Abstract

How can robots learn to move on their own? Consider machines driven by algorithms that allow them to learn everything from scratch, like animals, from their unique experiences. My goal is to explore the complexity of this question from very different directions, from motor control to learning procedures, from searching in simulated worlds to hardware experimentation.

Based on previous research, I substantiate a controller designed as a single-layer neural network suitable for general robot locomotion. Using physical simulation and artificial evolution, I find many forms of motion from the interaction of the controller, body, and objective function. The results show that even a minimal approach produces diverse behaviors like walking, running, and rolling, independent of the robot's shape. Thus, they demonstrate what is generally possible, regardless of how much effort it is to discover.

In the subsequent part, I integrate these findings into learning algorithms. Foremost, to shorten the learning time, but to eventually use them in real machines, too. My approach is an unsupervised technique, combining growing neural networks and intrinsically motivated reinforcement learning. I always maintain a minimalist, modular architecture and try to keep the assumptions on the specific robot as low as possible.

The last part comprises the design of robots suitable for research and application. I describe the mechanical and electronic structure of a quadruped and a legless robot. Their central element is a custom-made distributed motor controller, which collects rich sensory data and is suitable for implementing the controllers described in the previous parts.

Zusammenfassung

Wie können Roboter selbstständig lernen, sich zu bewegen? Maschinen, die von Algorithmen angetrieben sind, die es ihnen ermöglichen, wie Tiere aus ihren ureigenen Erfahrungen alles von Grund auf zu erlernen. Ziel meiner Arbeit ist es, die Vielschichtigkeit dieser Frage aus sehr unterschiedlichen Richtungen zu betrachten, von der Motorsteuerung bis zum Lernverfahren, von der Suche im simulierten bis zum Hardware-Experiment.

Basierend auf vorausgegangener Forschung begründe ich einen Regler-Entwurf, der aus einem einschichtigen rekurrenten neuronalen Netz besteht und der für die allgemeine Fortbewegung von Robotern geeignet ist. Mithilfe physikalischer Simulation und künstlicher Evolution finde ich viele Bewegungsformen in der Wechselwirkung von Regler, Körper und Zielfunktion. Die Ergebnisse zeigen, dass selbst ein minimaler Ansatz vielfältige Verhaltensweisen wie Laufen, Rennen und Rollen unabhängig von der Bauart des Roboters erzeugt. Sie demonstrieren damit, was generell möglich ist, ungeachtet der Frage, wie aufwendig die gewählte Methode danach suchen muss.

Im anschließenden Teil integriere ich diese Erkenntnisse in Lernalgorithmen. Zum einen, um die Lernzeit zu verkürzen und zum anderen, um sie schließlich in realen Maschinen zum Einsatz zu bringen. Mein Ansatz basiert auf selbst-überwachten Techniken, wachsenden neuronalen Netzen und intrinsisch motiviertem Verstärkungslernen. Ich bewahre dabei stets eine minimalistische, modulare Architektur und versuche, die Annahmen über die Gestalt des Roboters so gering wie möglich zu halten.

Der letzte Teil handelt vom Entwurf tauglicher Roboter für Forschung und Anwendung. Ich beschreibe die mechanische und elektrische Struktur eines Vierbeiners und eines beinlosen Roboters. Deren zentrales Element ist eine selbst entwickelte verteilte Motorsteuerung, die reichhaltige sensorische Daten erfasst und geeignet ist, die in den vorangegangenen Teilen beschriebenen Regler umzusetzen.

Preface

This writing documents my research activities in robotics and artificial intelligence over 12 years. Most of the work took place outside of academia, such as during my free time after work, during my sabbaticals, or as part of my current startup endeavors. However, it also complements topics I began exploring as a student or research associate at Humboldt-Universität zu Berlin.

My work focusses on the motion learning of biologically inspired autonomous mobile robots equipped with limbs, emphasising less on their specific physical form, whether humanoid, quadrupedal, or more abstract.

Over this time, various subfields of robotics and AI have evolved, sometimes very rapidly. For example, there has been an apparent push for deep neural networks. However, since I am mainly interested in minimalist methods, such networks are not part of this work. Nonetheless, I indicate the contact points for their integration into the framework.

While humanoid robots have been a prominent focus in science for quite a while, we've witnessed a delightful shift in recent years toward the development of quadruped robots through various open-source initiatives. This transformation is indeed exhilarating, holding vast economic promise. I hope my hardware and electronics concepts can complement the flourishing open-source hardware community hosting numerous parallel ventures. Among these, I would be proud when readers discover inspiring ideas here to spark their individual future hardware ambitions.

Contents

Introduction

Life on earth is exciting. It fascinates me. I feel the inner urge to recreate its mechanisms in robots to understand how they work, notably adaptivity and learning. Is this a good idea? I do not know. But in my view, it is beneficial for us to know more about ourselves, to understand how we learn and adapt to our environment. Building machines that could one day learn like us is a powerful vision I've had for years. Sure, it is uncertain where this development eventually leads to. How self-learning robots will develop in a few decades and how this will affect our society: I do not dare to estimate, and I guess no one can. There might be dangers to overcome. But I preferably participate in its development and make it somewhat better, open, and comprehensible than leaving it to others.

Learning by yourself is hard. It is often easier when someone shows us something than discovering it alone. But when we have learned to do something difficult for us before, we usually feel good. Self-learning means drawing knowledge from the structure of the world by action and observation. To make this process usable for a robot is, in my opinion, still not fully understood. We know well how to train machines to do what we want them to do. But we have comparatively little experience in how they can develop their own objectives or desires that are not fundamentally ours. From my perspective, the most promising approach is to give a machine something similar to curiosity, enabling it to pursue self-determined goals and make experiences.

Why is this essential for us? I find it highly relevant that we comprehend the mechanisms behind adaptive machines. We need to differentiate which got trained on preselected and potentially incomplete data and which learned themselves. The first merely fulfills a human-defined task. The second is shapeable in its behavior, makes its own decisions, and can change its mind in the face of new information. These might all be completely new thoughts for most people, but they frequently lead to irrational but real fears. The reactions of a few people to a robot video I published were extreme. They testified to a deep uncertainty about these new technological possibilities, which authors and filmmakers also like to stage in dystopian terms. I, in contrast, prefer to stick to the curious and kind robot visions, such as the «more input» demanding Johnny 5, the adorable WALL·E, or loyal R2-D2.

Motivation

I hypothesize that legged robots can learn locomotion skills independently from extrinsic rewards, exclusively by their intrinsic motivation. This motivation stems from an inner drive to acquire new knowledge, form fresh synaptic connections, or develop novel sensorimotor abilities, effectively rewarding the robots for exploring and learning.

In the context of locomotion learning, extrinsic rewards refer to incentives provided to the robot from its environment or another source to reinforce specific behaviors or actions related to walking. These rewards are not result of the robot's self-improvement but are instead given as external feedback to shape the robot's behavior. For a walking robot, an extrinsic reward might be a numeric score given for successfully taking steps forward, guiding it in selecting actions to make progress in walking. But what I am aiming for is a learning process in which we can eventually omit external rewards entirely and see the locomotion of the machine as a necessary tool with which it can achieve its inherent goals. If you want to learn something about your environment and your body, it has obvious advantages to move around to explore it.

Imagine a robot lying in a meadow, with no danger far and wide. Assume the robot runs a program that allows it to learn from experience, and we assume further that its memory is initially unwritten. We have its batteries charged and turn it on for the first time. What is there for it to do? We gave it no specific task. Does it imply it should do nothing? Since it is without experience, there are many things to learn, right?

Our robot has diverse sensors to observe the world and to obtain information about itself and its surroundings. It has various actuators with which it can change itself and the environment. Initially, it is unknown how the actuators affect the body or the environment and what the sensors measure. Is there any coupling between them at all? So, the inner drive of this particular machine described here could be to find out how its sensor values change when using its actuators.

Let us delve deeper into our thought experiment and discover why independent learning for robots is not merely a scientifically interesting topic but why it is imperative to continue in this direction. Our early-stage robot self-learning could continue like this: The machine is still in the grass, with fresh batteries and powered on. The first behavior could be slow leg movements, which appear untargeted similar to random motions. But after some time, different limbs begin to synchronize partially, and joints occasionally move in combination. In part, these combined movements get coordinated across its limbs. The more time passes, the more the impression might arise in the observer that the behavior becomes oriented towards a specific goal. The movements become directed enough to push the body a few centimeters from its current position. Maybe the robot lifts the trunk a bit. After some minutes of learning, unregular crawling movements appear, which increase successively in speed and gradually become smoother and more coordinated. Now and then, it might happen that the robot maneuvers into an undesirable position where it gets stuck with its legs or accidentally rolls onto its back and cannot yet turn around by itself. It will take longer in early learning until the robot discovers the correct action to free itself from such situations. It might be necessary that a person occasionally assists.

But why should we expose machines to such situations? Why let them learn in a tiresome way when we could easily teach them what we want and how to do it? While explicit programming of behavior quickly delivers noticeable results, it often runs into the same impasse again. The machine often cannot appropriately react to simple everyday situations the programmers did not foresee. Programming exact behavior for all conceivable situations in a complex world would require extensive effort and is considered practically impossible. And if the circumstances change unexpectedly, this effort is to be repeated. The results generalize poorly. Training data can only partially represent the world. There is usually a strong bias towards the group that compiles the data. Edge cases are often missing.

Our environment is complex and changing. Adaptation is a necessary prerequisite for the survival of any living being. Thus, it is now undisputed that autonomous mobile robots unable to adapt will ultimately turn useless in changing environments. But beyond that, there is still the opinion that machines, even if adaptive, can be trained with data that does not originate from their own experience. I am not talking about two identically produced robots that were pre-trained on the data of their forerunners. I am referring to training machines on data they cannot have individually experienced due to their abilities. So it is questionable to ask today's language models to create a unique non-trivial rhyme or a children's song if it has never spoken out these words and experienced how they sound and how it feels to sing that song to a child. What is missing?

Grounded Robots

I will refer to this property as *Robot Grounding.* Allow me to adopt this term from common language to describe a specific robot trait. In electronics, grounding means connecting a circuit's lower potential to an ideally infinite reservoir of charge carriers through a low-impedance pathway, assuring that the circuit's 0V is equal to the reference potential and cannot float or drift. Proper grounding is crucial for the reliability and safety of many electronic applications. In everyday language, we use the term to describe someone who makes good decisions and does not say or do stupid things. In contrast, kids are grounded when they are not allowed to leave their room as a reaction to them having disobeyed their family rules. In a broader context, being grounded refers to having a firm connection with reality, having a sense of stability and balance, or implying being rooted in one's values, responsibilities, and practical considerations.

Being grounded can thus be literally understood as being in touch with the ground, exchanging forces, and being subject to gravity. And as we will see later, controlling forces plays a significant role in robot locomotion. So we can understand a grounded robot to be in close connection and immediate exchange with its environment through its sensors and actuators. For some time now, I have been using this term to describe to which extent the machine has self-acquired the information it utilizes as the basis for its doing. In other words, how the information got into the system and motivated immediate or future actions. But more particularly, this term categorizes whether the motor output generated by the robot is a direct response to current or past sensor values, whether we have a closed sensorimotor loop or not.

Machines shall Learn from their Unique Experiences

Back to our robots: Ungrounded machines are easy to spot in our modern lives and appear in funny or tragic stories. Whether it was the voice assistant fooled by a parrot ordering random things online or the robotic lawnmower unnoticedly killing hedgehogs, it is evidence of immature technologies that strive into areas they don't fully understand. The approach used to solve these problems remains incomplete.

As of this writing, the most widely employed method has been training machines to learn from examples, continuing to demonstrate a hitherto unexpected potential. However, at the horizon, its limits and downsides are already unmistakably visible and non-negligible because of the training data that has not yet come without the risk of imbalance and bias and the lavish use of our planet's resources in generating, processing, and storing it. So we can only guess when the transition will come, and we will have robots around us in our daily lives that did not get *trained* but *learn* from their own experiences. The idea is not new. Its power was demonstrated often and early, e.g., by Tesauro (1995), where the machine played a board game against itself many times and thus learned to play at a remarkably high level.

In this respect, I believe—and allow myself to make the bold statement—that robots can achieve a potential that will be orders of magnitude above the known, with resources used orders of magnitude below the current, when they become empowered to learn on their own, when they are allowed to explore the unknown, and when they are sensitive and adaptive enough to incorporate changing environments into their behavior.

Methods

The methods I use throughout this work borrow tools from the theory of dynamical systems, control theory, and machine learning. I apply physics simulation with simplified robot shapes to conceptually prove technical approaches. Furthermore, I create electronics, mechanics, and embedded software to build robots for verifying ideas on hardware.

Force-based control methods caught my interest since they produce the most natural-looking robot behavior I have seen. I forgo too much theorizing and focus on applying them to physical motors to get a haptic intuition. Besides their pure vanilla implementations that usually come as a few lines of code, plenty of non-linear side effects from the real world need to be taken care of, and this is where the actual challenge lies.

When using machine learning, my main interest is in self-organizing neural networks and growing structures. Regarding reinforcement learning, I am fascinated by intrinsic motivation and artificial curiosity methods. I prefer open-ended learning methods. Hardware is always subject to wear, so the algorithms must constantly refine what the robot has already learned. But more essentially, for any non-trivial robot in natural environments, there is no end in learning because the behavioral possibilities are immense.

In building hardware, I embrace bounded computational resources and prefer to create systems with inexpensive computation modules. Less computation favors lower power demand, implies longer battery life, and creates more autonomy. Less resource demand is also beneficial for our environment. I am a fan of the open-source culture. For many years, I mostly used open-source software for every aspect of my work. Particularly when creating my electronic designs and firmware, I learned much from projects openly sharing their schematics, hardware designs, and code. That is why I share my work, too.

Contribution

Over the years of this work, theoretical and practical results emerged, which I would like to summarize in the following.

I tested novel force-based sensorimotor loops on a variety of robots, finding ways to use them in practice and devising, implementing, and improving necessary non-linear extensions such as an adaptive stall detection for users' safety. Efforts in this direction have yielded multiple results from experiments, such as a self-raising leg, the balance recovery of a humanoid robot, and the movements of a caterpillar-like pet robot.

In my search for a general robot controller for multi-jointed robots, I have investigated fully connected single-layer neural networks. Even if today's focus is often on deep networks, these computationally sparse structures have significant advantages in their application to embedded systems and, as I could show, already offer enormous potential by embedding PD controllers, the above-mentioned sensorimotor loops, as well as a variety of logic modules, oscillators, and relational operators. Furthermore, I have worked on extensions and formulated a symmetry condition for these networks for robots, which sensibly saves parameters and thus effectively shortens training times.

To test my ideas, I modified a minimalistic open-source simulation environment solely for my purposes and assembled several simulated robots with different shapes. To make it more realistic, I created a discrete-time friction model that is easy to implement and can replicate the non-linear effects of static friction and its transitions.

I used artificial evolution and simulated robots to find parameters for the generalized neural controllers for locomotion. Successfully, I discovered various parameter sets that enable very different robots to locomote, exploring fitness functions for different locomotion styles and directions, as well as necessary constraints and initializations. Consequently, even this relatively simple structure possesses tremendous motion potential for robots with limbs that distinguish themselves through incredibly natural movements. In this process, I developed a generation-free interpretation of the evolutionary method I initially considered for this purpose. My pool-based approach is suitable for multi-processor systems and yields results comparable to the standard variant.

