Bodenstedt, F. Both, P. Kessler, M. Wagner, B. Müller, H. Kenngott, S. Speidel, A. Kopp-Schneider, K. Maier-Hein, and L. Maier-Hein (June 2018). "Exploiting the Potential of Unlabeled Endoscopic Video Data with Self-Supervised Learning". In: *International*

Journal of Computer Assisted Radiology and Surgery 13.6, pp. 925–933. ISSN: 1861-6429. DOI: 10.1007/s11548-018-1772-0.

Rougier, N. P. (Mar. 2009). “Implicit and Explicit Representations”. In: *Neural Networks*. What It Means to Communicate 22.2, pp. 155–160. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2009.01.008.

Ruder, S. (Feb. 2019). “Neural Transfer Learning for Natural Language Processing”. PhD thesis. National University of Ireland.

Rudin, C. (May 2019). “Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead”. In: *Nature Machine Intelligence* 1.5, pp. 206–215. ISSN: 2522-5839. DOI: 10.1038/s42256-019-0048-x.

Rueß, H. and S. Burton (2022). *Safe AI - How Is This Possible?* Tech. rep. Fraunhofer Institute for Cognitive Systems IKS.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (Oct. 1986). “Learning Representations by Back-Propagating Errors”. In: *Nature* 323.6088, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

Russell, S. and P. Norvig (May 2021). *Artificial Intelligence: A Modern Approach*. 4th ed. Harlow: Pearson. ISBN: 978-1-292-40113-3.

Rusu, R. B. and S. Cousins (May 2011). “3D Is Here: Point Cloud Library (PCL)”. In: *2011 IEEE International Conference on Robotics and Automation*, pp. 1–4. DOI: 10.1109/ICRA.2011.5980567.

Sacerdoti, E. (Sept. 1975). “The Nonlinear Nature of Plans”. In: *International Joint Conference on Artificial Intelligence*.

Safronov, E., M. Colledanchise, and L. Natale (Oct. 2020). “Task Planning with Belief Behavior Trees”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6870–6877. DOI: 10.1109/IROS45743.2020.9341562.

Salemi, A. and H. Zamani (July 2024). “Evaluating Retrieval Quality in Retrieval-Augmented Generation”. In: *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’24. New York, NY, USA: Association for Computing Machinery, pp. 2395–2400. ISBN: 979-8-4007-0431-4. DOI: 10.1145/3626772.3657957.

Salvador, S. and P. Chan (Oct. 2007). “Toward Accurate Dynamic Time Warping in Linear Time and Space”. In: *Intelligent Data Analysis* 11.5, pp. 561–580. ISSN: 1088-467X.

Saveriano, M., F. J. Abu-Dakka, A. Kramberger, and L. Peternel (Nov. 2023). “Dynamic Movement Primitives in Robotics: A Tutorial Survey”. In: *The International Journal of Robotics Research* 42.13, pp. 1133–1184. ISSN: 0278-3649. DOI: 10.1177/02783649231201196.

Scao, T. L. et al. (June 2023). *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. DOI: 10.48550/arXiv.2211.05100. arXiv: 2211.05100 `[cs]`.

Schaal, S. and C. G. Atkeson (Nov. 1998). “Constructive Incremental Learning from Only Local Information”. In: *Neural Computation* 10.8, pp. 2047–2084. ISSN: 1530-888X. DOI: 10.1162/089976698300016963.

Schaal, S. (2006). “Dynamic Movement Primitives - A Framework for Motor Control in Humans and Humanoid Robotics”. In: *Adaptive Motion of Animals and Machines*. Ed. by H. Kimura, K. Tsuchiya, A. Ishiguro, and H. Witte. Springer, pp. 261–280. ISBN: 978-4-431-31381-6. DOI: 10.1007/4-431-31381-8_23.

Scheide, E., G. Best, and G. A. Hollinger (May 2021). “Behavior Tree Learning for Robotic Task Planning through Monte Carlo DAG Search over a Formal Grammar”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. Xi&apos;an, China: IEEE Press, pp. 4837–4843. DOI: 10.1109/ICRA48506.2021.9561027.

Schepman, A. and P. Rodway (Aug. 2023). “The General Attitudes towards Artificial Intelligence Scale (GAAIS): Confirmatory Validation and Associations with Personality, Corporate Distrust, and General Trust”. In: *International Journal of Human–Computer Interaction* 39.13, pp. 2724–2741. ISSN: 1044-7318. DOI: 10.1080/10447318.2022.2085400.

Scherlis, A., K. Sachan, A. S. Jermyn, J. Benton, and B. Shlegeris (July 2023). *Polysemanticity and Capacity in Neural Networks*. DOI: 10.48550/arXiv.2210.01892. arXiv: 2210.01892 [cs].

Schmidt-Rohr, S. R., R. Jäkel, and G. Dirschl (2013). *ArtiMinds Robot Programming Suite*. ArtiMinds Robotics GmbH.

Schöppenthau, F., F. Patzer, B. Schnebel, K. Watson, N. Baryschnikov, B. Obst, Y. Chauhan, D. Kaever, T. Usländer, and P. Kulkarni (Jan. 2023). “Building a Digital Manufacturing as a Service Ecosystem for Catena-X”. In: *Sensors* 23.17, p. 7396. ISSN: 1424-8220. DOI: 10.3390/s23177396.

Schramm, W. (1954). “How Communication Works”. In: *Process and Effects of Communication*, pp. 3–26.

Schulman, J., J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel (June 2013). “Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization”. In: *Robotics: Science and Systems IX*. Vol. 09. ISBN: 978-981-07-3937-9.

Schultheis, A., B. Alt, S. Bast, A. Guldner, D. Jilg, D. Katic, J. Mundorf, T. Schlagenhauf, S. Weber, R. Bergmann, S. Bergweiler, L. Creutz, G. Dartmann, L. Malburg, S. Naumann, M. Rezapour, and M. Ruskowski (Sept. 2024). “EASY: Energy-Efficient Analysis and Control Processes in the Dynamic Edge-Cloud Continuum for Industrial Manufacturing”. In: *KI - Künstliche Intelligenz*. ISSN: 1610-1987. DOI: 10.1007/s13218-024-00868-3.

Sejnowski, T. J. (Feb. 2023). “Large Language Models and the Reverse Turing Test”. In: *Neural Computation* 35.3, pp. 309–342. ISSN: 0899-7667. DOI: 10.1162/neco_a_01563.

Seker, M. Y., M. Imre, J. Piater, and E. Ugur (June 2019). “Conditional Neural Movement Primitives”. In: *Robotics: Science and Systems*. Vol. 15. ISBN: 978-0-9923747-5-4.

Shah, D., B. Osiński, B. Ichter, and S. Levine (Mar. 2023). “LM-Nav: Robotic Navigation with Large Pre-Trained Models of Language, Vision, and Action”. In: *Proceedings of The 6th Conference on Robot Learning*. PMLR, pp. 492–504.

Shah, R., V. Varma, R. Kumar, M. Phuong, V. Krakovna, J. Uesato, and Z. Kenton (Nov. 2022). *Goal Misgeneralization: Why Correct Specifications Aren’t Enough For Correct Goals*. DOI: 10.48550/arXiv.2210.01790. arXiv: 2210.01790 [cs].

Shaw, M. (Nov. 1990). “Prospects for an Engineering Discipline of Software”. In: *IEEE Software* 7.6, pp. 15–24. ISSN: 1937-4194. DOI: 10.1109/52.60586.

Shi, Z., J. Wei, and Y. Liang (Nov. 2023). “Provable Guarantees for Neural Networks via Gradient Feature Learning”. In: *Thirty-Seventh Conference on Neural Information Processing Systems*.

Shimano, B. (Nov. 1979). “VAL: A Versatile Robot Programming and Control System”. In: *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society’s Third International Applications Conference, 1979*. Pp. 878–883. DOI: 10.1109/CMPSAC.1979.762620.

Shimbun, N. K. (Dec. 1988). *Poka-Yoke: Improving Product Quality by Preventing Defects*. Cambridge, Massachusetts: Taylor & Francis Inc. ISBN: 978-0-915299-31-7.

Shysheya, A., J. F. Bronskill, M. Patacchiola, S. Nowozin, and R. E. Turner (Sept. 2022). “FiT: Parameter Efficient Few-shot Transfer Learning for Personalized and Federated Image Classification”. In: *The Eleventh International Conference on Learning Representations*.

Siaterlis, G., N. Nikolakis, K. Alexopoulos, and S. Makris (Jan. 2022). “Adoption of AI in EU Manufacturing. Gaps and Challenges”. In: *Proceedings of the 33rd International DAAAM Symposium 2022*, pp. 0547–0550. ISBN: 978-3-902734-36-5. DOI: 10.2507/33rd.daaam.proceedings.077.