I developed a great interest in self-organizing structures and growing neural networks. While trying to apply these to my robots, I encountered some problems to solve. I thus developed a new form that requires fewer hyperparameters than usual, resolves temporal dependencies, and shows no catastrophic forgetting when applied to non-stationary data. I could demonstrate that it is suited for reinforcement learning and how to extend it with more sophisticated prediction networks or my self-regulating form of the softmax layer. Parts of these developments I already started during my diploma thesis and could improve and merge them here.

The most concrete and immediate outcome of this work is the creation of robots and the electronic hardware components needed to construct them. I have developed networked motor controllers that simplify the assembly of robots of varying shapes and offer extensive sensory feedback capabilities for contemporary robot control methods. Using these components, I designed three distinct robots. Two of them, a four-legged and a bipedal robot, are published as do-it-yourself kits for research and development.

The most recent addition is a pet-like robot, a product of my startup company, aimed at entertainment and therapy applications. All electronic components are libre/free-design hardware and include open-source firmware.

List of Publications

I have already published parts of my research in conference papers. Some ideas that I continued here originated in my earlier academic writings. Selected relevant publications are listed below for reference.

(2022) *flatcat – playful robots that respond to touch*, M. Kubisch and O. Berthold, Robot Curiosity in Human-Robot Interaction (RCHRI) Workshop at HRI

(2017) *A growing multi-expert structure for open-ended unsupervised learning of sensory state spaces*, M. Kubisch, 7th Joint IEEE International Conference on Development and Learning and on Epigenetic Robotics

(2011) *Neural Implementation of Behavior Control*, M. Kubisch, C. Benckendorff, B. Werner, S. Bethge, and M. Hild, in Language Grounding in Robots

(2011) *Using Co-Existing Attractors of a Sensorimotor Loop for the Motion Control of a Humanoid Robot*, M. Kubisch, B. Werner, and M. Hild, International Conference on Neural Computation Theory and Applications

(2011) *Balance Recovery of a Humanoid Robot Using Cognitive Sensorimotor Loops (CSLs)*, M. Kubisch, C. Benckendorff, and M. Hild, 14th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines

(2011) *Myon: Concepts and Design of a Modular Humanoid Robot Which Can Be Reassembled During Runtime*, M. Hild, T. Siedel, C. Benckendorff, M. Kubisch, and C. Thiele, 14th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines

(2011) *Self-Exploration of Autonomous Robots Using Attractor-Based Behavior Control and ABC-Learning*, M. Hild and M. Kubisch, 11th Scand. Conf. on Artificial Intelligence

(2010) *Proposal of an Intrinsically Motivated System for Exploration of Sensorimotor State Spaces*, M. Kubisch, M. Hild, and S. Höfer, 10th Int. Conf. on Epigenetic Robotics

(2010) *Intrinsisch motivierte Exploration sensomotorischer Zustandsräume*, M. Kubisch, Diploma Thesis, Humboldt-Universität zu Berlin, Institut für Informatik

(2008) *Modellierung und Simulation nichtlinearer Motoreigenschaften*, M. Kubisch, Student Research Project, Humboldt-Universität zu Berlin, Institut für Informatik

Related Work

From Strogatz (1994), I learned that even the lowest-dimensional systems can already exhibit complex to chaotic dynamics. Braitenberg (1984) convincingly showed me that intricate behavior like chasing and escaping can emerge from just a few linear connections of sensors to motors. At that time, researchers often used discrete systems in grid mazes or pole carts, but their properties seemed inadequate for me. I wanted to try continuous systems with adjustable properties that are easy to visualize. Inspired by the rich non-linear dynamics of the two-neuron recurrent networks by Pasemann et al. (2003), I set up a small project in the C programming language. I imagined such systems as handy test environments for point mass robots. They were literally tiny dots floating in vector fields towards specific endpoints or curling around periodically. The robot-dots could move in their world according to simple control pad actions.

During a lecture, I heard of Growing Neural Gas (GNG) by Fritzke, and I got the homework assigned to implement it from scratch. I was impressed by its remarkable level of self-organization. Subsequently, I became aware of the sensorimotor map by Toussaint (2006), and I was captivated by the instantaneous adaptability of its structure. For the first time, I caught how self-organization shapes robot behavior. Intrigued by its potential, I aimed to apply the method to my miniature simulated robots as a starting point. Even with reduced dimensionality, I quickly realized that I needed to extend GNG to handle temporal dependencies in the inputs. I saw great potential in mapping the robot's sensory inputs to states in a sensorimotor map.

Schmidhuber (2006, 2010) presents a formal approach to comprehending how agents can be motivated by curiosity and the desire to explore their environment and seek novelty. His framework incorporates information theory and reinforcement learning to explain how agents can develop creativity and intrinsic motivation by lowering the agent's prediction error or improving data compression. Oudeyer et al. (2007) sought to improve our understanding of intrinsic motivation by putting the formal aspects into the actual programming of a quadruped pet-like robot placed on a baby playmat. Even though I did not find all aspects of their approach practical to implement, their groundbreaking experiment was, in my eyes, an important milestone that spurred me on to try my own implementation. It changed my way of thinking about robots fundamentally. My simplified robots now had a motivation to explore their environment. But I had to read Sutton and Barto (1998) first to get myself ready to implement reinforcement learning.

Rechenberg (1994) pioneered and popularized evolutionary algorithms, particularly his Evolution Strategy, a stochastic optimization technique that copies the process of biological evolution to find optimal solutions to complex problems. Sims (1994) applied evolutionary algorithms to simulated creatures and co-evolved morphology and behavior control. His work was visually impressive and demonstrated how lifelike simulated behavior can be. Nolfi and Floreano (2000) provided a comprehensive overview of evolutionary robotics and its potential for creating adaptive and intelligent robots. Bongard (2011) delved deeper into the evolution of robot bodies regarding physical structures and shapes. He investigated how effective robot forms emerge from evolved behavior, which might inspire the design of more efficient and adaptable robots. But it was Solomon et al. (2010) who focussed so purely on simulated bipedal walking evolution that got me hooked. On a drastically reduced setup of a 2D stick figure's lower body, they evolved their robots to walk down slopes and steps of significant size. They used fully interconnected neural structures that were simple but effective enough to run my simulations and got me to explore them. I immediately started modifying Simloid, a robot physics simulation written by Hein (2007) for his thesis and provided to students like me for a lecture exercise. So I went from dots to simulated legged robots, and after some weeks, my robots evolved to walk and run in simulation.

Inspired by the work of Dörner (2001) and Doya (2002), I wanted to put together all the parts: robot body, behavior, and motivation. Both authors drew bold pictures of frameworks that eventually might lead to intelligent, autonomous robots. It contained goals, self-regulation, and meta-learning. And since they are not giving us an easy manual for practical implementation, we can find thoughts in their work that will probably still take us years to master. One of the most crucial questions that drove me was: Where do goals come from? The exploration of intrinsic body dynamics or fundamental motor actions is explained with the Homoekinesic Principle by Der and Martius (2012). Intrinsic motivation, the goal to seek novelty, explains exploratory behavior. Homeostatic rules keep specific vitality parameters in safe bounds and thus explain conservative and deliberate behavior. Empowerment, as formally described by Polani et al. (2001) and Klyubin et al. (2008), extends the set of goals by seeking states of more options and keeping them. However, we still lack a framework that generalizes all these goals.

Outline

This document has three parts: Part one, consisting of three chapters, lays the foundation for my research. Chapter 1 delves into setting the theoretical groundwork, providing brief reviews of prior work to support my choice of control structure. Chapter 2 defines the experimental methods and tools I employ in my study. Finally, Chapter 3 is about evolving multiple walking controllers for legged robots.

Moving on to the second part, comprising two chapters, I focus on robot self-learning of locomotion. Chapter 4 offers an overview of various machine learning methods I apply, providing all algorithmic components for the experiments to follow. In Chapter 5, I introduce a custom algorithm and conduct experiments to demonstrate its applicability.

The final part encompasses four chapters, leaves the theoretical parts behind, and enters the practical side of this work. In Chapter 6, I write about hardware design aspects within the context of open-source culture. Chapter 7 examines the electronics designed specifically for legged robots. Chapter 8 introduces a DIY four-legged robot engineered for research purposes. Lastly, Chapter 9 presents a pet-like robot developed for entertainment and therapy applications. This work concludes with an outlook on ideas for future research and the appendix holding schematics of the robot electronics, complementing the main content.

Part I

In Search for Fundamental Modes of Robot Motion

1 Designing Learnable Control Structures

Legged robots should be able to teach themselves to move. We can enable them to acquire motions for various situations *by learning* and let them make their own decisions and experiences. Thoughts like this come from the conclusion that even for robots with simple morphologies, it is already challenging to create robust walking motions that can withstand moderate environmental disturbances. Furthermore, we might want these motions to be energy-efficient and material-friendly. Also, real robots are subject to wear, so the controller probably needs to adapt to subtle changes in hardware over time. But most critical, how could we foresee every possible situation the robot will experience?

The control structure we are searching for should be largely independent of the specific morphology to apply to various robots with a different number of joints. Ideally, this structure would require only a few assumptions on the robot's hardware. It must be strictly independent of how the robot's joints are composed, the location of sensors and motors, and how they influence each other.

Instead of defining and modeling every detail, we can take the world as it is: a wonderfully messy structure. We could define a parametrizable control scheme for our robots, enabling us to find various behaviors with generic optimization techniques. Finding motions that cover a large part of the robot's behavioral space, for instance, directed walking and running, turning, recovering balance, and the like.

The choice of initial parameters dramatically influences the time needed for the optimization process, and we benefit when starting from the already known. If the structure has a parameterization that allows for embedding already working controllers, it would enable us to find solutions much faster. For example, we can use position controllers to create initial conditions for keeping the robot stable. The CSL, presented in section 1.3, is suitable for robots to get up and recover balance. Oscillation capabilities could create the dynamics needed for initial walking movements.

Finally, mobile robots are subject to bounded energy resources. Lighter hardware saves energy but might bring additional elasticity, which is more challenging to control with traditional approaches. Saving computational efforts also saves energy. Thus, it is beneficial to make the motion controller work on embedded electronics and fulfill the real-time requirements for reactive motions like running or jumping.

With all this in mind, I want to propose a control structure that I will use for the rest of the thesis. To begin with, I will briefly recap the basic terms of dynamical systems theory and controller design. I will then discuss classical and modern controllers for systems with a single output and their application to real robots. At last, I will present a learnable generalized controller structure as the result of my experiences in manually crafting robot motions so far in earlier works. I will use this structure in simulated and hardware experiments conducted in this thesis.

1.1 Dynamical Systems

The theory of dynamical systems, referring to Guckenheimer and Holmes (1983), Thompson and Stewart (1986), and Strogatz (1994), is an important tool for researchers and engineers of robotics and machine learning to describe the properties of physical systems, sensorimotor control loops, and neural network dynamics.

A system is dynamic if its state changes over time. Consider the dynamical system f with the update rule

$$\boldsymbol{x}(t) \leftarrow f(\boldsymbol{x}(t-1), \boldsymbol{u}(t), \boldsymbol{\theta}(t)) \tag{1.1}$$

comprising the system's state $\boldsymbol{x}(t) \in \boldsymbol{M}$ at time step $t \in \mathbb{N}$, the input to the system $\boldsymbol{u} \in \mathbb{R}^K$, and the system's parameters $\boldsymbol{\theta} \in \mathbb{R}^D$. We call the manifold $\boldsymbol{M}$ of all possible system states the *state space* or *phase space* and we define it as $\boldsymbol{M} \subset \mathbb{R}^N$. The system starts at time $t = 0$ with $\boldsymbol{x}(0)$. The system is *autonomous* when it does not directly depend on time.

All the components can have multiple dimensions, $N, K, D \in \mathbb{N}^+$, and we denote them by convention as column vectors, e.g., $\boldsymbol{x} = (x_0, x_1, \ldots, x_{N-1})^\top$. Within this thesis, I focus on discrete time dynamical systems only, so each variable usually carries a time index. For the sake of simplicity, we omit the index if it is identical on both sides of the equation. To further increase readability for update rules, we denote $y(t) = g(y(t-1))$ simply as $y \leftarrow g(y)$ where the right-hand side always represents the previous time step's value of the concerned variable. Hence Equation 1.1 condenses to the more compact and convenient form

$$\boldsymbol{x} \leftarrow f(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\theta}). \tag{1.2}$$

A *trajectory* of the system is an ordered set of successive states along which the system evolves. Think of trajectories as the paths of the system through the state space. They can converge but never cross themselves or each other. A trajectory becomes an *orbit* if it exhibits a certain periodicity, i.e., when the same states are recurring on this trajectory. There are different types of periodic orbits: We denote a *p-orbit* when we can specify the periodicity by $p \in \mathbb{N},\ p \geq 2$ reiterating states $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{p-1}$. We speak of *quasi-periodic orbits* or *limit cycles* when the recurring states are not precisely the same but come arbitrarily close. For instance, we can describe a stable oscillator as a quasi-periodic orbit. In later chapters, such orbits will play an essential role since the gaits of walking robots form multi-dimensional oscillators. Their trajectories constitute quasi-periodic orbits through the phase space of joint angles and angular velocities.

The vector $\boldsymbol{\theta}$ constitutes the system's *configuration.* We use it to describe the system's properties and how they change according to some objective. For instance, we could consider a pendulum with a particular mass, length, and friction. One may object that the configuration should stay constant in this regard. But we are considering robots here. We should expect the system's configuration to change because robot hardware is subject to wear. However, we mostly suppose it to change way more slowly, so we can assume them as quasi-constant parameters. For some variables, it may be convenient to consider them either part of the configuration or the system's state, e.g., the battery voltage.

We use the control input, $\boldsymbol{u}$, to influence the system. It is part of our system's state but gets updated externally. We can use it to steer the system's evolution into a desired region of the phase space. Remembering the pendulum, we imagine $\boldsymbol{u}$ as the force, initially deflecting it before we release it to swing until it eventually halts.

Attractors

Attractors characterize a dynamical system. They are a set of states towards which a system evolves. Attractors can be *fixed points*, fixed curves (multiple equilibria nearby), or even sub-manifolds of the phase space as the already mentioned orbits. Attractors can *co-exist* within the same system. There must be at least one attractor in any dynamical system. If there is more than one attractor in a system, it depends on the initial conditions towards which attractor our system evolves. All initial conditions leading to the same attractor comprise the attractor's corresponding *basin*. Each state of the system must inevitably progress towards one of the system's attractors. The virtual border between two adjacent basins is the so-called *separatrix* See Figure 1.1 for illustration.

The *vector field* describes the evolution of the system. It can help to identify different attractors, basins, and separatrices. The vector field is the system's partial derivative according to each dimension and assigns in each state, $\boldsymbol{x}$, a vector

$$\operatorname{grad} f(\boldsymbol{x}) = \nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right)^\top . \tag{1.3}$$

Fixed points of the system are states for which the vector field (or the gradient) in that point is zero in all dimensions, $\nabla f = \mathbf{0}$. Hence, the system's evolution finally stops in fixed points due to $\boldsymbol{x}(t) = \boldsymbol{x}(t-1)$. Depending on the properties of the field around that fixed point, we either have a stable or unstable one. Stable fixed points are attractors. All trajectories starting within the corresponding basin evolve towards that point. So, small control inputs would have no effect. The system would return to it. For instance, consider a pendulum subjected to gravity hanging towards the ground: A nudge will cause the pendulum to swing for a moment, but it finally comes to a halt at the same fixed point.

Unstable fixed points, on the other hand, are usually located on separatrices with the vector field pointing away from them. So modest control inputs would probably suffice to drive the system away into the next corresponding stable fixed point. Imagine a balanced pole (or inverted pendulum) resting: A little nudge would make it fall toward stability. Unstable fixed points are repellers, the opposite of attractors.

Since quasi-periodic orbits are attractors, they show stability in the same meaning as stable fixed points. Small control inputs will go to the same attractor again. Also, such orbits have basins. There is always a specific region of the phase space leading towards this attractor. If we again consider the gait pattern of a walking robot as such a system's attractor, it is inherent that we have a certain stability margin. We could design the gait so that we maximize this area of stability. Exposing the robot to disturbances during learning creates walking patterns as multi-dimensional oscillators that are stable against disturbances. They naturally find their way back to the attractor, meaning the robot will recover balance after hitting minor obstacles with its legs.

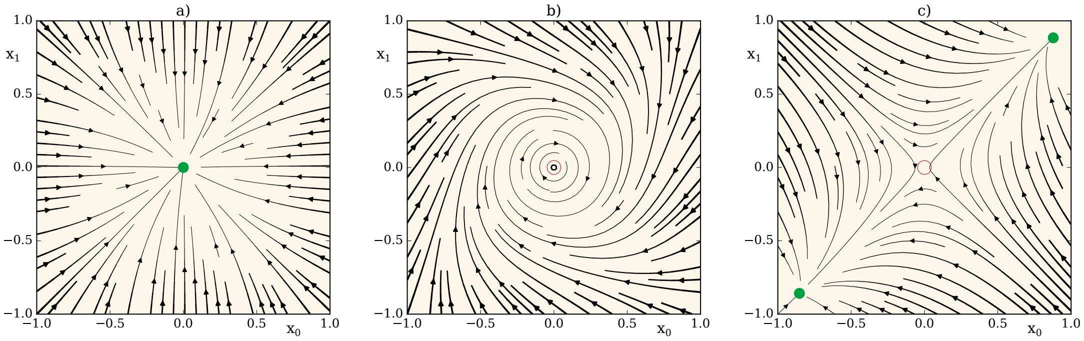


Figure 1.1: Vector fields to illustrate common attractors: a) stable fixed point at $(0, 0)$, b) quasi-periodic orbit, and c) two coexisting stable fixed points. For b) and c), there is also an unstable fixed point at $(0, 0)$. In c), the separatrix spans from $(-1, 1)$ to $(1, -1)$. Fixed points are denoted ● for stable and ○ for unstable.

Bifurcation

Given the parameterized system of Equation 1.3, if we change the system's configuration $\boldsymbol{\theta}$, the attractors of the system will probably change as well. The attractors might deform, e.g., fixed points shift their position, and orbits change their size or form. But when attractors appear, merge, or suddenly disappear due to recent parameter changes, we glimpse a *bifurcation*. The parameter value at which this occurs is called the bifurcation point. We call several bifurcations that occur in succession a bifurcation sequence. When the attractor landscape of a system changes according to the system's parameters, we should be aware of that effect during learning. Hence, depending on the system's architecture, small parameter changes can yield dramatic changes in behavior, for example, when we apply learning rules and then pass a bifurcation point. In particular, recurrent neural network dynamics are famously prone to bifurcation when subject to weight changes during learning as examined by Doya (1993).

Determinism and Noise

When not noted otherwise, we consider the system's transfer function f to be deterministic, that is, for each given tuple of inputs $(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\theta})$ the output is always identical. Even though this does not strictly hold for the given robotic systems, there is no explicitly modeled stochasticity in f. When the function f represents a simulated or physical robot, the (simulated) sensors exhibit a certain degree of noise, and obstacles may unforeseeably change the sensors' values. But even in the complete absence of noise, a deterministic system must not necessarily imply easy predictability. The transfer function might be highly non-linear. So, even for low-dimensional systems, minor deviations in initial conditions may quickly lead to totally different outcomes. Remember the chaotic trajectory of a simulated double-pendulum, where the differential equations are purely deterministic, but the system's evolution is not easy to predict. Besides the sensors' noise, the control input $\boldsymbol{u}$ will often be stochastic when, for instance, $\boldsymbol{u}$ stems from a reinforcement learning agent with a stochastic policy. Agents select actions according to preference estimations that need regularly exploring random selections for proper development.

1.2 Control Basics

Controllers monitor the system's state, denoted as $\boldsymbol{x}$, representing the *process variable*. They generate outputs, which we call the *control variable*, $\boldsymbol{u}$, computed for each time step to achieve a specific objective. In general, the controller could make its decisions on the history of system states, $\boldsymbol{x}(t), \boldsymbol{x}(t-1), ..., \boldsymbol{x}(0)$, but a more convenient approximation to this form is to provide the controller with its own state by memorizing its recent output, $\boldsymbol{u}(t-1)$, and only using the current state of the system, $\boldsymbol{x}(t)$. The general update rule for such a controller is

$$\boldsymbol{u} \leftarrow c(\boldsymbol{u}, \boldsymbol{x}, \boldsymbol{w}) \tag{1.4}$$

where $\boldsymbol{w} \in \mathbb{R}^L$ is the column vector of the controller's parameters or *weights* of dimension $L \in \mathbb{N}^+$. We refer to the dynamical system under control as the *plant*.

Objectives

Objectives can vary wildly. For instance, a simple one could be to minimize the error between the desired value, $\bar{\boldsymbol{x}}$, and the measured value, $\boldsymbol{x}$, of the process variable. We usually term the desired value the *setpoint* or *target value* and define the error to minimize as

$$\boldsymbol{e} = \bar{\boldsymbol{x}} - \boldsymbol{x}. \tag{1.5}$$

If a control objective is to minimize the system's distance to the desired state, we can imagine the controlled system as having a new attractor. The setpoint becomes a fixed point towards which the system evolves. Similar objectives could be to keep the state within certain limits and create control inputs accordingly. The goal must not necessarily be to bring the system into a single state but to tie it onto a sub-manifold of the state space. See Figure 1.2 for an illustration of a regular closed control loop with a setpoint.

In the simplest case, the objective function considers only a sub-state of the system, e.g., controlling a single robot joint's target position. But the controller can likewise affect all dimensions of the state variable, and our objective could be a derived measure of the accumulated state like the average walking speed of a robot being 1 m/s. With such an objective, we cannot easily specify simple target states. Instead, the state must undergo a specific sequence or trajectory. In the case of a walking robot, we expect a quasi-periodic orbit. Hence, in contrast to creating a fixed point, as for the simple setpoint, we must shape the state space to have another non-trivial attractor.

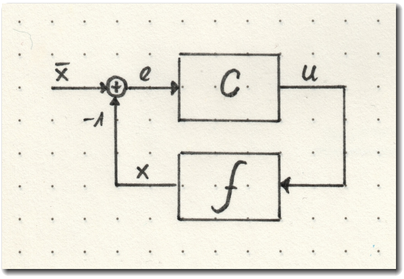


Figure 1.2: Structure of a closed control loop with a setpoint: The target value, $\bar{x}$, is compared to the system's actual state, x, resulting in an error signal denoted by e. The controller reads e and is trying to change the system's state via the control output, u, in order to minimize the error.

A comment on notation: As with the controlled systems, we also consider the controller acting in the discrete-time domain. In the structure diagrams, I explicitly denote values of passed time steps with the unit-delay operator, z^{-1}, a regular operator when using z-transformation. And just like with the equations, I often omit the time index to simplify the figures and increase readability.

Common Controllers

A typical application is (angular) position control. So the objective is to minimize the distance $e = \bar{\varphi} - \varphi$ of the measured joint angle, φ, to the target angle, $\bar{\varphi}$. The objective often incorporates that the process of distance minimization has further constraints, like minimizing the used time or energy for the transition. For instance, if the system is a DC motor, the control variable could be the motor's voltage, expecting a change in speed and position according to the amount of applied voltage.

In the same way, we can control the velocity of such a system. We choose a target velocity, $\bar{\omega}$, and compare it to the measured actual velocity $\omega = \dot{\varphi} = d\varphi/dt$. Controllers can be cascaded, e.g., a position controller outputs a target velocity that is then again fed into a velocity controller, creating voltage values. Controllers can also operate in parallel, e.g., the already mentioned controller cascade will improve by a limiter, ensuring the obeyance of mechanical boundaries by creating control values to protect the motor from breaking when the position is getting out of bounds. Another parallel controller could restrict the maximal current consumed or the allowed temperature of the system.

1.2.1 Proportional-Integral-Derivative Controller

PID controllers, referring to Aström and Hägelund (1995), are excellent in bringing the system into certain target states, e.g., for position or velocity control. With a given setpoint and the control deviation, the controller creates an attractor around the target value. We desire to install a fixed point there and around it, a non-curling strong vector field towards this fixed point. So, we usually want to minimize the time needed to reach the target and avoid larger overshoots.

A *proportional controller* is stateless. Its output, u_p, is directly proportional to the control deviation, $u_p = k_p e$. Its control response is, therefore, a step function. The proportional controller is very reactive and very well suited to bring the process variable quickly close to the target value. If no arbitrarily small outputs are possible, for instance, due to technical limitations or friction, the P-controller may not always fully compensate for the control deviation. For some purposes, it may be sufficient to adequately increase the gain, k_p, while limiting the output. However, we should consider that excessive amplification can lead to overshooting if the controlled system has inertia.

An *integral controller* has an internal state which accumulates the control deviation. Such a controller produces an output proportional to the sum of the control deviations, $u_i = k_i \sum e$. A residual control deviation, as in the case of the P-controller, must, after a few time steps, inevitably lead to a control output with a sufficiently large amplitude to cancel out the deviation. The name integral controller suggests using integrals, but in the discrete-time systems we are considering, we can approximate the integral with a simple summation. An integral controller reacts slower than a P-controller due to the integrator's low-pass effect, so it tends to overshoot. Permanent deviations lead to *wind-*

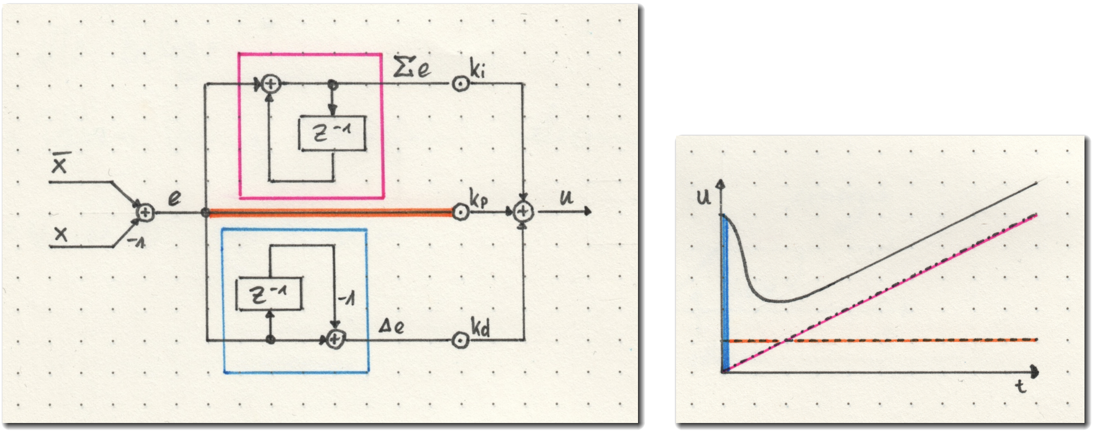


Figure 1.3: Structure of the PID controller (left), responses of control output to constant deviation (right). The PID is a parallel structure: The error signal e splits into three branches, and the control outputs of the individual components are added up and applied simultaneously.

up. If the process variable is subject to static friction, it usually comes to *limit cycles* that could require special handling. The proportional and integral controllers can be easily combined (PI) and partially compensate for each other's drawbacks.

Sudden changes to the setpoint lead to jumps in the control deviation, further amplified by the proportional component. But if we remember the last error, and if we add a *derivative* term $u_d = k_d \Delta e$, we can catch the fast transients of the P-controller. The D-component is, in principle, a simple high-pass filter, removing the transients from the signal and creating damping. Considering time-discrete systems only, we use the difference as the simplest form of discrete derivative. The differential quotient becomes the difference quotient

$$\frac{de}{dt} \approx \frac{\Delta e}{\Delta t} = \frac{e(t) - e(t - \Delta t)}{\Delta t} \tag{1.6}$$

where the factor Δt^{-1} is implicitly included in k_d. The D-component itself cannot compensate for a control deviation, so it usually occurs in combination with the P controller.

Combining all three fundamental elements results in the widely-known *proportional-integral-derivative controller* as depicted in Figure 1.3. Three controllers act in parallel on the same system and influence each other: The proportional term acts on the control variable akin to the restoring force of an ideal spring with the error as the deflection, while the derivative term acts on the error's velocity as a damper. Thus, the PD parts of the controller can be treated like a spring-damper system if the controlled quantity is a position signal. Considering our application to legged robots, this is an essential property since we can adjust the stiffness and damping of the robot's joints separately.

1.2.2 Non-Linear Extensions to PID Control

Linear control is often limited to certain linearized domains of (in general) non-linear control systems. Or maybe the underlying system is just assumed linear for a given

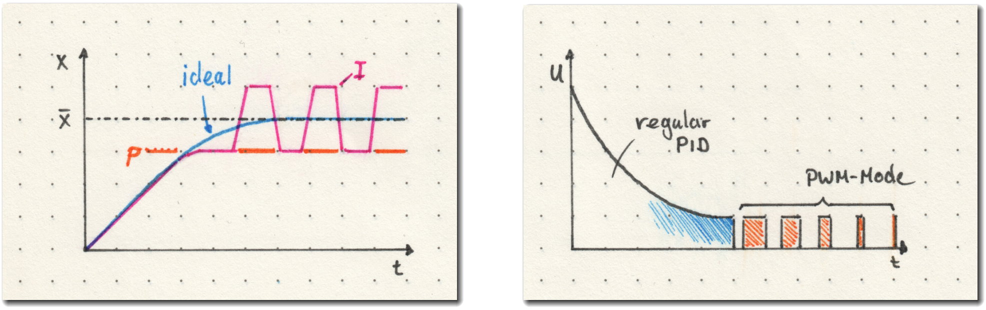


Figure 1.4: Approaching the setpoint under sticking friction, ideally vs. realistically (left). The control output of the *anti-stiction* PWM mode (right).

range of input values. However, at the edges of such a quasi-linear domain, we probably must change the controller's properties if the controller should be working beyond this domain. For instance, we can imagine handling a position controller differently if we are far away or close to the target position.

We might want a high proportional gain for quickly reaching the target. But with higher amplification, we usually must limit the output to avoid overshooting, so we have found our first non-linear element. When the control output is limited, whether by the algorithm or the physics, using an I-component leads to wind-up. A common anti-wind-up technique is to freeze the I-part when reaching a predefined limit and thus avoid an unnecessary delay when we have to rewind the error integrator again.

If we consider systems with friction, permanent control deviations (using P) or limit cycles (using I) can occur. A simple deadband like

$$f_{db}(x) = \begin{cases} 0, & |x| < \kappa \\ x & else \end{cases}, \tag{1.7}$$

can help to set the output to zero when we are below the motor's motion limit $\kappa \in \mathbb{R}^+$. However, this forces the controller to give up and settle for the residual control deviation. Considering systems with higher precision requirements, we need a different way to overcome the residual without causing potential limit cycles. Wescott (2014) proposes an elegant solution of switching to a low-frequency PWM-like mode if the system is very close to the target. Instead of further decreasing the control output (P) or winding it up (I), we measure the smallest motor output that can change the system within a single time step and create pulses with a decreasing duty cycle. We ensure the system will move because the voltage is high enough, but the pulsing gets shorter as we approach closer to the target. The controller actively shakes the system closer to the target value, trying to avoid the domain of sticking friction, which we will later address in section 2.2.