Silver, T., R. Chitnis, J. Tenenbaum, L. P. Kaelbling, and T. Lozano-Pérez (Sept. 2021). “Learning Symbolic Operators for Task and Motion Planning”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3182–3189. DOI: 10.1109/IROS51168.2021.9635941.

Singh, I., V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg (May 2023). “ProgPrompt: Generating Situated Robot Task Plans Using Large Language Models”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530. DOI: 10.1109/ICRA48891.2023.10161317.

Skalse, J., N. H. R. Howe, D. Krasheninnikov, and D. Krueger (Apr. 2024). “Defining and Characterizing Reward Hacking”. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS ’22. Red Hook, NY, USA: Curran Associates Inc., pp. 9460–9471. ISBN: 978-1-7138-7108-8.

Smith, D. R. (Sept. 1985). “Top-down Synthesis of Divide-and-Conquer Algorithms”. In: *Artificial Intelligence* 27.1, pp. 43–96. ISSN: 0004-3702. DOI: 10.1016/0004-3702(85)90083-9.

Smits, R. (Oct. 2020). *Orocos Kinematics and Dynamics Library*. Open Robot Control Software.

Smolensky, P. (Mar. 1988). “On the Proper Treatment of Connectionism”. In: *Behavioral and Brain Sciences* 11.1, pp. 1–23. ISSN: 1469-1825, 0140-525X. DOI: 10.1017/S0140525X00052432.

Sobania, D., D. Schweim, and F. Rothlauf (Feb. 2023). “A Comprehensive Survey on Program Synthesis With Evolutionary Algorithms”. In: *IEEE Transactions on Evolutionary Computation* 27.1, pp. 82–97. ISSN: 1941-0026. DOI: 10.1109/TEVC.2022.3162324.

Sofianidis, G., J. M. Rozanec, D. Mladenic, and D. Kyriazis (2021). “A Review of Explainable Artificial Intelligence in Manufacturing”. In: *CoRR* abs/2107.02295. arXiv: 2107.02295.

Sohn, J., S. Lee, and S. Yoo (2016). “Amortised Deep Parameter Optimisation of GPGPU Work Group Size for OpenCV”. In: *Search Based Software Engineering*. Ed. by F. Sarro and K. Deb. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 211–217. ISBN: 978-3-319-47106-8. DOI: 10.1007/978-3-319-47106-8_14.

Sohn, K. and O. Kwon (Apr. 2020). “Technology Acceptance Theories and Factors Influencing Artificial Intelligence-based Intelligent Products”. In: *Telematics and Informatics* 47, p. 101324. ISSN: 0736-5853. DOI: 10.1016/j.tele.2019.101324.

Srinivas, A., A. Jabri, P. Abbeel, S. Levine, and C. Finn (July 2018). “Universal Planning Networks: Learning Generalizable Representations for Visuomotor Control”. In: *Proceedings of the 35th International Conference on Machine Learning*. PMLR, pp. 4732–4741.

Srinivas, N., A. Krause, S. Kakade, and M. Seeger (June 2010). “Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Madison, WI, USA: Omnipress, pp. 1015–1022. ISBN: 978-1-60558-907-7.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15, pp. 1929–1958.

Staffell, I. and R. Green (June 2014). “How Does Wind Farm Performance Decline with Age?” In: *Renewable Energy* 66, pp. 775–786. ISSN: 0960-1481. DOI: 10.1016/j.renene.2013.10.041.

Staron, M., S. Abrahão, G. Lewis, H. Muccini, and C. Honnenahalli (Sept. 2024). “Bringing Software Engineering Discipline to the Development of AI-Enabled Systems”. In: *IEEE Software* 41.5, pp. 79–82. ISSN: 1937-4194. DOI: 10.1109/MS.2024.3408388.

Stöckl, F., M. Strand, S. Müller, M. Huber, J. Raible, C. Braun, D. Katic, B. Alt, and H. Merkt (July 2023). “Autonomous Surface Grinding of Wind Turbine Blades”. In: *Intelligent Autonomous Systems 18*. Ed. by S.-G. Lee, J. An, N. Y. Chong, M.

Strand, and J. H. Kim. Cham: Springer Nature Switzerland, pp. 451–457. ISBN: 978-3-031-44981-9. DOI: 10.1007/978-3-031-44981-9_38.

Stone, A., T. Xiao, Y. Lu, K. Gopalakrishnan, K.-H. Lee, Q. Vuong, P. Wohlhart, S. Kirmani, B. Zitkovich, F. Xia, C. Finn, and K. Hausman (Aug. 2023). "Open-World Object Manipulation Using Pre-Trained Vision-Language Models". In: *7th Annual Conference on Robot Learning*.

Strazdas, D., J. Hintz, A.-M. Felßberg, and A. Al-Hamadi (2020). "Robots and Wizards: An Investigation Into Natural Human–Robot Interaction". In: *IEEE Access* 8, pp. 207635–207642. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3037724.

Strudel, R., R. Garcia, I. Laptev, and C. Schmid (2021). "Segmenter: Transformer for Semantic Segmentation". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7262–7272.

Stulp, F., G. Raiola, A. Hoarau, S. Ivaldi, and O. Sigaud (Oct. 2013). "Learning Compact Parameterized Skills with a Single Regression". In: *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pp. 417–422. DOI: 10.1109/HUMANOIDS.2013.7030008.

Stulp, F. and S. Schaal (Oct. 2011). "Hierarchical Reinforcement Learning with Movement Primitives". In: *2011 11th IEEE-RAS International Conference on Humanoid Robots*, pp. 231–238. DOI: 10.1109/Humanoids.2011.6100841.

Su, L., X. Zuo, R. Li, X. Wang, H. Zhao, and B. Huang (Oct. 2023). *A Systematic Review for Transformer-based Long-term Series Forecasting*. DOI: 10.48550/arXiv.2310.20218. arXiv: 2310.20218 [cs].

Summers, P. D. (Jan. 1977). "A Methodology for LISP Program Construction from Examples". In: *Journal of the ACM* 24.1, pp. 161–175. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321992.322002.

Sun, Q., Y. Liu, T.-S. Chua, and B. Schiele (2019). "Meta-Transfer Learning for Few-Shot Learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 403–412.

Sun, R. (2017). "The CLARION Cognitive Architecture: Toward a Comprehensive Theory of the Mind". In: *The Oxford Handbook of Cognitive Science*. New York, NY, US: Oxford University Press, pp. 117–133. ISBN: 978-0-19-984219-3 978-0-19-984417-3.

Sünderhauf, N., O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford, and P. Corke (Apr. 2018). "The Limits and Potentials of Deep Learning for Robotics". In: *The International Journal of Robotics Research* 37.4-5, pp. 405–420. ISSN: 0278-3649. DOI: 10.1177/0278364918770733.

Sussman, G. J. (Oct. 2005). "Why Programming Is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas". In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. New York, NY, USA: Association for

Computing Machinery, p. 6. ISBN: 978-1-59593-193-1. DOI: 10.1145/1094855.1094860.

Sutton, R. S. (Mar. 2019). *The Bitter Lesson*.

Swanson, K., E. Wu, A. Zhang, A. A. Alizadeh, and J. Zou (Apr. 2023). "From Patterns to Patients: Advances in Clinical Machine Learning for Cancer Diagnosis, Prognosis, and Treatment". In: *Cell* 186.8, pp. 1772–1791. ISSN: 0092-8674, 1097-4172. DOI: 10.1016/j.cell.2023.01.035.

Tate, A. (1977). "Generating Project Networks". In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence*. Vol. 2. Cambridge, USA: Morgan Kaufmann, pp. 888–893.

Taylor, R., P. Summers, and J. Meyer (Sept. 1982). "AML: A Manufacturing Language". In: *The International Journal of Robotics Research* 1.3, pp. 19–41. ISSN: 0278-3649. DOI: 10.1177/027836498200100302.

Tellex, S. A., T. F. Kollar, S. R. Dickerson, M. R. Walter, A. Banerjee, S. Teller, and N. Roy (2011). "Approaching the Symbol Grounding Problem with Probabilistic Graphical Models". In: *AI Magazine* 32.4, pp. 64–76. ISSN: 0738-4602. DOI: 10.1609/aimag.v32i4.2384.

Tenorth, M. and M. Beetz (Apr. 2013). "KnowRob: A Knowledge Processing Infrastructure for Cognition-Enabled Robots". In: *The International Journal of Robotics Research* 32.5, pp. 566–590. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364913481635.

— (June 2017). "Representations for Robot Knowledge in the KnowRob Framework". In: *Artificial Intelligence*. Special Issue on AI and Robotics 247, pp. 151–169. ISSN: 0004-3702. DOI: 10.1016/j.artint.2015.05.010.

Theis, S., S. Jentzsch, F. Deligiannaki, C. Berro, A. P. Raulf, and C. Bruder (2023). "Requirements for Explainability and Acceptance of Artificial Intelligence in Collaborative Work". In: *Artificial Intelligence in HCI*. Ed. by H. Degen and S. Ntoa. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, pp. 355–380. ISBN: 978-3-031-35891-3. DOI: 10.1007/978-3-031-35891-3_22.

Thomason, J., J. Sinapov, M. Svetlik, P. Stone, and R. Mooney (July 2016). "Learning Multi-Modal Grounded Linguistic Semantics by Playing "I Spy"". In: *International Joint Conference on Artificial Intelligence*.

Thompson, N. C., K. Greenewald, K. Lee, and G. F. Manso (July 2022). "The Computational Limits of Deep Learning". In: *Ninth Workshop on Computing within Limits*. arXiv. DOI: 10.48550/arXiv.2007.05558. arXiv: 2007.05558 `[cs, stat]`.

Thrun, S. and L. Pratt (1998). "Learning to Learn: Introduction and Overview". In: *Learning to Learn*. Ed. by S. Thrun and L. Pratt. Boston, MA: Springer US, pp. 3–17. ISBN: 978-1-4615-5529-2. DOI: 10.1007/978-1-4615-5529-2_1.

Tong, X. and W. Lei (Jan. 2017). "A Systematic Analysis of Functional Safety Certification Practices in Industrial Robot Software Development". In: *MATEC Web of Conferences* 100, p. 02011. DOI: 10.1051/matecconf/201710002011.

Tosatto, S., G. Chalvatzaki, and J. Peters (May 2021). "Contextual Latent-Movements Off-Policy Optimization for Robotic Manipulation Skills". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10815–10821. DOI: 10.1109/ICRA48506.2021.9561870.

Toussaint, M. (July 2015). "Logic-Geometric Programming: An Optimization-Based Approach to Combined Task and Motion Planning". In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, pp. 1930–1936. ISBN: 978-1-57735-738-4.

Toussaint, M., K. Allen, K. Smith, and J. Tenenbaum (June 2018). "Differentiable Physics and Stable Modes for Tool-Use and Manipulation Planning". In: *Robotics: Science and Systems XIV*. Vol. 14. ISBN: 978-0-9923747-4-7.

Toussaint, M. and M. Lopes (May 2017). "Multi-Bound Tree Search for Logic-Geometric Programming in Cooperative Manipulation Domains". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4044–4051. DOI: 10.1109/ICRA.2017.7989464.

Touvron, H., T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample (Feb. 2023). *LLaMA: Open and Efficient Foundation Language Models*. DOI: 10.48550/arXiv.2302.13971. arXiv: 2302.13971 [cs].

Trafton, J. G., L. M. Hiatt, A. M. Harrison, F. P. Tamborello, S. S. Khemlani, and A. C. Schultz (Feb. 2013). "ACT-R/E: An Embodied Cognitive Architecture for Human-Robot Interaction". In: *Journal of Human-Robot Interaction* 2.1, pp. 30–55. DOI: 10.5898/JHRI.2.1.Trafton.

Truhn, D., J. S. Reis-Filho, and J. N. Kather (Dec. 2023). "Large Language Models Should Be Used as Scientific Reasoning Engines, Not Knowledge Databases". In: *Nature Medicine* 29.12, pp. 2983–2984. ISSN: 1546-170X. DOI: 10.1038/s41591-023-02594-z.

Ude, A., B. Nemec, T. Petrić, and J. Morimoto (May 2014). "Orientation in Cartesian Space Dynamic Movement Primitives". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2997–3004. DOI: 10.1109/ICRA.2014.6907291.

Universal Robots (2018a). *PolyScope Manual*.

— (Dec. 2018b). *The URScript Programming Language*. Tech. rep. Universal Robots.

Uren, V. and J. S. Edwards (Feb. 2023). "Technology Readiness and the Organizational Journey towards AI Adoption: An Empirical Study". In: *International Journal of Information Management* 68, p. 102588. ISSN: 0268-4012. DOI: 10.1016/j.ijinfomgt.2022.102588.

Urieli, D., P. MacAlpine, S. Kalyanakrishnan, Y. Bentor, and P. Stone (2011). "On Optimizing Interdependent Skills: A Case Study in Simulated 3D Humanoid Robot Soccer". In: *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*. Taipei, Taiwan.

Vahrenkamp, N., M. Wächter, M. Kröhnert, K. Welke, and T. Asfour (Apr. 2015). "The Robot Software Framework ArmarX". In: *it - Information Technology* 57.2, pp. 99–111. ISSN: 2196-7032. DOI: 10.1515/itit-2014-1066.

van Krieken, E., E. Acar, and F. van Harmelen (Jan. 2022). "Analyzing Differentiable Fuzzy Logic Operators". In: *Artificial Intelligence* 302, p. 103602. ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103602.

Vandegar, M. (June 2020). "Differentiable Surrogate Models to Solve Nonlinear Inverse Problems". MA thesis. Liège, Belgium: Université de Liège. Chap. Université de Liège.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (Dec. 2017). "Attention Is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., pp. 6000–6010. ISBN: 978-1-5108-6096-4.

Vemprala, S., R. Bonatti, A. Bucker, and A. Kapoor (July 2023). *ChatGPT for Robotics: Design Principles and Model Abilities*. DOI: 10.48550/arXiv.2306.17582. arXiv: 2306.17582 `[cs]`.

Verma, D., N. Winovich, L. Ruthotto, and B. v. B. Waanders (Feb. 2024). *Neural Network Approaches for Parameterized Optimal Control*. DOI: 10.48550/arXiv.2402.10033. arXiv: 2402.10033 `[math]`.

Vernon, D. (May 2022). "Cognitive Architectures". In: *Cognitive Robotics*. Ed. by A. Cangelosi and M. Asada. Intelligent Robotics and Autonomous Agents. Cambridge, Massachusetts: MIT Press. ISBN: 978-0-262-36932-9.

Vernon, D., M. Beetz, and G. Sandini (July 2015). "Prospection in Cognition: The Case for Joint Episodic-Procedural Memory in Cognitive Robotics". In: *Frontiers in Robotics and AI* 2. ISSN: 2296-9144. DOI: 10.3389/frobt.2015.00019.

Vernon, D., C. von Hofsten, and L. Fadiga (2011). "The iCub Cognitive Architecture". In: *A Roadmap for Cognitive Development in Humanoid Robots*. Ed. by D. Vernon, C. von Hofsten, and L. Fadiga. Cognitive Systems Monographs. Berlin, Heidelberg: Springer, pp. 121–153. ISBN: 978-3-642-16904-5. DOI: 10.1007/978-3-642-16904-5_7.

Villani, V., F. Pini, F. Leali, C. Secchi, and C. Fantuzzi (2018). "Survey on Human-Robot Interaction for Robot Programming in Industrial Applications". In: *IFAC-PapersOnLine* 51.11, pp. 66–71. ISSN: 24058963. DOI: 10.1016/j.ifacol.2018.08.236.

Vollmer, A.-L. and N. J. Hemion (2018). "A User Study on Robot Skill Learning Without a Cost Function: Optimization of Dynamic Movement Primitives via Naive User Feedback". In: *Frontiers in Robotics and AI* 5. ISSN: 2296-9144.

von Rueden, L., S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, M. Walczak, J. Pfrommer, A. Pick, R. Ramamurthy, J. Garcke, C. Bauckhage, and J. Schuecker (2021). "Informed Machine Learning - A Taxonomy and Survey of Integrating Prior Knowledge into Learning Systems". In: *IEEE Transactions* *on Knowledge and Data Engineering*, pp. 1–1. ISSN: 1558-2191. DOI: 10.1109/TKDE.2021.3079836.

Vu, B., T. Migimatsu, and J. Bohg (May 2024). *COAST: Constraints and Streams for Task and Motion Planning*. DOI: 10.48550/arXiv.2405.08572. arXiv: 2405.08572 [cs].

W3C (Feb. 2014). *RDF 1.1 Turtle: Terse RDF Triple Language*. Tech. rep. World Wide Web Consortium.

Wake, N., A. Kanehira, K. Sasabuchi, J. Takamatsu, and K. Ikeuchi (2023). "ChatGPT Empowered Long-Step Robot Control in Various Environments: A Case Application". In: *IEEE Access* 11, pp. 95060–95078. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3310935. arXiv: 2304.03893 [cs].

Wang, J. and Y. Chen (Nov. 2023). "A Review on Code Generation with LLMs: Application and Evaluation". In: *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pp. 284–289. DOI: 10.1109/MedAI59581.2023.00044.

Wang, J., C. Lan, C. Liu, Y. Ouyang, T. Qin, W. Lu, Y. Chen, W. Zeng, and P. S. Yu (Aug. 2023a). "Generalizing to Unseen Domains: A Survey on Domain Generalization". In: *IEEE Transactions on Knowledge and Data Engineering* 35.8, pp. 8052–8072. ISSN: 1558-2191. DOI: 10.1109/TKDE.2022.3178128.