So, even though most of our controller models are considered linear in their pure theory, real-world controller applications are often highly non-linear. We probably need to implement modifications to make them work as desired. To deal with non-linearities is the general case with real hardware. But before we continue with fully non-linear controllers, we should consider refining the objective and questioning the approach of controlling setpoints and then see what benefit we can get from this strategy.

1.3 Cognitive Sensorimotor Loops

Position control is an intuitive approach to generating motion for robots. We plan trajectories as a sequence of target angles for each joint and then try to maintain them using PID position controllers. This approach works best in predictable settings like industrial applications and offers the benefit of separately considering controller design and trajectory planning. The main disadvantage, however, is that typical environments for humans—where we also might want our robots to be—tend to be more unstructured. This fact also applies to robots with legs. We would require a detailed and constantly updated world model for each environment. But, the world outside the laboratory is particularly complex to model in advance, and estimating it during operation brings additional challenges.

What can we do to improve the situation? We can take forces into account. High-precision sensors and a sophisticated robot model would allow for measuring active forces during operation so we can integrate them into the step-planning process to compensate for potential disruptions. However, as the number of joints increases, the effort of modeling and calculating the inverse dynamics becomes increasingly problematic. Additionally, we cannot assume the robot model to remain unchanged over time, as we need to incorporate wear and payloads. Therefore, we would benefit from more robust control approaches that can cope with variations in the situation, inaccuracies in the model, and with light loads. But before we consider the general approach of locomotion self-learning, we can discuss control loops that can already generate very robust movements without many assumptions. This section introduces Cognitive Sensorimotor Loops (CSL) that are able to generate a wide range of different behaviors with only a few parameters, as shown by Hild and Kubisch (2011); Kubisch et al. (2011a); Werner (2013); Bethge (2014); Meier (2015), and Kubisch and Berthold (2022).

Definition

A sensorimotor loop generally refers to a connection of a system's sensor inputs to its actuators, affecting the environment and, in turn, leading to measurable changes to the sensors again. CSL is such a sensorimotor loop, as depicted in Figure 1.5. The associated system in the following is a single rotary robot joint with an axially mounted sensor and motor. The only sensor input is the joint angular velocity, $\dot{\varphi}(t)$. The output of the CSL is the motor voltage $u(t)$. The controller is defined as the discrete-time update rule:

$$z(t) = g_f u(t-1) \tag{1.8}$$

$$u(t) = -g_i \dot{\varphi}(t) + z(t) \tag{1.9}$$

In the model it is assumed that the joint angle $\varphi(t)$ and the motor voltage $u(t)$ have the same direction of rotation, i.e., if $u(t) > 0$ the motor accelerates to $\dot{\varphi}(t) > 0$. CSLs are able to generate astonishingly complex behavior with minimal computational effort.

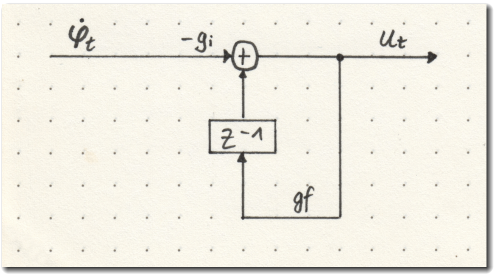


Figure 1.5: Schematics of the Cognitive Sensorimotor Loop. The loop gain g_i amplifies the velocity input $\dot{\varphi}$, which feeds into an integrator with feedback gain g_f. Depending on the configuration (g_i, g_f), a CSL makes a robot joint behave differently.

Support	Release	Hold	Contraction
$g_i < 0$	$g_i > 0$	$g_i > 0$	$g_i > 0$
$g_f = 0$	$0 \leq g_f < 1$	$g_f = 1$	$g_f > 1$

Table 1.1: Overview of different behavioral modes. Each configuration generates a different specific behavior. Figure 1.7 illustrates the behavior on a simple pendulum arm.

They are closed loop controllers and have an internal state, z. If the joint velocity is not available as a raw sensor value, but the joint angle, or if the velocity would be anyway derived from the position sensor, then it is more convenient to use the following equivalent form of the CSL, as it was originally published:

$$z(t) = g_f u(t-1) + g_i \varphi(t-1) \tag{1.10}$$

$$u(t) = -g_i \varphi(t) + z(t) \tag{1.11}$$

This form uses the joint position and derives the velocity with the help of a simple differentiator in form of a first order FIR high-pass filter.

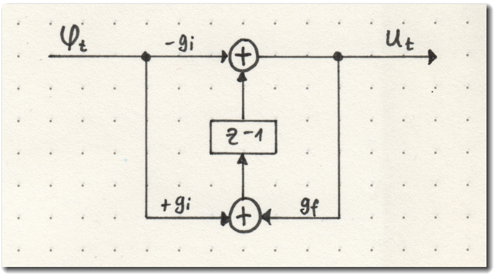


Figure 1.6: Schematics of the CSL's original form with position input: A simple differentiator derives the position signal and turns it into a velocity signal, making this form preferable if no velocity sensor value is available. Similar to the *direct form II* of digital filters, the differentiator and the integrator share a common delay-unit.

Configuration

Cognitive Sensorimotor Loops have four qualitatively different behavioral modes: Contraction, Hold, Release, and Support. All four are variable in their strength and cover a 2-dimensional parameter range; if this range is traversed slowly in a certain way, the behaviors also blend smoothly into each other. When following this path through the configuration space, as described in Figure 1.9, we travel through several bifurcations, where the behavior qualitatively changes.

Support Mode This mode proportionally feeds back the positive velocity, i.e., in the direction of the current motion, thus allowing partial compensation of friction (which is

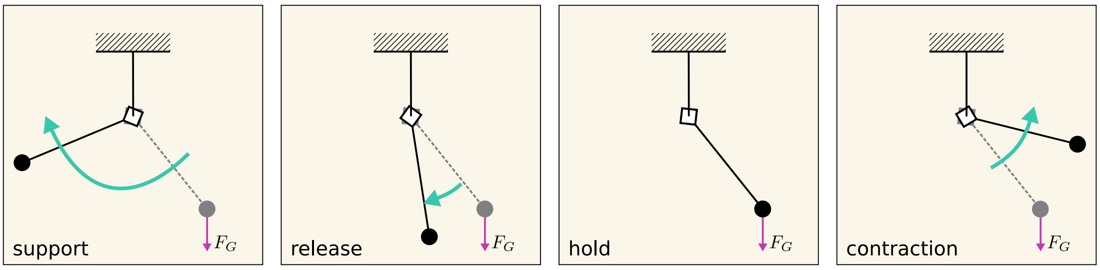


Figure 1.7: Behavioral modes of the CSL demonstrated on a single joint (from left to right): The *support* mode further amplifies the measured velocity and compensates for friction. The *release* mode negates the velocity and acts like additional friction. The *hold* mode remembers the relative position and tries to restore it while *contraction* mode detects applied forces and works against them, e.g., gravity.

negatively proportional to the movement direction). It can also generate comparatively slow oscillations if there is a restoring force around a stable fixed point in the physical system, e.g., to swing up a pendulum subjected to gravity.

Release Mode This mode slows down movements, similar to friction. It counteracts the motion, decelerates, and finally halts the system. Combining a loop gain $g_i > 0$ and a regulating $0 < g_f < 1$, we can adjust the release mode from soft damping to instantaneous braking.

Hold Mode This mode works similarly to a position controller, setting the target position as soon as it activates. The perfect integrator ($g_f = 1$) sums up the joint's velocity to a relative position, which preserves as long as the deflecting force exists. The hold mode memorizes the deviation and aims to restore it.

Contraction Mode The most complex behavior is generated in contraction mode, where the system works against all external forces, friction, and gravity. The negatively fed back velocity is integrated and further amplified by the feedback parameter $g_f > 1$. The controller detects the direction of the applied force and applies reaction forces. This mode works against the current movement direction, but not in proportion. Due to the feedback weight of greater than 1, the contraction mode is able to *remember* the last applied force and works against it. In addition, the contraction mode can move out of stable fixed points and hence converts stable fixed points into unstable and vice versa. The contraction mode works against gravity. It lifts a pendulum arm and stabilizes it in the previously unstable fixed point. When there are forces acting on the system, the contraction mode will convert the detected speed into an opposite force until the cause of the speed change was overcome. If this should not be possible, e.g., when hitting an obstacle or joint limits, the CSL in contraction amplifies to the maximum and retains its state, pressing with full power. Referring to the dynamical system's view: If contraction mode changes roles of fixed points from stable to unstable and from unstable to stable, what if there was no unstable fixed point beforehand? Or what if the force to be applied by the controller is not sufficient to go there directly? Pure application of contraction mode will let the controller get stuck in systems like these and should be avoided. There will be comments on this topic later on in this section.

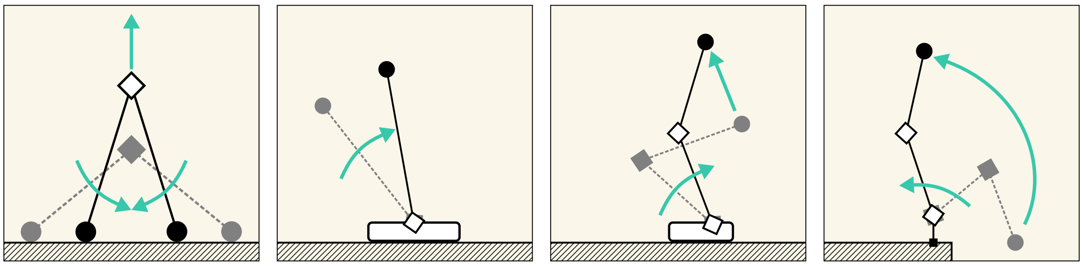


Figure 1.8: Illustration of the contraction mode acting on sample joint configurations subjected to gravity. Joints in contraction mode will defy external forces and raise themselves against gravity.

Implementation and Parameter Adjustment

The arithmetic effort for a joint with CSL is constant in time and is only a few lines of code using basic math operations:

```
z = gi * p                # init. int. state
def csl_step(p):
  u = -gi * p + z         # output voltage
  z = gi * p + gf * u     # refr. int. state
  return u
```

The central parameter is the loop gain, g_i. To adjust it, we keep the CSL in hold mode with the feedback weight constant as $g_f = 1$, so we can gently increase g_i. For hold mode, we can use the analogy of a mechanical spring, as with lower loop gain, the motorized joint will behave as such. The loop gain corresponds to the spring constant, i.e., a higher value leads to a stiffer spring. Starting at $g_i = 1$, we can slowly increase the parameter to the desired stiffness. At some point, the system's friction is no longer sufficient. An increased loop gain will initially lead to slight overshoots, and even further increasing it will end in heavy oscillations. For instance, we could set the loop gain to reach a *critical damping*: We deflect the joint in hold mode while adjusting g_i, so it returns as quickly as possible without overshooting. After the loop gain's adjustment, we can continue with the feedback gain, g_f. For contraction mode, this parameter must be greater than 1. But please be forewarned: increasing feedback gain needs careful tuning! I recommend starting with $g_f = 1.001$ since a too-fast amplification of the internal state will, due to the exponential growth, quickly produce maximal power at the output.

Handling Contraction Mode near Joint Limits

For most real-life applications, contraction mode cannot simply be left in the vanilla version as described in Figure 1.5 and 1.6. Applied to a robotic joint, it would lead to the motor most likely pressing against its own mechanics at some point. Generally, a

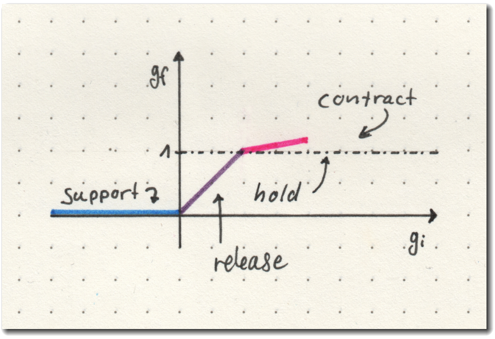


Figure 1.9: CSL parameter space restriction: To reduce the 2-dimensional parameter space of (g_i, g_f) to a 1-dimensional control mode, the parameters have been appropriately restricted. However, the specific values will need to be adjusted for each individual system. Here is a possible visualization of the parameter space restriction.

robot joint has end stops, and since a mechanical limit is recognized as the (ultimate) counterforce, using contraction mode fails to work here. Now, we could easily switch off the controller when the joint reaches its end stops. But that doesn't solve the problem of angle positions in multi-articulated robots that mutually restrict their working area. Imagine, for instance, the two arms of the same robot acting in the same space in front of the body, eventually touching each other. Since these joints will reciprocally increase their motor outputs in contraction mode, stall detection is mandatory to avoid overheating or damage.

Angle Limitation

A simple strategy for angle limitation is decaying the internal state, z, when an end stop is reached. We define inner and outer limits, $L_1^- < L_0^- < L_0^+ < L_1^+$, and calculate the decay function

$$d(\varphi) = \begin{cases} (\varphi - L_0^+)/(L_1^+ - L_0^+) & \text{if } z \geq 0 \\ (\varphi - L_0^-)/(L_1^- - L_0^-) & \text{else,} \end{cases} \tag{1.12}$$

letting $z \leftarrow z \cdot (1 - \varepsilon \cdot f_{lim}\,(d(\varphi)))$ decay to zero with rate $0 < \varepsilon \ll 1$ if an upper (L_0^+) or lower (L_0^-) angle limit[1] was reached, and decreasing faster depending on how much the limit was crossed. We start with decaying lightly at the inner limits, continuously raising to the maximum rate at the outer limits. To escape a limiting region again, we must only limit z in its effective direction and forgo decaying when it is already pointing away from the limits we currently detect.

Adaptive Limiting

For stall conditions other than those caused by end stops, e.g., the user holding tight the robot's limb or some obstacle blocking the movement, we can create a simple rule to adjust the maximum driving motor voltage used. The idea is to build a stall detection that gradually turns down the output if the maximum load sustains too long. Further, it can give a tiny input in the opposite direction, letting the controller temporarily ignore the cause. We define the *Adaptive Limits*, U^+ and U^-, as dynamic upper and lower bounds, clamping the motor output voltage u between them with

$$u \leftarrow f_{lim}(u, U^-, U^+). \tag{1.13}$$

[1] The limiting (or clamping) function's definition is $f_{lim}(\,\cdot\,, a, b) = \min(a, \max(b,\,\cdot\,))$ with default arguments $a = -1$ and $b = 1$.

The limits decrease the CSL's active range as soon as the maximal value of u is crossed, since a stall condition in contraction mode will inevitably amplify u towards its maximum through $g_f > 1$. We tighten the limits when exceeded and relax them otherwise. The adaptive limits (U^+, U^-) change according to the adaptation rule

$$\Delta U^{\pm} = \begin{cases} \eta(\mp\beta - U^{\pm}) & \text{tighten bounds if } (u(t) < U^-) \vee (u(t) > U^+) \\ 2\eta(\pm U_{max} - U^{\pm}) & \text{relax bounds else} \end{cases} \tag{1.14}$$

with $U^{\pm} \leftarrow U^{\pm} + \Delta U^{\pm}$ and with U_{max} the predefined maximal output, threshold $\beta = 0.01$ and apdation rate $\eta = 0.005$. Instead of 0 we let the bounds decrease to $\mp\beta$ (note the sign!), that will inject a tiny value into the CSL's integrator to encourage turning towards the other direction. This way, if a bound was reached it works like a repelling force.