Wang, L., Y. Ling, Z. Yuan, M. Shridhar, C. Bao, Y. Qin, B. Wang, H. Xu, and X. Wang (Oct. 2023b). "GenSim: Generating Robotic Simulation Tasks via Large Language Models". In: *The Twelfth International Conference on Learning Representations*.

Wang, T., J. Chen, Q. Jia, S. Wang, R. Fang, H. Wang, Z. Gao, C. Xie, C. Xu, J. Dai, Y. Liu, J. Wu, S. Ding, L. Li, Z. Huang, X. Deng, T. Yu, G. Ma, H. Xiao, Z. Chen, D. Xiang, Y. Wang, Y. Zhu, Y. Xiao, J. Wang, Y. Wang, S. Ding, J. Huang, J. Xu, Y. Tayier, Z. Hu, Y. Gao, C. Zheng, Y. Ye, Y. Li, L. Wan, X. Jiang, Y. Wang, S. Cheng, Z. Song, X. Tang, X. Xu, N. Zhang, H. Chen, Y. E. Jiang, and W. Zhou (Jan. 2024a). *Weaver: Foundation Models for Creative Writing*. DOI: 10.48550/arXiv.2401.17268. arXiv: 2401.17268 [cs].

Wang, X., Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji (July 2024b). "Executable Code Actions Elicit Better LLM Agents". In: *Proceedings of the 41st International Conference on Machine Learning*. PMLR, pp. 50208–50232.

Wang, Y., Q. Yao, J. T. Kwok, and L. M. Ni (June 2020). "Generalizing from a Few Examples: A Survey on Few-shot Learning". In: *ACM Computing Surveys* 53.3, 63:1–63:34. ISSN: 0360-0300. DOI: 10.1145/3386252.

Webster, M., C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, K. Dautenhahn, and J. Saez-Pons (Apr. 2016). "Toward Reliable Autonomous Robotic Assistants Through Formal Verification: A Case Study". In: *IEEE Transactions on Human-Machine Systems* 46.2, pp. 186–196. ISSN: 2168-2305. DOI: 10.1109/THMS.2015.2425139.

Wei, C. and K. V. Hindriks (2013). "An Agent-Based Cognitive Robot Architecture". In: *Programming Multi-Agent Systems*. Ed. by M. Dastani, J. F. Hübner, and

B. Logan. Berlin, Heidelberg: Springer, pp. 54–71. ISBN: 978-3-642-38700-5. DOI: 10.1007/978-3-642-38700-5_4.

Weintrop, D., A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin (2018). “Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices”. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI ’18*. Montreal QC, Canada: ACM Press, pp. 1–12. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173940.

Weißmann, M., S. Bedenk, C. Buckl, and A. Knoll (2011). “Model Checking Industrial Robot Systems”. In: *Model Checking Software*. Ed. by A. Groce and M. Musuvathi. Berlin, Heidelberg: Springer, pp. 161–176. ISBN: 978-3-642-22306-8. DOI: 10.1007/978-3-642-22306-8_11.

Wen, H., X. Chen, G. Papagiannis, C. Hu, and Y. Li (May 2021). “End-to-End Semi-supervised Learning for Differentiable Particle Filters”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5825–5831. DOI: 10.1109/ICRA48506.2021.9561889.

White, W. T. (May 2023). *Introducing Intrinsic Flowstate*.

Wielemaker, J., T. Schrijvers, M. Triska, and T. Lager (2012). “SWI-prolog”. In: *Theory and Practice of Logic Programming* 12.1-2, pp. 67–96. ISSN: 1471-0684.

Wiethof, C. and E. Bittner (Dec. 2021). “Hybrid Intelligence – Combining the Human in the Loop with the Computer in the Loop: A Systematic Literature Review”. In: *ICIS 2021 Proceedings*.

Williams, R. J. (1986). “Inverting a Connectionist Network Mapping by Backpropagation of Error”. In: *Proc. of 8th Annual Conference of the Cognitive Science Society*, pp. 859–865.

Wolf, Y., N. Wies, D. Shteyman, B. Rothberg, Y. Levine, and A. Shashua (Feb. 2024). *Tradeoffs Between Alignment and Helpfulness in Language Models*. DOI: 10.48550/arXiv.2401.16332. arXiv: 2401.16332 [cs].

Wong, E. and J. Z. Kolter (2017). “Neural Network Inversion beyond Gradient Descent”. In: *OPTML 2017*, p. 5.

Wu, F., W. Weimer, M. Harman, Y. Jia, and J. Krinke (July 2015). “Deep Parameter Optimisation”. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. GECCO ’15. New York, NY, USA: Association for Computing Machinery, pp. 1375–1382. ISBN: 978-1-4503-3472-3. DOI: 10.1145/2739480.2754648.

Wu, J. J. (June 2024). “An Exploratory Study of V-Model in Building ML-Enabled Software: A Systems Engineering Perspective”. In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*. CAIN ’24. New York, NY, USA: Association for Computing Machinery, pp. 30–40. ISBN: 979-8-4007-0591-5. DOI: 10.1145/3644815.3644951.

Wu, J., R. Antonova, A. Kan, M. Lepert, A. Zeng, S. Song, J. Bohg, S. Rusinkiewicz, and T. Funkhouser (Dec. 2023a). “TidyBot: Personalized Robot Assistance with Large Language Models”. In: *Autonomous Robots* 47.8, pp. 1087–1102. ISSN: 1573-7527. DOI: 10.1007/s10514-023-10139-z.

Wu, Y.-L., H.-H. Shuai, Z.-R. Tam, and H.-Y. Chiu (Oct. 2021). “Gradient Normalization for Generative Adversarial Networks”. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6353–6362. DOI: 10.1109/ICCV48922.2021.00631.

Wu, S., T. Jakab, C. Rupprecht, and A. Vedaldi (Oct. 2023b). “DOVE: Learning Deformable 3D Objects by Watching Videos”. In: *International Journal of Computer Vision* 131.10, pp. 2623–2634. ISSN: 1573-1405. DOI: 10.1007/s11263-023-01819-5.

Wu, X., Y.-L. Li, J. Sun, and C. Lu (Dec. 2023c). “Symbol-LLM: Leverage Language Models for Symbolic System in Visual Human Activity Reasoning”. In: *Advances in Neural Information Processing Systems* 36, pp. 29680–29691.

Xie, S., L. Sun, Z. Wang, and G. Chen (May 2022). “A Speedup Method for Solving the Inverse Kinematics Problem of Robotic Manipulators”. In: *International Journal of Advanced Robotic Systems* 19.3, p. 17298806221104602. ISSN: 1729-8806. DOI: 10.1177/17298806221104602.

Xu, X., Y. Lu, B. Vogel-Heuser, and L. Wang (Oct. 2021). “Industry 4.0 and Industry 5.0—Inception, Conception and Perception”. In: *Journal of Manufacturing Systems* 61, pp. 530–535. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2021.10.006.

Xu, Z., Z. Shi, J. Wei, F. Mu, Y. Li, and Y. Liang (Oct. 2023). “Towards Few-Shot Adaptation of Foundation Models via Multitask Finetuning”. In: *The Twelfth International Conference on Learning Representations*.

Yampolskiy, R. V. (May 2022). “On the Controllability of Artificial Intelligence: An Analysis of Limitations”. In: *Journal of Cyber Security and Mobility*, pp. 321–404. ISSN: 2245-4578. DOI: 10.13052/jcsm2245-1439.1132.

Yang, D., F. Jiang, W. Wu, X. Fang, and M. Cao (June 2023). “Low-Complexity Acoustic Echo Cancellation with Neural Kalman Filtering”. In: *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1–5. DOI: 10.1109/ICASSP49357.2023.10096597.

Yang, K., J. Liu, J. Wu, C. Yang, Y. Fung, S. Li, Z. Huang, X. Cao, X. Wang, H. Ji, and C. Zhai (Mar. 2024). “If LLM Is the Wizard, Then Code Is the Wand: A Survey on How Code Empowers Large Language Models to Serve as Intelligent Agents”. In: *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Yang, Q., A. Steinfeld, C. Rosé, and J. Zimmerman (Apr. 2020a). “Re-Examining Whether, Why, and How Human-AI Interaction Is Uniquely Difficult to Design”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. New York, NY, USA: Association for Computing Machinery, pp. 1–13. ISBN: 978-1-4503-6708-0. DOI: 10.1145/3313831.3376301.

Yang, Q., J. Suh, N.-C. Chen, and G. Ramos (June 2018). “Grounding Interactive Machine Learning Tool Design in How Non-Experts Actually Build Models”. In: *Proceedings of the 2018 Designing Interactive Systems Conference*. DIS ’18. New York, NY, USA: Association for Computing Machinery, pp. 573–584. ISBN: 978-1-4503-5198-0. DOI: 10.1145/3196709.3196729.