Remark: In contrast to the angle limitation, the adaptive limiting extension to CSLs is a more recent development and was, therefore, not yet part of the Myon experiments described in the next section. I developed it specifically for its use in the *flatcat* robot introduced in Chapter 9. It is one of the measures taken to ensure that users are safe from possible harm and that the robot can cope with unknown stall conditions.

1.3.1 Experiments with the Humanoid Robot Myon

I applied Cognitive Sensorimotor Loops to the humanoid robot Myon by Hild et al. (2011) in three different experiments that offered a lot of potential follow-up applications and heavily inspired the generalized control scheme presented at the end of the chapter as the result of this process.

We originally developed Myon at the Neurorobotics Research Laboratory (NRL) at Humboldt-Universität zu Berlin. Meanwhile, the robots moved and are now part of the Berliner Hochschule für Technik (BHT). Myon is the first humanoid robot whose body parts can be completely detached and reattached during operation. All body parts retain their independent functionality as they are autonomous in energy supply, computing power, and their neural network distributing throughout the decentralized and microcontroller-based robot's computing modules. Autonomous robot parts have many advantages for research. For example, behavioral patterns can first be developed on an isolated limb (like the single leg or the lower body presented here) and then gradually extended to a complete behavior.

Experiment 1) Self-raising Leg

In the first experiment, three independent CSLs control a single leg of Myon. It started as an attempt to stabilize the leg against gravity using contraction mode, viewing it as a double inverted pendulum (refer to Figure 1.8). But when I had put the robot leg into a horizontal orientation, I recognized that it could slightly raise itself. After half an hour of manually tuning the CSL configuration, the leg could raise from lying to standing upright. The capability to work against forces in contrast to setting target angles seemed superior in reaching certain positions in phase space without pre-defining a trajectory. There was no need for a state machine to switch targets during this complex motion. The leg merely followed the path of maximal resistance, which, in this particular configuration, ended in the upright position. Figure 1.10 shows the sequence of events combined with the individual joints' angle and motor output data.

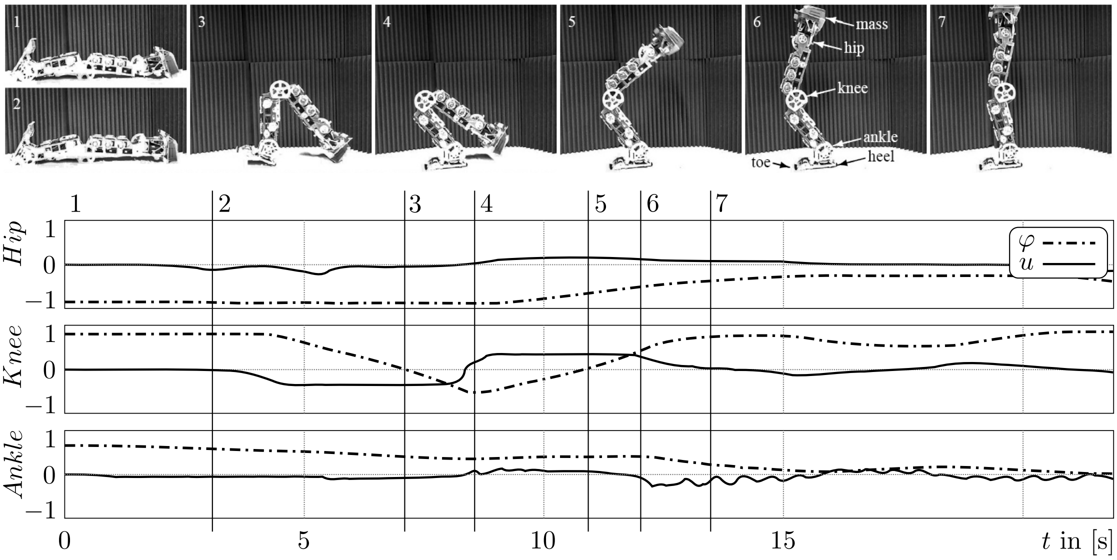


Figure 1.10: Example 1) A single leg of the modular humanoid robot Myon raising: CSLs in contraction mode independently control each of the three joints. Though there is no direct communication between the controllers, complex behavior arises solely from interacting with present forces. The essential part happens between 3 and 4, where the leg rolls over the heel so the thigh suddenly loses ground contact, forcing the knee CSL to turn direction.

Experiment 2) Balance Recovery

In the second experiment, I applied CSL contraction mode to Myon's lower torso and legs to create a robust push rejection and balance recovery. In contrast to the first experiment, we now have two contact points with the ground, leading to the robot's feet occasionally sliding towards each other. Thus, we introduced a coupling between symmetrically mounted CSL-driven joints to hinder them from working against each other but favor *cooperation.* All joints need to cooperate to stabilize the body against external perturbations. The coupling measures the activity of the joints and synchronizes the torque direction. For details, please refer to Kubisch et al. (2011a). Within reasonable bounds, the robot could recover balance even with moderate nudging or tilting of the ground, as shown in Figure 1.11. The balancing algorithm was in intense use for many *Myon Language Games* within the EU project ALEAR[2], where multiple robots performed gestures with head and arms, yielding quite a dynamical disturbance to the robot's center of mass.

[2] Please refer to the EU project results: `https://cordis.europa.eu/project/id/214856/results` (visited April 6th 2023)

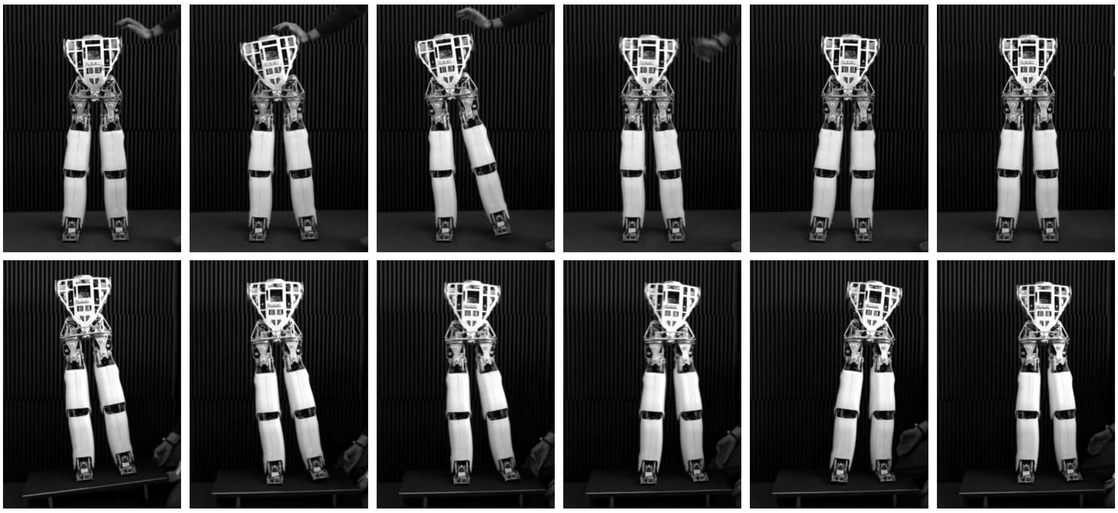


Figure 1.11: Example 2) Balance Recovery of Myon's torso with legs. The top sequence shows the torso recovering from being pushed and the bottom sequence the balance recovery after being tilted. As can be seen, the hip and torso joints start from a situation where the CSL already adapted to the tilt.

Experiment 3) Upper Body Gestures

The third experiment uses different CSL modes for creating said gestures for lifting arms and weaving, similar to how aircraft marshallers do. CSL modes change according to the current arm and body pose so that sequences of motions occur, for instance, lifting the arm and moving it away from the body when it reaches the maximal position. I altered the CSL so I could change its modes from contraction to hold mode and finally to release mode by only modifying a single control input. Concurrently, the algorithm described in the second experiment stabilizes the body.

Even though I defined the parameters for these experiments manually, I could demonstrate that applications with increasing complexity derive from a simple principle. The result encourages even more complex setups, considering more interconnections to modulate behavior and regarding the adaptation of parameters w.r.t. an objective function.

Limitations of CSL

The CSL approach shows that we can achieve complex robot behavior in principle with simple sensorimotor control loops. But most of the time, it needs a bit of sensitivity and implementation work around it to master a CSL. In particular, we must separately protect the end stops since a CSL in contraction mode tries to overcome this limiting resistance, if necessary, with maximum power.

The CSL is not suitable for positioning tasks. It is not a classical controller because there is no explicit target quantity. If any, we could say a zero velocity is the implicit target to reach. CSLs have much potential for push rejection and balance recovery, as demonstrated in the example of Figure 1.11. They are also well-suited for creating slower movements, e.g., for body poses that must overcome forces such as gravity, or when it

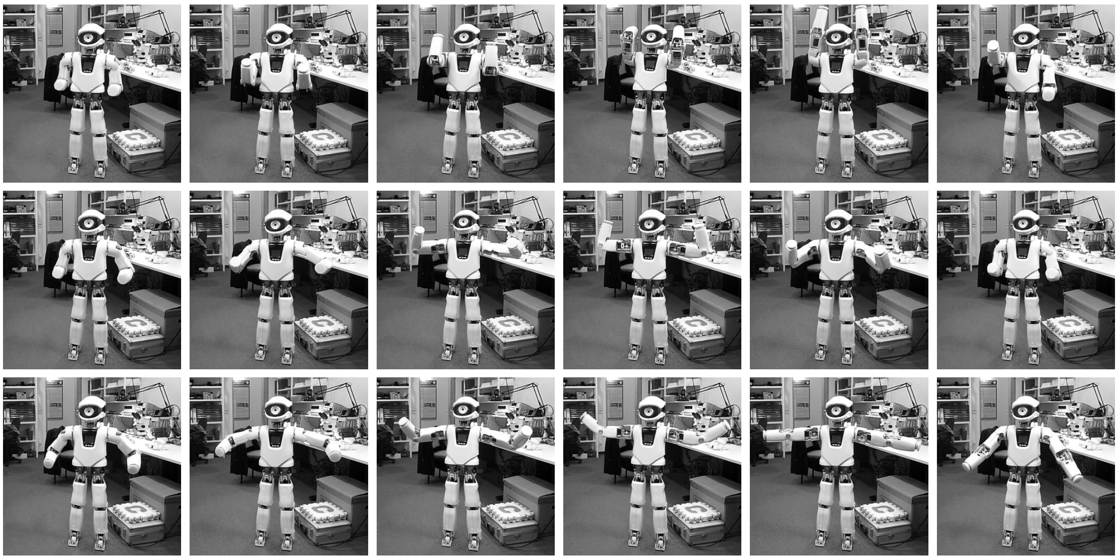


Figure 1.12: Example 3) Sequence of gestures using CSLs. Arms are raised using contraction mode and taken down using release mode. The CSLs for shoulder pitch/roll and elbow joints are activated according to the position of the other joints in order to create different contraction/release sequences. The execution speed of each gesture sequence can be modulated by different contraction strength and even halted at any time using hold mode. Since the angular momentum generated by the moving arms would easily make the robot fall over, CSL-driven joints are used concurrently to stabilize the torso and legs as already described in Figure 1.11.

comes to adapting to body model changes such as applied loads as demonstrated in the last example in Figure 1.12.

However, considering high-dynamic movements, CSLs seem to be incomplete. There are attempts to make CSL motions faster, as examined by Werner (2013), or to create more multi-joint behaviors like an assisted stand-up from sitting, as studied by Meier (2015). And even though it is possible to create a stable oscillation using a contracting CSL with a low loop gain, there is, to my knowledge, no example yet of a full-body high-dynamic motion such as walking or running for legged robots.

1.4 Non-linear Control for Multi-Joint Robots

In the previous sections, I defined the basic terms for dynamical systems and presented common controllers, as well as more recent sensorimotor loops which mostly acted on single joints. And even though they were used in behaviors for multi-joint robots, mostly their interconnections between joints were sparse. I will now present an approach of how to generalize both, the PID and the CSL for multi-joint robots. We will see that both structures can be regarded special cases of the same more general structure and that the result will be even more powerful in the sense that it also contains a lot of more functions such as logic and oscillators.

For instance, with the following structure, we will find multi-dimensional PD controllers with their setpoint created from the full body state or we could identify CSLs that change their behavioral mode dynamically for one joint—e.g., from contracting to release—depending on the state of other joints. These structures will also have the capability to oscillate, either open or closed loop.

But before we continue, we need to specify the sensor and motor space, which is input and output to the generalized controller. So far we only needed to regard a single joint's state, its angle, and angular velocity. Now we do consider robots with an unspecified number of joints. Even though it is not mandatory to do so, we shall further on assume that for each joint the angle, velocity and the control output exist and will have consistent signs. When discussing the properties it is convenient to think of revolute joints with their usual actuators and sensors, but at the end of the chapter it will be quite clear, that the intention points towards a controller structure which does not necessarily depend on having labeled inputs and outputs, rather it could be handled as if we do not know what kind of inputs we have, what the actuator is doing and how it is influencing the system. So we should have in mind that algorithms that build on top of this structure will not care about the kind of inputs and outputs, especially when it comes to learning the parameters. Rather we will change the parameters according to some objective measure as if there were no information about the model available, i.e., unlabeled sensors and actuators.

Sensor and Motor Space Definition

For our robots presented in section 2.3, the system's state $\boldsymbol{x}$ comprises all input readings. It is the observable state of the system. Staying close to hardware experiments, we only incorporate state variables that the sensors can measure or low-level features derived from said basic measurements.

The sensor vector is defined as $\boldsymbol{x} = (\boldsymbol{\varphi}, \dot{\boldsymbol{\varphi}}, \boldsymbol{v}, b)^\top$ with $\boldsymbol{\varphi} \in [-1, 1]^K$ being the vector of all joint angles, $\dot{\boldsymbol{\varphi}} \in [-1, 1]^K$ the vector of their corresponding angular velocities, optionally the estimated body velocity $\boldsymbol{v} = (v_x, v_y, v_z)^\top$ and the bias $b = 0.1$. The dimension of $\boldsymbol{x}$ is $D = 2K + 3 + 1$, where K is the number of joints. All sensors are normalized, quantized to limited resolution and a little noisy as described in section 2.1. Analogously, the motor outputs are defined as $\boldsymbol{u} \in [-1, 1]^K$ and will be treated as raw motor voltages without additional control loops involved.

Requirements

There are potentially many controller architectures, so we better stick to the simplest form that generalizes the aforementioned architectures. The possibility of embedding already found structures is enormously important here. So there are some requirements to its structure that shall be named here:

We need non-linearity to create saturation. Limiting is mandatory since we might have many interconnections from inputs to outputs, potentially leading to overexcitation and large output values. Even worse, systems with positive feedback will quickly grow out of bounds without saturation. On the other hand, the non-linearity must be monotonic and symmetric to zero for directed motor output. It must be differentiable to make it easier for optimizers to find suitable parameters.

For the beginning, we can try to keep the explicit state information low. We should not put too much into the controller, but rather consider to switch or blend controllers (automatically) if the situation changes and the current controller is not suitable anymore.

1.4.1 Generalized Non-linear Controller

For the following experiments, I define the *generalized controller*, as

$$\boldsymbol{u}_t = f(W\boldsymbol{y}_t) \qquad \boldsymbol{y}_t = \begin{pmatrix} \boldsymbol{x}_t \\ \boldsymbol{u}_{t-1} \end{pmatrix} \tag{1.15}$$

a simple-to-implement controller structure powerful enough to create complex locomotion behavior. The topology is similar to the one used by Solomon et al. (2010, 2012), who employed a linear, fully interconnected network to make bipedal robots walk down slopes and steps. The controller's core is the weight matrix $W \in \mathbb{R}^{K\times(D+K)}$ whose *weights* are subject to optimization and shall be found during learning. The transfer function is $f(\cdot) = tanh$ which squashes the motor output smoothly to the interval $[-1, +1]$ (refer to Figure 1.14). This controller can be seen as a simple neural network with fully connected weights and a differentiable transfer function, i.e., it is learnable with (stochastic) gradient descend. It is a closed loop controller featuring short term memory through recurrent connections.

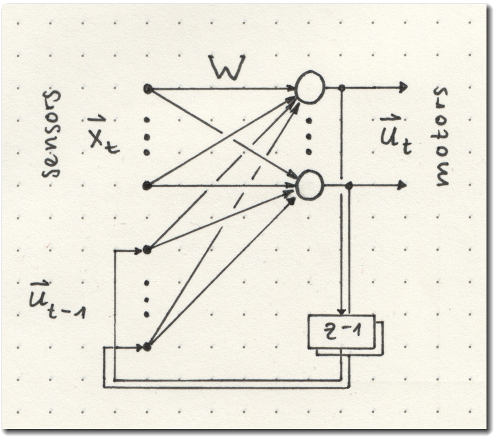


Figure 1.13: A generalized controller structure: The inputs comprise the states of all joints, $\boldsymbol{x}_t$, as well as the controller's last (delayed) outputs, $\boldsymbol{u}_{t-1}$, resembling networks introduced by Elman (1990). The mapping from input to output is fully connected and a non-linear squashing function creates saturation. A constant bias is employed through $\boldsymbol{x}$. All free parameters form the weight matrix, W.

This controller structure was selected in order to reduce the assumptions about the robot platform under test. A big plus is that this structure is general enough to embed classical PI and PD controllers as well as the already discussed *Cognitive Sensorimotor Loops*. Hence, we are able to initialize our controller's weights with known parameters and control loops which we have already found to work.

In fact, as will be seen, commonly used PI and PD controllers, as well as the CSL, are just special cases of Equation 1.15. This feature ensures appropriate start conditions for the robot in learning experiments, e.g., we can place the robot safely standing upright when starting evolution or reinforcement learning algorithms. We will make use of this

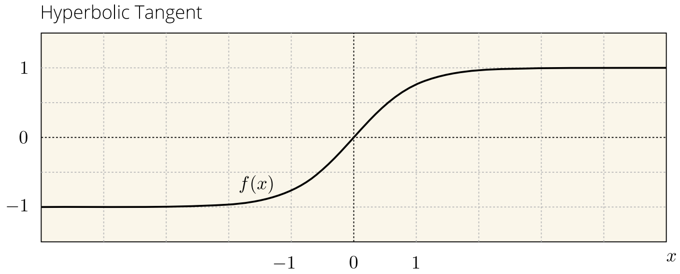


Figure 1.14: The Hyperbolic Tangent transfer function.

in Chapter 3 either by giving PD setpoints or use the hold mode ($g_f = 1$) of the CSL. However, this embedding will only be exact within the approximately linear domain of the transfer function.

Relation to Cognitive Sensorimotor Loops

The embedding of CSL is trivial, which can be seen from the short form of Equation 1.9, which is $u(t) = -g_i\dot{\varphi} + g_f u(t-1)$. Thus everything we need for a CSL for each joint is already there. Furthermore, cross-connections will allow for CSLs which are able to influence each other, e.g., inhibitory or excitatory connections, which promisses enormous potential.

Relation to PI and PD Control

The embedding of PI, PD, or even PID controllers is not directly obvious. Even though the necessary sensory inputs are available in form of the individual joint angles φ_i and their corresponding velocities $\dot{\varphi}_i$, it is neither evident what the setpoint will be nor how the integral of the deviation is calculated.

For an arbitrary joint, let u_{pd} be the output of the linear PD controller which tries to minimize the output error $e(t) = \bar{\varphi}(t) - \varphi(t)$ between a setpoint angle $\bar{\varphi}$ and current joint angle φ, and let k_p, k_d be the constant parameters for the proportional and derivative terms. Furthermore, let the setpoint angle $\bar{\varphi}$ be constant, i.e., the setpoint shall not depend on time here:

$$\begin{aligned}
u_{pd}(t) &= k_p e(t) + k_d \Delta e(t) \\
&= k_p \left[\bar{\varphi} - \varphi(t)\right] + k_d \left[(\bar{\varphi} - \varphi(t)) - (\bar{\varphi} - \varphi(t-1))\right] \\
&= k_p \bar{\varphi} - k_p \varphi(t) - k_d \left[\varphi(t) - \varphi(t-1)\right] \\
&\approx k_p \bar{\varphi} - k_p \varphi(t) - k_d \dot{\varphi}(t) \qquad (1.16)
\end{aligned}$$

What is left to do is to associate the weights for the angle input $w_{ii} = -k_p$, the velocity input $w_{i(i+M)} = -k_d$, and the bias weight as $w_{iN} = k_p\bar{\varphi}$ (assuming a constant setpoint $\bar{\varphi}$). All other weights are set to zero. We begin to imagine that the setpoint might also be composed of other position inputs and not necessarily need to be constant.

For the PI controller, we make use of the z-transformation to bring the right side of the equation in a more convenient form and proceed similarly as in Equation 1.16.

$$\begin{aligned}
u_{pi}(t) &= k_p e(t) + k_i \sum_{n=-\infty}^{t} e(n) \\
u_{pi}(t) &\circ\!\!-\!\!\bullet\ U_{pi}(z) \\
U_{pi}(z) &= k_p E(z) + k_i \frac{1}{(1-z^{-1})} E(z) \\
(1-z^{-1})U_{pi}(z) &= k_p(1-z^{-1})E(z) + k_i E(z) \\
u_{pi}(t) - u_{pi}(t-1) &= k_p \Delta e(t) + k_i e(t) \\
u_{pi}(t) &= k_p \Delta e(t) + k_i e(t) + u_{pi}(t-1) \\
&= -k_p \dot{\varphi}(t) + k_i \bar{\varphi} - k_i \varphi(t) + u_{pi}(t-1) \qquad (1.17)
\end{aligned}$$

Again we see that all elements are provided and we only need to associate the weights appropriately.

Remarks: If PID controllers shall be embedded, we would furthermore need $\varphi(t-2)$, which is not included in the current form, but would be easy to incorporate. And while the aforementioned controllers work mainly in the linear domain of the tanh transfer function, they are only approximations of their original form in the non-linear domains.

Boolean Logic

Looking at individual neurons of this structure, we can see that Boolean logic can be represented easily. Figure 1.15 shows, that in principle all logical operators can be composed of the elements contained in the controller. It is thus possible for the controller to activate or inhibit outputs depending on inputs or other outputs. Common to all of them is a high preamplification through large weights in order to saturate the signal via Hyperbolic Tangent[3]. Thus we quantize the signal to quasi-binary values close to $(-1;+1)$. Those values represent the truth values, true ($\approx +1$) and false (≈ -1). For the sake of convenience, in the figures, the bias is written into the neuron. Noteworthy is the implementation of the XOR gate. As the only exception, we need a further time step and multiple neurons due to the non-linear separability. The logic operations exemplified here come with only two inputs, but we can easily generalize them by adjusting the bias accordingly, e.g., to form multiple-input OR gates.

Relational Operators

When it comes to comparing values, neurons are also useful for creating relational operators such as $<$, $>$, and $=$. By subtracting two inputs, and amplifying the outcome, we can create a simple *comparator*, which tells us in a binary output, e.g., if $A > B$. If we further add a self-coupling[4] of greater than one, we add some hysteresis and hence stabilize the result and avoid flickering around if both inputs are already close to each other. Hild (2007) demonstrated that this is a neural equivalent of the *Schmitt-Trigger*

[3] The value of $\tanh(5) = 0.999909204\ldots$ (a transcendental number) is close enough to 1 for boolean logic operations.

[4] Please note, the use of the more convenient graphical notation of the self-coupling. For our generalized controller this is equivalent to having a connection from the delayed output $u_j(t-1) \rightarrow u_j(t)$.

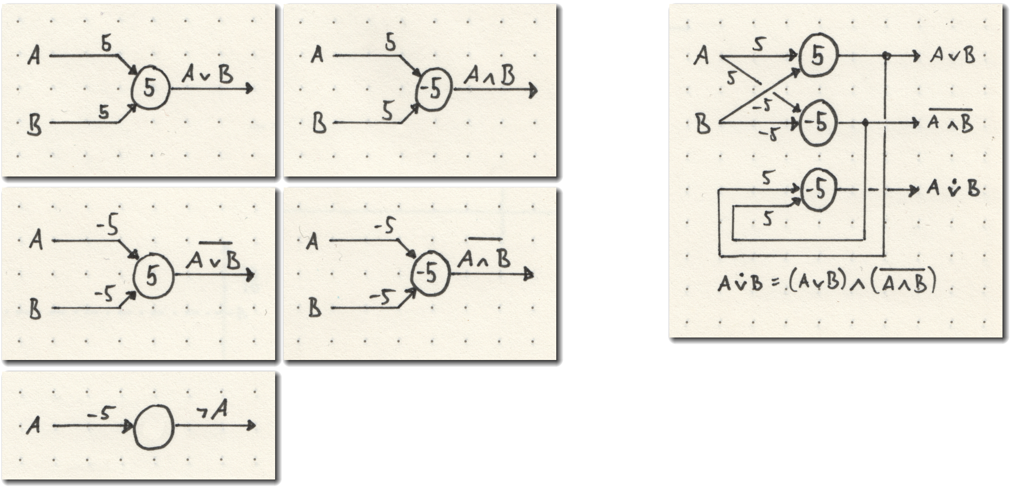


Figure 1.15: Examples for basic logic operators created using *tanh*-neurons within the structure of the generalized controller: OR, AND, NOR, NAND, NOT. The transfer function saturates the output due to the large input weights and makes it quasi-binary. Negative signs create inversions of logic inputs. Using the bias we can offset the neuron and create simple logic gates, which can be composed to form practically any gate as demonstrated with the XOR.

invented by Schmitt (1938). Since we usually work with continuous input values, the $=$ operator is not particularly useful in its current form. We get 0 when A is exactly equal to B, and small deviations are amplified. What we need for natural sensor values is an adjustable $\approx$ operator. However, through bias shifts, we can implement this function using two comparators and a subsequent AND operator. So over two time steps and three motor outputs, the function $(A > B + \frac{\Delta}{2}) \wedge (B > A + \frac{\Delta}{2})$ can be mapped creating a positive output when $|A - B| < \Delta$.

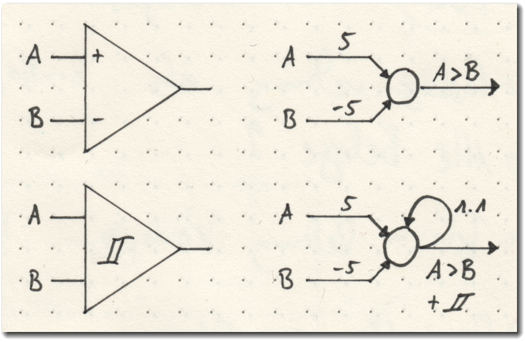


Figure 1.16: Relational operators build using neurons: The difference between inputs is amplified to either get 1 if $A > B$, -1 if $A < B$ or zero if $A = B$. When more robust representations are needed we can create some hysteresis by simply adding a self-coupling weight $w_s > 1$. The breadth of the hysteresis depends on the input weight and size of w_s.

Oscillation Capabilities

As stated earlier, we can regard the gaits of a robot as multi-dimensional oscillators. Thus, our proposed control structure must be able to create an oscillation, either on its own in a feedforward mode, as a central pattern generator, or preferably must oscillate together with the connected system as a whole. We can realize the feedforward version by utilizing an SO(2)-network as proposed by Pasemann et al. (2003). To create a stable oscillation, we only need to fully interconnect two output neurons by weights r (denoting the ring coupling) and s (denoting the self-coupling). This structure is a neural approximation to a 2-dimensional rotation matrix, rotating the output vector as

$$\boldsymbol{x}_t = \tanh(W\boldsymbol{x}_{t-1}) \qquad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \qquad W = \begin{pmatrix} s & r \\ -r & s \end{pmatrix} = (1+\epsilon)\begin{pmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{pmatrix} \tag{1.18}$$

with the angle step size θ and $\epsilon > 0$ being either a small constant to just compansate for the tanh-losses or for larger values going away from the sine/cosine outputs to an increasingly rectangular wave form. The SO(2)-network can likewise be seen as two interconnected low-pass filters with negative feedback. Due to the low-pass filter's *inertia*, the negative feedback has no immediate but a delayed effect.

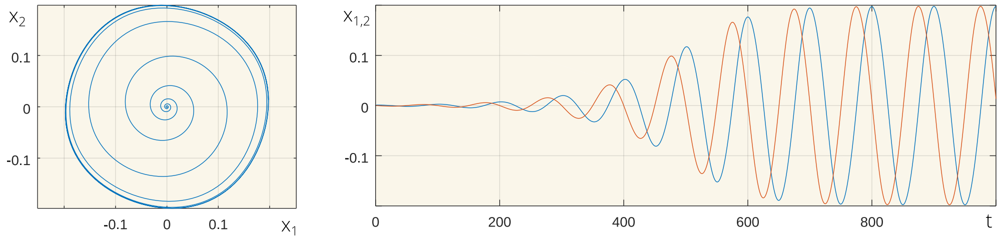


Figure 1.17: Output of an SO(2)-Oscillator with inital condition $\boldsymbol{x}(0) = (0.001, 0)^\top$, $\theta = \pi/50$ and $\epsilon = 0.01$. Depending on $\boldsymbol{x}(0)$ the oscillation increases slowly, a feature useful for initiating a walking with smaller steps.

For the second (preferred) case, we incorporate the hardware and naturally get the inertia since neither the motors nor the attached limbs will immediately react to inputs but take some time to gain velocity. Due to their mass and inertia, they withstand an instantaneous acceleration and act like the required low-pass filter. In contrast to the feedforward generated oscillation, we will refer to them as *inherent oscillators* as they will stop when the hardware does. They directly react to the environment and oscillate synchronically with the controlled limbs.

Referring to Figure 1.18, we see a single simplified limb assembled as an actuated revolute joint acting on a lever with mass. We have at least two possibilities to create an inherent oscillation with such a simple system. First, we can measure the deviation angle, derive the velocity, and positively feed it back to the motor. Positive velocity feedback correlates to the CSL in support mode, but due to the restoring force caused by gravity, the system will soon start to oscillate if we deflect it a little from the origin. We achieve a stable oscillation when we adjust the loop gain g_i just as high as to compensate for the dry friction.