Yang, S., X. Yu, and Y. Zhou (June 2020b). "LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example". In: *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*, pp. 98–101. doi: 10.1109/IWECAI50956.2020.00027.

Yeh, C.-K., W.-C. Wu, W.-J. Ko, and Y.-C. F. Wang (Feb. 2017). "Learning Deep Latent Spaces for Multi-Label Classification". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI'17. San Francisco, California, USA: AAAI Press, pp. 2838–2844.

Yildirim, I. and L. A. Paul (May 2024). "From Task Structures to World Models: What Do LLMs Know?" In: *Trends in Cognitive Sciences* 28.5, pp. 404–415. issn: 1364-6613, 1879-307X. doi: 10.1016/j.tics.2024.02.008.

Yonetani, R., T. Taniai, M. Barekatain, M. Nishimura, and A. Kanezaki (July 2021). "Path Planning Using Neural A* Search". In: *Proceedings of the 38th International Conference on Machine Learning*. PMLR, pp. 12029–12039.

Yosinski, J., J. Clune, Y. Bengio, and H. Lipson (Dec. 2014). "How Transferable Are Features in Deep Neural Networks?" In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Cambridge, MA, USA: MIT Press, pp. 3320–3328.

Yu, T., C. Finn, S. Dasari, A. Xie, T. Zhang, P. Abbeel, and S. Levine (2018). "One-Shot Imitation from Observing Humans via Domain-Adaptive Meta-Learning". In: *Robotics: Science and Systems XIV, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, June 26-30, 2018*. Ed. by H. Kress-Gazit, S. S. Srinivasa, T. Howard, and N. Atanasov. doi: 10.15607/RSS.2018.XIV.002.

Yu, W. (July 2022). "Retrieval-Augmented Generation across Heterogeneous Knowledge". In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Student Research Workshop*. Ed. by D. Ippolito, L. H. Li, M. L. Pacheco, D. Chen, and N. Xue. Hybrid: Seattle, Washington + Online: Association for Computational Linguistics, pp. 52–58. doi: 10.18653/v1/2022.naacl-srw.7.

Yu, W., C. Zhu, Z. Li, Z. Hu, Q. Wang, H. Ji, and M. Jiang (Nov. 2022). "A Survey of Knowledge-enhanced Text Generation". In: *ACM Comput. Surv.* 54.11s, 227:1–227:38. issn: 0360-0300. doi: 10.1145/3512467.

Yuan, J., B. Zhang, X. Yan, B. Shi, T. Chen, Y. Li, and Y. Qiao (Dec. 2023). "AD-PT: Autonomous Driving Pre-Training with Large-scale Point Cloud Dataset". In: *Advances in Neural Information Processing Systems* 36, pp. 47914–47933.

Zeng, F., W. Gan, Y. Wang, N. Liu, and P. S. Yu (Nov. 2023). *Large Language Models for Robotics: A Survey*. doi: 10.48550/arXiv.2311.07226. arXiv: 2311.07226 [cs].

Zhang, B., Z. Tian, Q. Tang, X. Chu, X. Wei, C. Shen, and Y. Liu (Dec. 2022). "SegViT: Semantic Segmentation with Plain Vision Transformers". In: *Advances in Neural Information Processing Systems* 35, pp. 4971–4982.

Zheng, B., S. Verma, J. Zhou, I. W. Tsang, and F. Chen (2022). "Imitation Learning: Progress, Taxonomies and Challenges". In: *IEEE Transactions on Neural Networks* *and Learning Systems*, pp. 1–16. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2022.3213246.

Zhong, W., R. Cui, Y. Guo, Y. Liang, S. Lu, Y. Wang, A. Saied, W. Chen, and N. Duan (June 2024). "AGIEval: A Human-Centric Benchmark for Evaluating Foundation Models". In: *Findings of the Association for Computational Linguistics: NAACL 2024*. Ed. by K. Duh, H. Gomez, and S. Bethard. Mexico City, Mexico: Association for Computational Linguistics, pp. 2299–2314. DOI: 10.18653/v1/2024.findings-naacl.149.

Zhong, Y., B. Shirinzadeh, X. Yuan, G. Alici, and J. Smith (2006). "A Cellular Neural Network for Deformable Object Modelling". In: *Information Technology For Balanced Manufacturing Systems*. Ed. by W. Shen. Boston, MA: Springer US, pp. 329–336. ISBN: 978-0-387-36594-7. DOI: 10.1007/978-0-387-36594-7_35.

Zhou, Y., J. Gao, and T. Asfour (June 2020). "Movement Primitive Learning and Generalization: Using Mixture Density Networks". In: *IEEE Robotics & Automation Magazine* 27.2, pp. 22–32. ISSN: 1070-9932, 1558-223X. DOI: 10.1109/MRA.2020.2980591.

Zhuang, F., Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He (Jan. 2021). "A Comprehensive Survey on Transfer Learning". In: *Proceedings of the IEEE* 109.1, pp. 43–76. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.3004555.

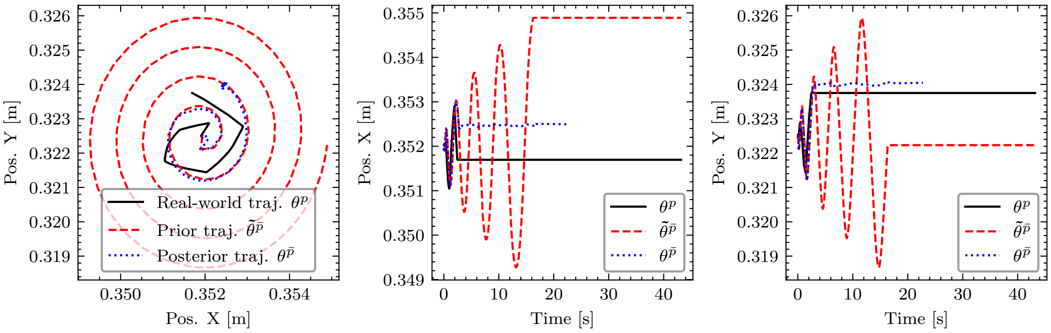
Zitkovich, B., T. Yu, S. Xu, P. Xu, T. Xiao, F. Xia, J. Wu, P. Wohlhart, S. Welker, A. Wahid, Q. Vuong, V. Vanhoucke, H. Tran, R. Soricut, A. Singh, J. Singh, P. Sermanet, P. R. Sanketi, G. Salazar, M. S. Ryoo, K. Reymann, K. Rao, K. Pertsch, I. Mordatch, H. Michalewski, Y. Lu, S. Levine, L. Lee, T.-W. E. Lee, I. Leal, Y. Kuang, D. Kalashnikov, R. Julian, N. J. Joshi, A. Irpan, B. Ichter, J. Hsu, A. Herzog, K. Hausman, K. Gopalakrishnan, C. Fu, P. Florence, C. Finn, K. A. Dubey, D. Driess, T. Ding, K. M. Choromanski, X. Chen, Y. Chebotar, J. Carbajal, N. Brown, A. Brohan, M. G. Arenas, and K. Han (Aug. 2023). "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control". In: *7th Annual Conference on Robot Learning*.

Appendix

APPENDIX A

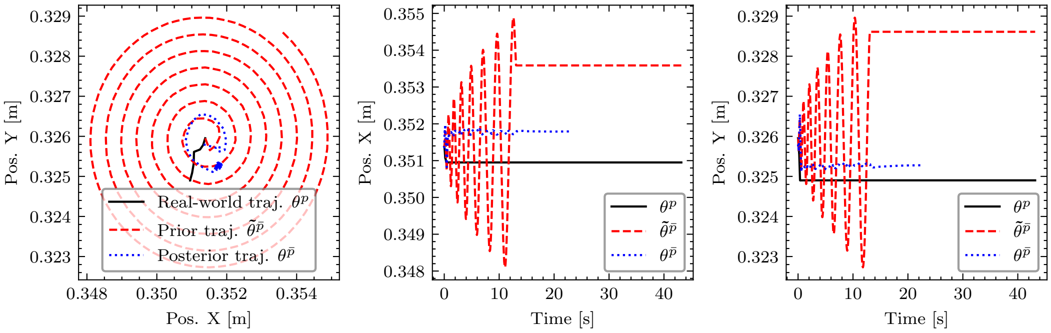
Differentiable Shadow Programs: Predicted Trajectories

Figure 2.9 shows exemplary prior, posterior and ground-truth trajectories for a shadow skill $\bar{p}$ for a `Spiral Search Relative` ARTM skill. $\bar{p}$ has 6 stacked GRU layers, a hidden size of 256, and a dropout rate of 0.2. It was trained on 10.000 simulated spiral searches over 250 epochs with a learning rate $\alpha = 1e - 4$ and a batch size of 16, and finetuned on 2717 real-world spiral search trajectories collected on the Experiment setup described in Experiment 4.3.2 with a learning rate of $\alpha = 1e - 5$.