The second possibility to create an inherent oscillation is similar to the SO(2)-network, but instead of having two neurons, we replace one by the hardware. Again we measure the angle, but then feed it back negatively and delayed by the integration, so that we create an *overtuned* I-controller. In contrast to a regular I-controller our setpoint is implicitly set to zero and the integral term's amplification is set a little too high so that we intentionally create overshooting and increase the gain even further until the oscillation got large and stable enough. While the first variant would oscillate around the equilibrium point with respect to gravity, the second one would oscillate around the zero position.

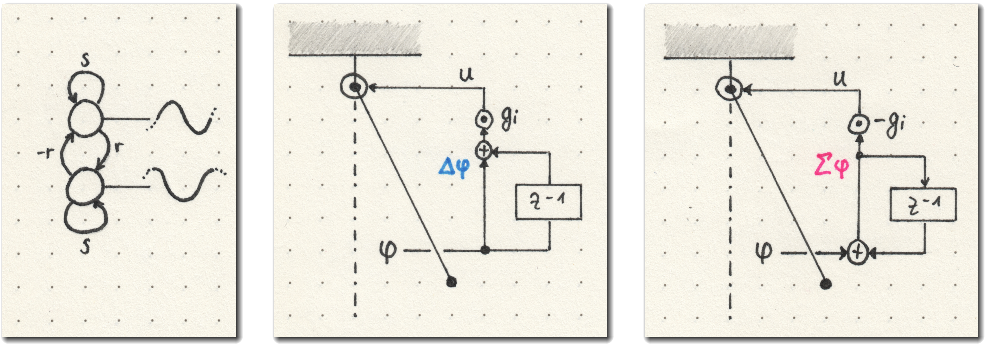


Figure 1.18: Demonstration of oscillators that can be embedded in the generalized controller: The SO(2)-Network as a feedforward *driving* oscillator (on the left) and two additional mechanisms to create *inherent* oscillation going through the robot's physics using a single joint. A positive velocity feedback oscillating the joint around the equilibrium point with respect to gravity (shown in the middle) and a negative delayed position feedback oscillating around the zero position (on the right).

1.4.2 A Simple Bipedal Gait Pattern

When working at the NRL, I was eager to make the robot Myon walk its first steps most simply. The basic idea behind Myon's walking behavior was to generate a clock signal from the robot's sensory data and feed it back to the motors' PD controllers. This idea originates from the assumption that during a periodic walking motion, the sensors would also generate periodically recurring signals and that negative feedback of these signals, including a phase shift induced by the body's inertia, would lead to an oscillation, as described in the last section.

With this principle, we can generate periodically recurring motions, as demonstrated by Werner (2008) and Kubisch et al. (2011b) for a small-scale humanoid robot, including forward and backward walking, sideward stepping, turning, and the like. Exceptions are very slow or non-periodic movements. There is some similarity to the (open-loop) central pattern generator approaches with the difference that the robot's body dynamic generates the periodic pattern (closed-loop).

For walking, the created behavior is symmetric, i.e., the left and right half of the body perform the same movements apart from phase shifts. Hence, the structure requires a sensor space calibrated almost symmetrically to the sagittal plane. We assume the robot's joints come in pairs, so the symmetry exists, and the parameter space is reduced meaningfully to the most desirable symmetrical walking behavior. However, there is

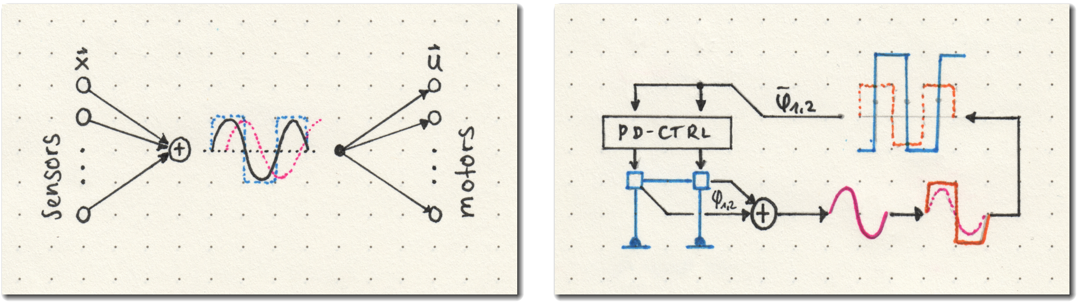


Figure 1.19: The basic idea of a simple Myon walking (left): A periodic signal is generated from the sensor data and negatively fed back to the joint motors. By suitable choice of parameters, it has so far been possible to develop forward, backward, and turn-around. Analogous to idea 3, an oscillation emerges through the robot's body dynamics. The sensor signals are actively delayed and fed back negatively to the actuators, resulting in oscillatory motions. Simplified illustration of the practical implementation of the basic idea (right): The signal originates at the hip angle sensors, gets processed separately for each joint pair, and feeds back as setpoints to the joints controlled by PD controllers.

nothing wrong with partially softening this symmetry, for example, to achieve directional or sideways steps. The Figure 1.19 shows the idea schematically.

I could identify the hip roll angle sensors as appropriate sensory input data for the gait presented here. Either joint angles increase as their corresponding legs move outward. Subtracting one signal from the other represents the width of the legs' shearing. This signal is saturated, phase-shifted, and eventually returned to the joints. The joints, controlled by simple PD controllers, will compliantly try to keep their target angles. Our pre-processed signal comes as an offset to this target angle to shift the desired position of the leg. For each joint pair participating in the derived movement, we must define the default target position, the coupling strength offset, and the phase shift.

The picture series of Figure 1.20 depicts the resultant gait. Although this walking gait appears penguin-like, it demonstrates that even with minimal parameter tunings, a considerably simple and stable walking pattern could be achieved (even without any help of automated parameter search or modeling). This motion principle is almost compatible with the general controller architecture discussed in this chapter. All building blocks, except the explicitly shifted phase, are available. However, when feeding the inputs not exclusively from a single source—the hip roll joint angles in Myon's case—but rather feeding them from several sources, e.g., other joint angles or velocities, there is much potential for finding sources with the correct phase.

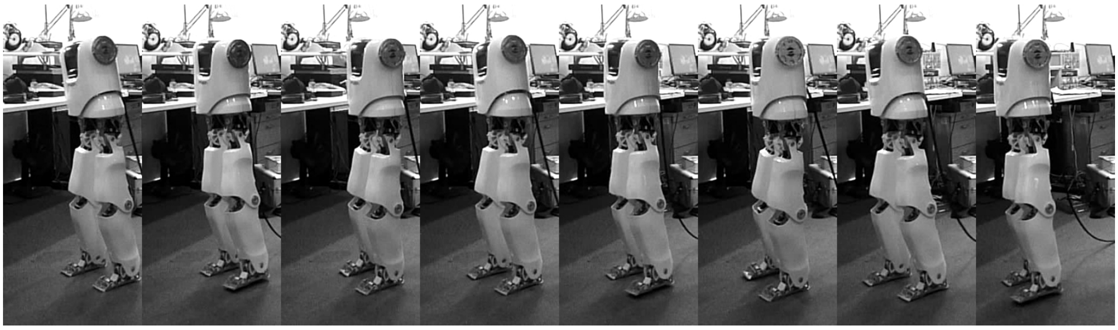


Figure 1.20: The Myon robot's trunk with legs while walking small steps: A penguin-like gait pattern, similar to toddling, emerges from reconstructing a periodic (or clock) signal from sensory data and feeding it back to the leg motors with individual phase and amplitude. The created dynamical system exhibits co-existing attractors, such as a fixed point for standing and a quasi-periodic orbit for toddling. An intended laterally applied nudge suffices to leave the equilibrium and enter the basin of the stable gait pattern's attractor. Vice versa, gently holding the robot stops the oscillation again.

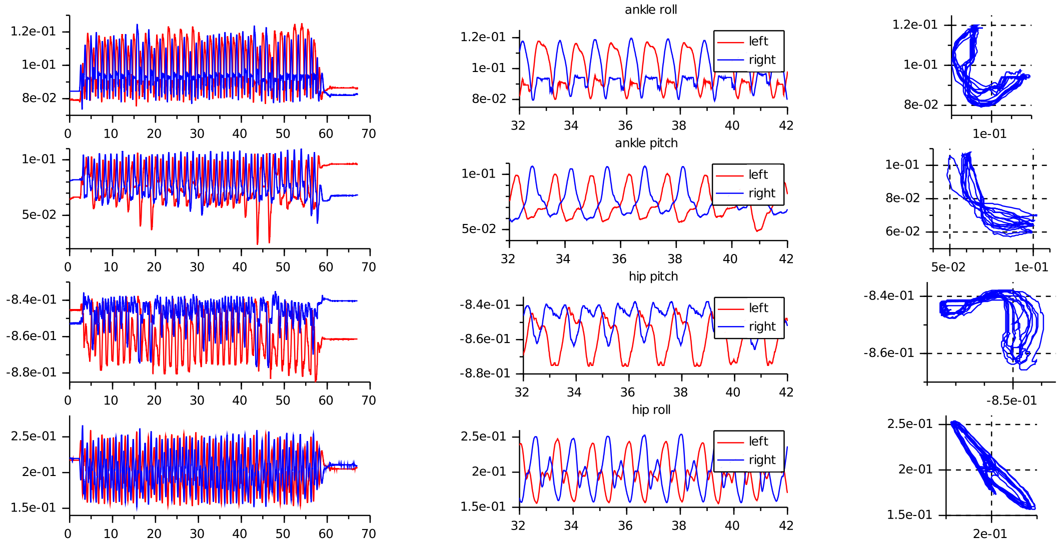


Figure 1.21: Sensory data of Myon walking: We see one minute of Myon's joint angle data while the robot walks approximately 42 steps. The roll/pitch angles of hips and ankles are majorly involved. The other leg joints are less active in the gait pattern and only compliantly hold their positions. The robot was walking over ordinary office carpet, so the walking is robust enough to compensate for little disturbances due to floor irregularities and ground friction. The gait always returns to the attractor again. The data shows the time development of four leg angle pairs (time in seconds). A 10-second section of the complete time series highlights the approximate symmetry between corresponding left and right pair motors. Additionally, a 2-dimensional plot shows the partial projection of the joints' phase space.

1.4.3 Reducing the Number of Weights

The weight count for a robot using the above defined generalized controller is $n = K(D+K)$, for K joints, e.g., a simple 4-DoF robot already has $n = 52$. So a major drawback of the aforementioned controller is that the weight matrix grows quadratically with the degrees of freedom. In order to reduce the number of weights, we can apply the following two constraints:

Symmetries

First, we share weights by using inherent body symmetries. For behaviors symmetric to the sagittal plane such as walking forward or backward, we can constrain

$$w_{ij} \overset{!}{=} w_{ji} \tag{1.19}$$

if i, j are symmetric joints. For instance, imagine the motor output of the right ankle joint is composed using the velocity input of the left knee, then, in turn, the left ankle's output must be composed using the right knee's velocity input. We will refer to this as the *symmetry constraint.* However, this can only be employed to non-symmetric behaviors (such as turning), if there is enough non-symmetric sensor input available to help breaking symmetry.

The symmetry constraint reduces computational effort and learning time. For instance, walking forwards using symmetrically constrained gaits is more likely to move with less curvature. If the fitness function is to move straight without leaving the track, then it probably takes more time for the non-symmetric gait to get straightened. In the end, the result might be similar, but it usually takes longer and the resulting weight matrix has more unique parameters. Even though the behavior may finally evolve symmetrically, the parameters must not necessarily evolve symmetrically too. There is plenty of possible configurations yielding the same behavior for the general case, but this could lead to different behavior in the edge cases, depending on the inputs. So e.g., stepping over small obstacles will look different depending on whether it is the left or right leg stepping first. The reduction in computing time is accompanied by a reduction in memory requirements for such weight matrices, which is particularly interesting for microcontroller-based systems.

Normalization

Second, we can encourage sparse weight matrices using L_1 *normalization.* By adding a penalty term

$$P = L_1 : ||W||_1 = \sum_{ij} |w_{ij}| \tag{1.20}$$

to the cost function C we get a new cost function $C^\mu = (1-\mu)C + \mu P$, with μ as the regularization strength. Two things happen here: Large weights are discouraged, which is good since they tend to drive the inputs towards saturation of the transfer function where learning is usually slower because the gradient starts to vanish. On the other hand, using the absolute value function for penalizing individual weights tends to drive one weight towards zero in favor of another. For example, the original cost function C

may look similar to the one depicted in Figure 1.22 with each line denoting the points of the same value (contour plot) for two arbitrary individual weights w_i and w_j. If we add a penalty term P using the absolute value function this penalty term will overlay a rhombus-like contour and the resulting cost function C^μ will have its minimum closer to the axis, where one of the weights may become zero. Finally, weights that are sufficiently small (i.e., beyond a certain threshold) are considered as not needed and are set to zero (truncation). Zero weights are computationally more efficient than very small weights and they also point out where the relevant structure is. With their help, we can identify inputs or outputs which are less important for a certain stable gait.

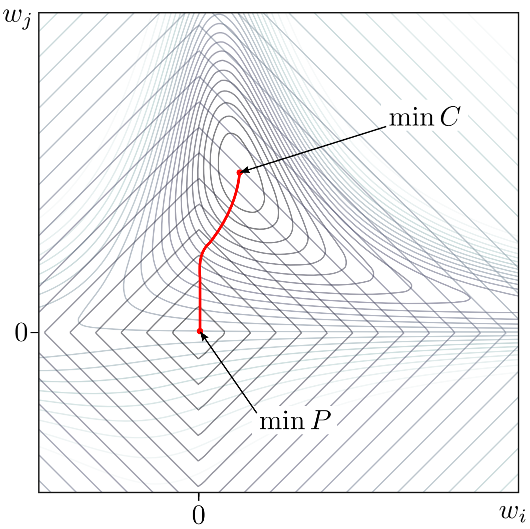


Figure 1.22: Enforcing sparse weight matrices by L1-normalization. Increasing the penalty term drives weights closer to zero. The red line shows the min C^μ trajectory for $\mu \in [0, 1]$ of two weights while fading cost functions from C to P.

Conclusion and Discussion

The discussed control structure generalizes methods with which motions for real robots like Myon could be successfully implemented. These motions, including reactive walking were created in mostly isolated experiments with only a few limbs involved. This was due to the fact that in all cases, the parameters were selected manually and fine-tuned by human observation. The generalized controller now holds the potential not only for these structures but also enables us to use much more inputs, outputs and cross-connections between joints. The multitude of new parameters can hardly be determined by hand anymore and must be found with the help of optimization algorithms. Nonetheless, starting with good initial values is still possible because the known structures can be used almost unchanged. The controller is prepared so that automated optimization can be applied. In Chapter 3, the focus is to explore what potential the controller can achieve beyond what has been seen so far. To set the ground for these experiments the next chapter introduces the simulation environment and the robot platforms to extensively test the structure.

There are a lot of possible extensions to that structure. We could easily introduce hidden layers or time embeddings to improve the versatility of a single controller to make it more capable. But with every bit of complexity we add, we make the analysis a lot harder. If we keep the controller simple we gain a better understanding, a better pretraining, easier implementation, and testability, as well as less computational resources needed. However the most promising extension from my point of view is to add *context* neurons $\boldsymbol{c} = (c_0, c_1, \dots)^\top$ which are not dedicated as motor outputs. This brings additional *memory* to the structure but is neither expected to make the current results invalid nor does it dramatically increases computational complexity. The generalized controller with context neurons would be defined as

$$(\boldsymbol{u}_t, \boldsymbol{c}_t)^\top = f\left(W\boldsymbol{y}_t\right) \qquad \boldsymbol{y}_t = (\boldsymbol{x}_t, \boldsymbol{u}_{t-1}, \boldsymbol{c}_{t-1})^\top \tag{1.21}$$

as if there would be additional output neurons that are not connected to motors but whose state will serve as input to the next step. However for the sake of simplicity, let us move forward to exploring the potential of what we have discussed already.