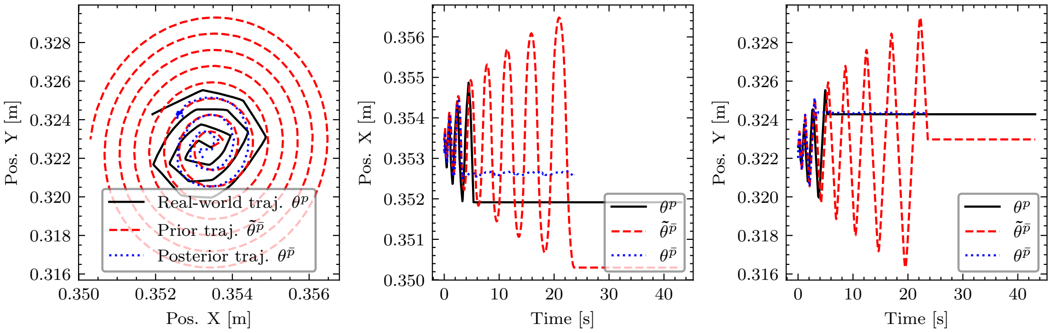


(a) $\bar{p}$ correctly predicts the time and location at which the hole is found. The regularizing effect of the prior causes it to mispredict the final end-effector position by 0.6 mm along the X axis.

Figure A.1: Ground-truth (solid), prior (red) and posterior (blue) Cartesian end-effector trajectories predicted by shadow skill $\bar{p}$ for a spiral search task.



(b) $\bar{p}$ is able to handle the degenerate case in which the hole is immediately found, and no spiral motion is executed. As the end-effector positions along the very short trajectory are constrained by the shape of the hole, the ground-truth trajectory resembles noise.



(c) The ABB-specific resampling of the spiral trajectory into linear segments sometimes induces a phase offset to the prior. $\bar{p}$ learned to partially compensate these phase shifts, but they remain a source of prediction errors.

Figure A.1: Ground-truth (solid), prior (red) and posterior (blue) Cartesian end-effector trajectories predicted by shadow skill $\bar{p}$ for a spiral search task.

APPENDIX B

Dialogue Transcripts

```
MetaWizard2: What task do you want me to help with?
Programmer: Grinding.
MetaWizard2: Please specify the material of the workpiece.
Programmer: Fiberglass.
MetaWizard2: I've set the material of the workpiece to fiberglass.
MetaWizard2: Please specify the type of the workpiece.
Programmer: Rotor blade.
MetaWizard2: I've set the type of the workpiece to rotor blade.
MetaWizard2: What tool should the robot use?
Programmer: OnRobot sander.
MetaWizard2: Tool OnRobot sander chosen. I've inserted the 3D model at the default position relative to the robot.
MetaWizard2: What is the desired finish?
Programmer: Rough.
MetaWizard2: The currently loaded RPS program does not contain a Path Loader template. Creating one.
# Program is simulated and executed.
MetaWizard2: Is another run required?
Programmer: Yes.
MetaWizard2: Should I change the finish type?
Programmer: Yes.
# Program is simulated and executed.
MetaWizard2: Is another run required?
Programmer: No.
```

Listing B.1: Transcript of a dialogue between MetaWizard2 and a human programmer for a wind turbine blade refurbishing task (see Experiment 4.2.4.1).

```
MetaWizard2: What task do you want me to help with?
Programmer: Surface treatment.
MetaWizard2: To what value should I set the cutting depth?
Programmer: Two millimeters.
MetaWizard2: I've set the material of the workpiece to 2.0 millimeter.
MetaWizard2: Please specify the material of the workpiece. Viable options known to me are listed below.
Programmer: It's aluminium.
MetaWizard2: I've set the material of the workpiece to aluminium alloys.
MetaWizard2: Please specify the shape of the workpiece. Viable options known to me are listed below.
Programmer: It's a rotational mold.
MetaWizard2: I've set the shape of the workpiece to rotational mold.
MetaWizard2: I am inserting the workpiece into the RPS.
```

```
MetaWizard2: What tool should the robot use? Viable tools that work for this task are listed below.
Programmer:  Use an orbital sander.
MetaWizard2: Tool Mirka orbital sander chosen. I've inserted the 3D model at the default position relative to the robot.
MetaWizard2: Choose cutting speed for the grinder. Tell me minimal, medium or maximal and I will calculate appropriate values based on the material of the workpiece and the grinding tool in use.
Programmer:  Let's do medium.
MetaWizard2: Adding template: Path Loader.
MetaWizard2: Adding template: Move to Point.
# Program is simulated and executed.
```

Listing B.2: Transcript of a dialogue between MetaWizard2 and a human programmer for sanding a rotational mold (see Experiment 4.2.4.2).

```
MetaWizardLLM: What task do you want me to help with?
Programmer: Pick up the small gear and insert it onto the shaft. After insertion, release the object and depart to a position a few millimetres above the shaft.
MetaWizardLLM: small_gear chosen.
MetaWizardLLM: Looking for 'tip of the shaft'.
MetaWizardLLM: What part do you want to choose? the lime part or the magenta part ?
Programmer: The magenta part.
MetaWizardLLM: Chosen part: magenta
# Program is simulated and executed.
```

Listing B.3: Transcript of a dialogue between MetaWizardLLM and a human programmer for a gear assembly task (see Experiment 4.3.2).

APPENDIX C

Prompts

```
<data>
    <message role="system">
        Your task is to extract numerical values from spoken language. Your answer may only contain the numerical value and the unit identifier that the user is most likely referring to.
        Do not convert units into different ones.
        Do not use exponents, e.g. 10^3 or 10e3 instead of 1000.
        Do not write out quantities like thousand, million or billion instead of numbers.
    </message>
    <message role="user">
        <content type="text">
            one point three meters.
        </content>
    </message>
    <message role="assistant">
        <content type="text">
            1.3m
        </content>
    </message>
    <message role="user">
        <content type="text">
            The acceleration is three thousand and ninety six point 1 four meters per second squared.
        </content>
    </message>
    <message role="assistant">
        <content type="text">
            3,096.14m/s^2
        </content>
    </message>
    <message role="user">
        <content type="text">
            It weighs one million three hundred and five grams.
        </content>
    </message>
    <message role="assistant">
        <content type="text">
            1,000,305g
        </content>
    </message>
    <message role="user">
        <content type="text">
            {user_input}
        </content>
    </message>
</data>
```

Listing C.1: Prompt for extracting numerical parameters from Llama3.

```
<data>
    <message role="system">
        You are an assistant that has to guess what a user means based on an input and different options.
        The input semantically relates to the options, with one of the options being a match.
        Only return the full, exact string of the matching element
        Return nothing but the full string of this element.!
        Only return 'unknown' if no option fits.
    </message>
    <message role="user">
        <content type="text">
            'mexico' ['africa', 'europe', 'north america', 'south america', 'asia', 'australia']
        </content>
    </message>
    <message role="assistant">
        <content type="text">
            'north america'
        </content>
    </message>
    <message role="user">
        <content type="text">
            'material' ['text', 'chair', 'flower']
        </content>
    </message>
    <message role="assistant">
        <content type="text">
            'unknown'
        </content>
    </message>
    <message role="user">
        <content type="text">
            'round' ['cube', 'triangle', 'circle', 'prisma', 'banana']
        </content>
    </message>
    <message role="assistant">
        <content type="text">
            'circle'
        </content>
    </message>
    <message role="user">
        <content type="text">
            'car part' ['concrete', 'steering wheel', 'tombola']
        </content>
    </message>
    <message role="assistant">
        <content type="text">
            'steering wheel'
        </content>
    </message>
    <message role="user">
        <content type="text">
            '{user_input}' {options}
        </content>
    </message>
</data>
```

Listing C.2: Prompt for parsing chosen options with Llama3.

```
<data>
    <message role='system'>
        You are an assistant for planning multi-step robot actions for a robotic manipulator equipped with a simple gripper.
        You will be able to call Python functions representing your possible actions.
        Every task is solvable using existing functions and the context given to you. The output should always be a simple list of function calls, including their parameters.
        Output only Python code. Do not output textual descriptions or any text other than Python code. Avoid nested function calls, and create local variables instead.

        natural_language_scene_description = (
            "In this scene, you can see one small gear, one large gear and a motor assembly next to each other."
            "Both gears are moveable objects. The motor assembly has two features where the gears can be inserted."
            "The small gear belongs onto the tip of the shaft, the large one belongs in the gear hole."
            "The small gear must be inserted before the large gear.")

        moveable_objects = [
            small_gear
            large_gear
        ]

        stationary_objects = [
            motor_assembly
        ]

#These implementations are hidden
def cad_point(object_identifer: str, prompt: str) -> am_control_plugin_python.data.common_data.Point3d:
        """
        Returns a point from the CAD-Environment based on a natural language description of the point/feature.
        @param object_identifier: string, identifier of the object.
        @param prompt: string, natural language description of the point/feature.
        @return: Point3d that's being referred to in the prompt.
        """
        pass

def object_location(object_identifier: str) -> am_control_plugin_python.data.common_data.Point3d:
        """
        Returns the location of an object from the CAD-Environment given the identifier of the object.
        @param object_identifier: string, identifier of the object.
        @return: Point3d, the location of the object in world coordinates.
        """
        pass

def move_linear_relative_contact(offset_x: float, offset_y: float, offset_z: float):
        """
        Move on a relative, linear Cartesian path until a minium force in Z is measured.
        @param offset_x: Relative motion along the X direction (in local tool coordinates)
        @param offset_y: Relative motion along the Y direction (in local tool coordinates)
        @param offset_z: Relative motion along the Z direction (in local tool coordinates)
        """
        pass

def move_to_point(point_to: am_control_plugin_python.data.common_data.Point3d, offset_x: float, offset_y: float, offset_z: float):
        """
        Move on a planned joint motion to a specified cartesian pose
        The caller must ensure that the goal pose is reachable for the robot.
        @param point_to: Target point.
        @param offset_x: Relative positioning offset to point_to in mm
        @param offset_y: Relative positioning offset to point_to in mm
        @param offset_z: Relative positioning offset to point_to in mm
        """
        pass

def insert(*args, **kwargs):
        """
        Inserts insertable into receptacle. This can refer to all kinds of peg-in-hole tasks.
        The caller must ensure that the insertable and receptacle are aligned before calling insert.
        The caller must ensure that the insertable is grasped by the robot before calling insert.
        The caller must ensure that the robot is positioned above the receptacle before calling insert.
        @param insertable: object to be inserted
        @param receptacle: object that receives the inserted object
        """
        pass

def open_gripper(*args, **kwargs):
        """
        Opens the gripper.
        """
        pass

def close_gripper(*args, **kwargs):
        """
        Closes the gripper.
        """
        pass

def grasp_relative(offset_z: float):
        """
        Grasp an object. The grasping motion is defined relative to the current position of the object. The caller must ensure the robot is positioned above the target object before calling grasp_relative.

        Robot opens gripper, moves down and closes gripper, grasping the object.
        After that, the robot moves back to the original position.
        @param offset_z: Offset along the Z direction, in mm. An offset_z of 20 means that the robot moves down by 20 mm to grasp the object.
        """
        pass

    </message>
        Move the large gear from point a to point b.
    <message role='assistant'>
        point_a = cad_point("large_gear", "The point that is most likely point a")
        move_to_point(point_a, 20)
        grasp_relative(20)
        point_b = cad_point("motor_assembly", "point b as specified in the scene context")
        move_to_point_(point_b, 10)
    </message>
    <message role='user'>pick up the small Gear and insert it onto the shaft
    after insertion release the object and depart to a position a few
    millimetres above the shaft</message>
</data>
```

Listing C.3: Generated prompt for gear assembly with MetaWizardLLM.

APPENDIX D

List of Publications

This work includes elements derived from previously published research in international conferences, workshops, books and patents. The relevant prior work is cited in the corresponding sections of this dissertation. This appendix presents a comprehensive overview.

Conference Papers

Alt, B., F. Aumann, L. Gienger, F. Jordan, D. Katic, R. Jäkel, and B. Graf (2020). “Modulare, Datengetriebene Roboterprogrammierung Für Die Lösung Komplexer Handhabungsaufgaben in Alltagsumgebungen”. In: *AAL-Kongress 2020*. Berlin: VDE Verlag, pp. 17–22. ISBN: 978-3-8007-5342-0.

Alt, B., J. Dvorak, D. Katic, R. Jäkel, M. Beetz, and G. Lanza (Jan. 2024a). “BANSAI: Towards Bridging the AI Adoption Gap in Industrial Robotics with Neurosymbolic Programming”. In: *Procedia CIRP*. Vol. 130. Póvoa de Varzim, Portugal: Elsevier B.V., pp. 532–537. DOI: 10.1016/j.procir.2024.10.125. arXiv: 2404.13652 [cs].

Alt, B., D. Katic, R. Jäkel, and M. Beetz (Oct. 2022b). “Heuristic-Free Optimization of Force-Controlled Robot Search Strategies in Stochastic Environments”. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Kyoto, Japan: IEEE, pp. 8887–8893. ISBN: 978-1-6654-7927-1. DOI: 10.1109/IROS47612.2022.9982093.

Alt, B., D. Katic, R. Jäkel, A. K. Bozcuoglu, and M. Beetz (May 2021). “Robot Program Parameter Inference via Differentiable Shadow Program Inversion”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. Xi’an, China: IEEE, pp. 4672–4678. ISBN: 978-1-7281-9077-8. DOI: 10.1109/ICRA48506.2021.9561206.

Alt, B., F. K. Kenfack, A. Haidu, D. Katic, R. Jäkel, and M. Beetz (Sept. 2023). “Knowledge-Driven Robot Program Synthesis from Human VR Demonstrations”. In: *Proceedings of the 20th International Conference on Principles of Knowledge*

Representation and Reasoning. Rhodes, Greece: IJCAI, pp. 34–43. isbn: 978-1-956792-02-7. doi: 10.24963/kr.2023/4.

Alt, B., U. Keßner, A. Taranovic, D. Katic, A. Hermann, R. Jäkel, and G. Neumann (Mar. 2024b). "Domain-Specific Fine-Tuning of Large Language Models for Interactive Robot Programming". In: *European Robotics Forum 2024*. Ed. by C. Secchi and L. Marconi. Vol. 32. Springer Proceedings in Advanced Robotics. Rimini, Italy: Springer Nature Switzerland, pp. 274–279. isbn: 978-3-031-76424-0. doi: 10.1007/978-3-031-76424-0_49. arXiv: 2312.13905 [cs].

Alt, B., C. Kienle, D. Katic, R. Jäkel, and M. Beetz (May 2025). "Shadow Program Inversion with Differentiable Planning: A Framework for Unified Robot Program Parameter and Trajectory Optimization". In: *2025 IEEE International Conference on Robotics and Automation (ICRA)*. Atlanta, USA: IEEE. doi: 10.48550/arXiv.2409.08678. arXiv: 2409.08678 [cs].

Alt, B., F. Stöckl, S. Müller, C. Braun, J. Raible, S. Alhasan, O. Rettig, L. Ringle, D. Katic, R. Jäkel, M. Beetz, M. Strand, and M. F. Huber (May 2024c). "RoboGrind: Intuitive and Interactive Surface Treatment with Industrial Robots". In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. Yokohama, Japan: IEEE, pp. 1–8. isbn: 979-8-3503-8457-4. doi: 10.1109/ICRA57147.2024.10611143. arXiv: 2402.16542 [cs].

Alt, B., J. Zahn, C. Kienle, J. Dvorak, M. May, D. Katic, R. Jäkel, T. Kopp, M. Beetz, and G. Lanza (Apr. 2024d). "Human-AI Interaction in Industrial Robotics: Design and Empirical Evaluation of a User Interface for Explainable AI-Based Robot Program Optimization". In: *Procedia CIRP*. Vol. 130. Póvoa de Varzim, Portugal: Elsevier B.V., pp. 591–596. doi: 10.1016/j.procir.2024.10.134. arXiv: 2404.19349 [cs].

Kienle, C., B. Alt, D. Katic, and R. Jäkel (May 2025). "QueryCAD: Grounded Question Answering for CAD Models". In: *2025 IEEE International Conference on Robotics and Automation (ICRA)*. Atlanta, USA: IEEE. doi: 10.48550/arXiv.2409.08704. arXiv: 2409.08704 [cs].

Raible, J., O. Rettig, B. Alt, A. Yaman, I. Gauger, L. Biasi, S. Müller, D. Katic, M. Strand, and M. F. Huber (Aug. 2023). "Artificial Neural Network Guided Compensation of Nonlinear Payload and Wear Effects for Industrial Robots". In: *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*. Auckland, New Zealand: IEEE, pp. 1–8. isbn: 979-8-3503-2069-5. doi: 10.1109/CASE56687.2023.10260559.

Stöckl, F., M. Strand, S. Müller, M. Huber, J. Raible, C. Braun, D. Katic, B. Alt, and H. Merkt (July 2023). "Autonomous Surface Grinding of Wind Turbine Blades". In: *Intelligent Autonomous Systems 18*. Ed. by S.-G. Lee, J. An, N. Y. Chong, M. Strand, and J. H. Kim. Cham: Springer Nature Switzerland, pp. 451–457. isbn: 978-3-031-44981-9. doi: 10.1007/978-3-031-44981-9_38.

Journal Papers

Schultheis, A., B. Alt, S. Bast, A. Guldner, D. Jilg, D. Katic, J. Mundorf, T. Schlagenhauf, S. Weber, R. Bergmann, S. Bergweiler, L. Creutz, G. Dartmann, L. Malburg, S. Naumann, M. Rezapour, and M. Ruskowski (Sept. 2024). “EASY: Energy-Efficient Analysis and Control Processes in the Dynamic Edge-Cloud Continuum for Industrial Manufacturing”. In: *KI - Künstliche Intelligenz*. ISSN: 1610-1987. DOI: 10.1007/s13218-024-00868-3.

Patents

Alt, B., R. Jäkel, and D. Katic (Feb. 2022a). “Method and System for Determining Optimized Program Parameters for a Robot Program”. WO2022022784A1 (Karlsruhe).

List of Abbreviations

AI artificial intelligence. i, 4, 5, 7–13, 15, 17–20, 22, 23, 34, 69, 70, 73, 76–79, 82, 84, 86, 87, 111, 126, 146, 148, 150, 152, 186, 205, 206, 212, 213, 217, 218, 221–237, 239–241, 243–245, 247, 248

API application programming interface. 30, 78, 81, 182, 189, 191, 207–209, 211, 213, 222, 231, 234, 245

ARTM ArtiMinds Robot Task Model. 30, 31, 33, 38, 44, 48–51, 53, 55, 69, 72, 73, 82, 88, 94, 97, 101, 102, 114, 120, 131, 134, 139, 148, 159, 160, 168–170, 172, 173, 176, 195, 209, 211, 212, 231, 244–246, 303

AST abstract syntax tree. 209

BANSAI Bridging the AI Adoption Gap with Neurosymbolic AI. 224–226, 229, 231–237, 240, 245, 246

BT Behavior Tree. 24, 33, 71

CAD computer-aided design. 207, 209–213, 215, 217, 218, 231, 240

CaP Code as Policies. 78, 182, 206, 212, 213

CNMP Conditional Neural Movement Primitive. 24, 67, 68, 79

CNN convolutional neural network. 57, 74, 105, 185

CPL CRAM Plan Language. 78, 168, 170, 185, 234

CRAM Cognitive Robot Abstract Machine. 26, 78, 83, 168, 171, 184, 185, 233, 234

CSP Constraint Satisfaction Problem. 69

DAG directed acyclic graph. 33, 227

DCG differentiable computation graph. 13, 20, 25, 35–37, 40, 41, 43, 44, 46–50, 52, 53, 57, 64, 77, 79, 80, 82, 84, 85, 90, 91, 105–107, 131, 132, 142, 144, 231, 239

DGPMP2 Differentiable Gaussian Process Motion Planning. 55–59, 61, 142

DGPMP2-ND Differentiable Gaussian Process Motion Planning for N-DoF Manipulators. 13, 55, 58–64, 79, 100, 131–133, 135–140, 143, 144, 147, 231, 236, 240

DH Denavit-Hartenberg. 45

DL deep learning. 5–8, 10, 23, 224

DMP Dynamic Movement Primitive. 18, 25, 26, 28–31, 35, 39, 66–68, 79, 95, 96, 131

DNN deep neural network. 17, 40–42, 53, 67, 71, 80, 82, 105, 141, 142, 210

DoF degree of freedom. xv, 45, 46, 55, 57, 58, 66, 67, 82, 141, 150

DSL domain-specific language. 78, 213

DUL DOLCE+DnS Ultralite. 155, 158, 159

EOS end-of-sequence. 36, 37, 52, 92

FK forward kinematics. 36, 42, 44–46, 59–61, 90

FOMAML First-Order MAML. 118, 119, 126, 144

GMM Gaussian Mixture Model. 66, 122

GP Gaussian process. 55–59, 61, 62, 66–68, 133, 136, 138, 141, 142

GPMP2 Gaussian Process Motion Planning 2. 55, 57, 59

GPU graphics processing unit. 85, 91, 119, 229, 231, 245

GRU Gated Recurrent Unit. 39, 40, 52, 53, 109, 129, 147, 244, 303

GUI graphical user interface. 65, 72, 84, 146, 201, 203, 212, 222, 227, 229, 231, 242

HTN Hierarchical Task Network. 29, 69, 70, 81

HTTP Hypertext Transfer Protocol. 231

IAI Institute for Artificial Intelligence. v, 174, 178

IDE integrated development environment. 55, 148, 172, 173, 191, 211, 218, 231

IK inverse kinematics. 19, 36, 42, 44–47, 105

IRI Internationalized Resource Identifier. 158

KB knowledge base. 154–157, 187–189, 191, 192, 199–201, 243

KR&R Knowledge Representation & Reasoning. 12, 13, 78, 153, 154, 167, 171, 181, 184, 188, 189, 191, 196, 203, 206, 234

L-BFGS limited-memory Broyden–Fletcher–Goldfarb–Shanno. 46, 47

LAR Learning and Analytics for Robots. 148, 231, 245, 246

LfD Learning from Demonstration. 70, 107

LLM large language model. 76, 78, 79, 84, 152, 181–183, 204–218, 231, 233, 240, 244, 247

LM Levenberg-Marquardt. 46, 104

LRP Layer-wise Relevance Propagation. 229

MAE mean absolute error. 101, 102, 228

MAML Model-Agnostic Meta Learning. 118, 119, 126

MAP maximum a posteriori. 55–58

MES manufacturing execution system. 65

ML machine learning. 111, 112, 124, 141, 146, 215, 227, 230, 231, 233

MP Movement Primitive. 24, 34, 66–69, 71, 74, 107

MSE mean squared error. 37, 102, 119, 197

MuTT Multimodal Trajectory Transformer. 147, 148, 246

NEEM Narrative-Enabled Episodic Memory. 163–167, 174, 175, 184, 185, 233

NLP natural language processing. 189, 198, 217

NNII Neural Network Iterative Inversion. 105, 106, 124

NRP Neurosymbolic Robot Program. i, iii, 11–13, 15, 16, 20, 23–31, 33, 34, 36, 37, 39, 45, 47–52, 54, 55, 58, 62, 64, 77–86, 88–90, 102, 103, 107–113, 117–119, 129, 144, 146, 147, 151–153, 187, 215, 217, 218, 222, 225, 226, 231, 236, 239–241, 243, 245–247

NSGA-II Non-dominated Sorting Genetic Algorithm. 114–117, 122

ODE ordinary differential equation. 18, 37, 67, 79

OEM original equipment manufacturer. 65

OWL Web Ontology Language. 157

PbD Programming by Demonstration. 84, 202, 235, 242

PCB printed circuit board. 7, 32, 39, 50, 93, 95, 97, 120

PDDL Planning Domain Definition Language. 69, 71, 184

PGH Physical Grounding Hypothesis. 17

PID proportional-integral-derivative. 87

PLC programmable logic controller. 223

POMDP partially observable Markov decision process. 74, 203

ProDMP Probabilistic Dynamic Movement Primitive. 67, 68

ProMP Probabilistic Movement Primitive. 67, 68

PSSH Physical Symbol System Hypothesis. 17

QA quality assurance. 138, 139, 143, 147

RAG Retrieval-Augmented Generation. 128, 151, 206, 207, 209, 210, 213, 215, 218, 240, 244, 247

RDF Resource Description Framework. 154–158

RL Reinforcement Learning. 5, 67, 68, 74, 75, 81, 105–108, 141, 145

RMP Riemannian Motion Policies. 68

RPS Robot Programming Suite. 55, 131, 148, 172, 173, 191, 192, 211, 231, 245

RRT Rapidly Exploring Random Tree. 69

SDF signed distance field. 59, 135, 140, 143, 144

SELU scaled exponential linear unit. 40, 41

SGD stochastic gradient descent. 91, 105, 118, 126, 144

SOMA Socio-Physical Model of Activities. 155, 158–161, 163, 164, 166, 180, 188, 189, 234

SPI Shadow Program Inversion. 12, 13, 88–103, 105–117, 121–124, 128–133, 135, 140, 143–148, 150, 206, 222, 225–231, 236, 240, 242, 244–247

SPI-DP Shadow Program Inversion with Differentiable Planning. 130, 132–136, 138–144, 147, 240, 246

TAMP Task and Motion Planning. 24, 70, 71, 81, 142, 184, 185, 247

TCP tool center point. 35, 111, 156, 170

TF transformation. 155, 156, 163, 166, 171, 183

THT through-hole technology. 7, 93, 97, 120

UR Universal Robots. 72

URDF Unified Robot Description Format. 45, 155

UX user experience. 230

VLA vision-language-action. 76, 77, 125, 127, 128, 183, 204, 206, 218

VR virtual reality. 13, 78, 84, 100–104, 129, 151–153, 156, 163–166, 171, 173–177, 180, 181, 183, 186, 215–217, 234, 240

WLK Williams, Linden and Kinderman. 105, 106

XAI explainable artificial intelligence. 23, 83, 146, 226

XUI Explanation User Interface. 148, 221, 226–230, 236, 240, 244–246