2 Simulated Environment and Robots

Simulation has long been a useful tool in robotics for testing ideas and implementations. The ability to run simulate robot behavior faster than real-time is an enormous gain, in particular when it comes to learning tasks. Simulated worlds are robust and it is possible to test algorithms on robot hardware free of wear and with lowest possible costs. With open-source simulated physics like *Open Dynamics Engine*[1] or complete simulation environments like *Gazebo* by Koenig and Howard (2004), it is possible for everybody to have access to simulated robots and to test and compare their algorithmic ideas.

There will probably always be a difference between simulation and the real world because simulation is only an approximate model of the environment. To steadily improve these models is subject to ongoing research, so with increasing precision of both, physics engines and models, the *simulation-to-reality gap* will further decrease. But we as roboticists should not focus too much on it, but rather adapt our algorithms to robustly deal with a certain degree of model variation—just as wear and production tolerances can create deviations between two identically manufactured robots. We better accept that the world is complex and focus on creating robust methods.

Just to name a special candidate, friction is still hard to simulate. Bodies interact at the molecular level and models for macroscopic objects are no longer exact. So non-linear effects like static friction are still hard to handle.

In simulations, there is a constant need to balance between speed and accuracy. Highly detailed models and small simulation step sizes demand significant computing resources. But we must try to keep the total simulation time low, since most experiments will take many iterations to come up with meaningful results. And the advantage of running simulations faster than real-time can quickly disappear as the quest for precision intensifies. As a matter of fact, simulated system outcomes will never precisely replicate the real world. However, by designing simulated robots that function with closed loop controls, adapt themselves and do not make explicit assumptions, we can expect better transferability of simulated results into the real world.

This chapter introduces the simulation environment with which all simulated results of this thesis have been produced. We start with how the robots are modeled and set a special focus on the actuator design, especially when it comes to friction. After that, the robots and their environment used for the simulated learning experiments are presented.

[1] Open Dynamics Engine, initial release by Russel Smith 2001: `www.ode.org`

2.1 Simloid

Simloid is a robot simulation environment started by Hein (2007). It is lightweight, open, and fast. Simloid is written in C++ using OpenGL for visualization. The underlying physics comes from the Open Dynamics Engine (ODE), which is also open source, widely used, and documented well. The ODE v0.13 used in this thesis supports rigid body dynamics and collisions but cannot simulate soft materials. Within the scope of this work, all the simulated robots will consist of rigid bodies with more compliant collision behavior and less strict joint axis constraints. The simulation step size is 10 ms, i.e., 100 Hz, which has turned out to be a good compromise between acceptable numerical accuracy of the physics and moderate computation speed.

Simloid has a TCP/IP interface to read status messages and send control commands to the robots. It also transmits the robot's characteristics. The client can set specific environmental settings or reset, safe, and restore simulation states. Since the interface decouples the simulator from the user's control program, it is possible to start multiple simulation instances on different remote machines and run several simulations in parallel. By running multiple instances, as we will see later, we can conduct evolution experiments by separately evaluating trials of individuals from the current generation.

In each simulation step, the client sends its new control commands, the simulation applies the controls, calculates the physics, and finally transmits the current sensory and environmental state back to the client. The protocol allows for a thin communication channel. I developed an optional *interlaced mode*, making the simulator execute the motor commands *while* the client is calculating. It brings the simulation closer to the actual machine's behavior and helps to bridge the reality gap, a topic we will discuss in section 8.3.

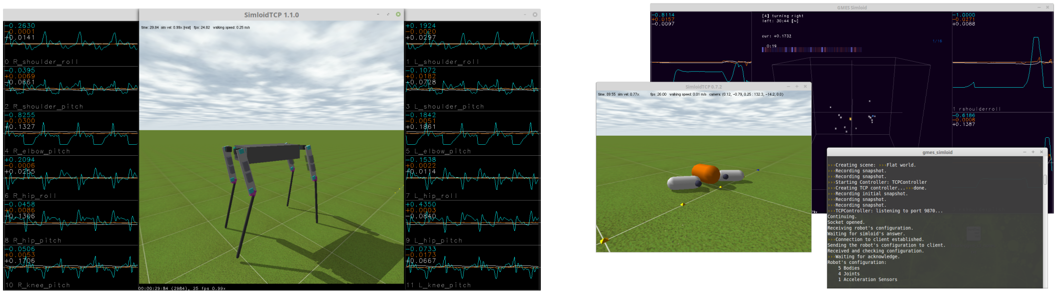
Even though there are versatile options for creating joints, I restricted myself to using only the *hinge* (or *revolute*) joints for all robots. Most of today's legged robot constructions are similar. For instance, a motorized ball joint with all three axes going through the same point is trickier to fabricate than employing three separate servos with displaced axes. Generally, robot engineers use a single type of motor throughout the robot's body, identical for all joints.

Ground and body friction is approximated as Coulomb friction. We can adjust it by defining specific material-to-material coefficients. Air friction is neglected. But I implemented a more sophisticated joint friction model as described in section 2.2 because the actuated joints presumably have the most significant influence on the resultant robot motion.

To further increase the robustness of walking gaits against external disturbances, it is possible to instruct the simulation to apply additional forces to specific robot body parts. With this method, we can create scenarios in which the robot gets specifically disturbed by nudges while trying to walk.

Bodies and Joints

The simulated robots I used for locomotion self-learning consist of simple shaped body segments and revolute joints. A servo motor actuates each joint. I omitted everything



```
>>>Creating scene: >>>Flat world.
>>>Recording snapshot.
>>>Recording snapshot.
>>>Starting Controller: TCPController
>>>Creating TCP controller...>>>done.
>>>Recording initial snapshot.
>>>Recording snapshot.
>>>Recording snapshot.
>>>TCPController: listening to port 9870...
Continuing.
Socket opened.
Receiving robot's configuration.
Waiting for simloid's answer.
>>>Connection to client established.
Sending the robot's configuration to client.
Received and checking configuration.
>>>Waiting for acknowledge.
Robot's configuration:
    5 Bodies
    4 Joints
    1 Acceleration Sensors
```

Figure 2.1: Illustration of the Simloid simulation environment.

unnecessary for walking. The simulated robots have no counterpart in the real world. I do not consider building them like this from hardware. But simplicity applies to body design only. Motors and sensors that majorly affect behavior have more realistic simulation models. I used basic symmetrical shapes, such as cuboids or capsules, for the robot segments and gave them equally distributed mass. Each body part can have individual surface properties, for instance, increased friction between feet and ground.

The maximum joint range is $[-\pi, \pi]$. Each joint has configurable end stops, constraining the limb's range of motion. Alternatively, they can run freely, e.g., for the pendulum. A joint always connects exactly two bodies. Its center of rotation can have an arbitrary position, i.e., it must not necessarily lie within a body. For simplification, all joints are either of type pitch, yaw, or roll, meaning that all joint rotation axes are parallel or perpendicular. However, this restriction seems to be commonly used in robotics since it lowers the modeling, implementation, and construction effort. Two bodies connected by a joint never collide except for their predefined end stops.

Sensors

Each joint has angular position and velocity sensors. The angle φ with range $[-\pi, \pi]$ is mapped to $[-1, 1]$. The angular velocity ω is normalized by a maximal value of 12 rad/s. When using free run, we should mind that the angle has a discontinuity when moving from 1 to -1 and vice versa. We have to take this into account when conducting experiments with freely running joints, e.g., pendulum experiments.

To stay closer to reality, the maximal resolution of the sensors is intentionally limited to 16 bit and can even be reduced down to 10 bit. Additionally, the sensors are explicitly made noisy by adding Gaussian noise with a standard deviation of 4 bit. This means there is a specific but acceptable loss of information that the controllers and learning algorithms have to deal with.

For each segment, we can attach a 3-dimensional accelerometer with the sensor's axes aligned to the segment's main axes. These sensors measure the acceleration forces influencing this body including gravity. The output values of these sensors are restricted to $[-8g, 8g]$ and mapped to $[-1, 1]$ each. As with the position and velocity sensors, the resolution is artificially decreased and Gaussian noise is added. The orange color of a robot's body in simulation figures denotes an attached accelerometer (mostly used for the trunk).

Real accelerometers are usually implemented by measuring the deflection of a proof mass which is attached to a reference frame by a spring-damper-system. For illustration see Figure 2.2. The output value is an estimation of the spring-reaction force $F = k\Delta x$. The damping η of the system must be close to the so-called critical damping $\eta = 2\sqrt{km}$ to avoid unwanted oscillations.

For the implementation of virtual accelerometers within Simloid, we derive an acceleration value $\ddot{\boldsymbol{x}} = \frac{d}{dt}\dot{\boldsymbol{x}}$ from the corresponding body's velocity (provided by the simulation) where the sensor is attached to. We subtract the vector of gravity to get our sensor's output signal $\boldsymbol{r} = \ddot{\boldsymbol{x}} - \boldsymbol{g}$. Since when at rest, a real accelerometer measures gravity, but while in free fall, it measures zero acceleration.

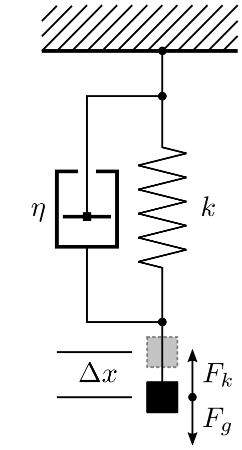


Figure 2.2: Simplified structure of a 1D accelerometer. The motion of the outer frame yields a deflection of the proof mass m, the sensor's output is proportional to the measured displacement Δx.

Electromechanical Motor Model

The robot's actuators are simulated electrical drives for which the input signals are conceived as pure motor voltages. It is considered a simulated gear motor. There is no additional internal control loop, no position or velocity controller. The motor's input, denoted as u, has a valid range of $[-1, 1]$ which is internally mapped to $\pm U_s$, the supply voltage.

The electrical characteristics are simplified in the sense, that some properties are neglected by the model, others are subsumed. For instance, the gear reduction is included in the motor constants and the motor's inner friction is subsumed in the joint friction model of the next section. Temperature effects are neglected fully. The basis for the parameter choice is the Robotis RX-28 smart servo. This versatile servo motor was used for all joints of the Humanoid Robot Myon. My choice of utilizing the properties of this servo is therefore based on personal work experience while creating motion control loops with this robot. Fortunately, Bethge (2014) already created a detailed model, inferred parameters and proposed an implementation and on whose results the used model is based.

A DC motor is usually characterized by the speed constant k_n and the torque constant k_M, the coil's resistance R and the supply voltage U_s. The inductance L would have a low-pass filter effect to the current, $U_L = L\, dI/dt$, but is neglected since the simulation step size of $\Delta t = 10\, ms$ is too large to make this effect visible. Also, DC motors are usually driven by PWM outputs of H-bridges with frequencies in the kHz domain. Rather the voltage is assumed as constant for the time Δt in this model. See the simplified electrical model schematics in Figure 2.3 and Equation 2.1.

The electrical motor model is given as a simple set of equations with associated parameters given in Table 2.1:

$$U_i = U_s\, u \qquad U_b = \omega / k_n \qquad I = (U_i - U_b)/R \qquad M = k_M I \tag{2.1}$$

which transforms the input signal u and the rotational speed ω into output torque M

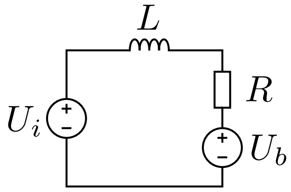


Figure 2.3: Schematics of the equivalent electrical circuit of a DC motor with input voltage U_i, coil resistance R, as well as the back EMF voltage U_b. The coil's inductance L is intentionally omitted in the Equation 2.1.

with current I, back EMF voltage U_b and. The model has no internal state.

The only mechanical component being modeled currently is the gear transformation with the given ratio of $G = 1 : 195$. Backlash and gear deformation were not modeled since the RX-28 uses metal gears with very low backlash and deformation. However, when using motors with plastic gears, deformation becomes strongly influential and cannot be neglected anymore, as I needed to learn in Kubisch (2008). A good overview and special implementation tips on modeling and handling the effects of backlash and friction in control systems is provided by Wescott (2014).

Even though there is a lot of friction in the gears and some in the ball bearings, the friction model is not considered as part of the actuator model but instead regarded a joint property. This way the friction model can also be applied to non-actuated joints. The next section describes a more elaborate friction model which I use to simulate the robots' joint friction.

U_s	supply voltage	16 V
k_n	speed constant	0.48 rad/Vs
k_M	torque constant	1.39 Nm/A
R	motor coil resistance at $65°C$	9.59 Ω

Table 2.1: Motor parameters determined by data sheets and measurements conducted by Bethge (2014). The gear reduction ratio of the servo is 1:195 with efficiency 62.5% which is already included in k_n and k_M.

2.2 Modeling Friction

Motors are the robot's interface to the world and naturally rely on friction since we have mechanically moving parts such as bearings or gears. When applying motor control loops to machines, we are facing the difficulties of the non-linear effects of friction, in particular when using low-cost gear motors. Probably the most bothering friction effect is static friction or simply *stiction*. It affects our motors in the domain of low velocities and directional changes. Therefore, we must employ appropriate friction models in our simulations of walking robots, avoiding the dependence of our results on too many idealized assumptions.

If the results of simulations are to say anything about real machines, then friction is an essential property to handle. However, when I started, it turned out that my physics simulations used only very coarse approximations of friction. If I wanted a more realistic model, I would need to implement it. Therefore, I took the—from my perspective—most promising modern friction model and made it work for the ODE-based Simloid environment. Meanwhile, the model and my implementation approach changed, so I gave it a new name, but the original (excellent) idea still resides in it. But let us start with the basics of friction modeling.

Different Types of Friction

Friction has various effects. When two objects move with non-zero relative velocity and their surfaces are in contact, a force acts against their motion. If speed and surface properties remain the same, this force is constant, too. We know it as *dry* or *Coulomb friction*. When the surfaces are lubricated or when we consider objects moving through liquids, this friction force becomes velocity-dependent and usually increases with higher speeds. We call this *fluid friction*. Regarding particular slow movements, we have to deal with *stiction*, a notable increase in friction force near zero velocity.

When the motion stops completely, the friction force compensates for all external forces until they sum up to a certain threshold, the *break-away force*. If the external forces exceed it, the motion begins abruptly, quickly decreasing the friction force. It showed that when the bodies at rest are subject to external forces lower than the break-away force, they actually move a little but will return to their original position if the external forces disappear. They act similar to bodies connected with tiny springs. Their positions' deflection is called the *pre-sliding displacement*. Stiction and the break-away are usually the most significant non-linear effects of friction. We can spot it easily in everyday life, but the more subtle consequence is the pre-sliding displacement. For an extensive overview of friction modeling for control applications, I recommend reading Olsson (1996).

2.2.1 Modern Friction Models

Currently, we must accept that friction can only be modeled approximately. There are various models available from the literature covering most of the experimental observations. But there does not seem to exist a general model, so the choice is left to the experimenter to pick one out of the many and make it work for the desired application. Olsson et al. (1998) state it as follows:

> «There are many different mechanisms. To construct a general friction model from physical first principles is simply not possible. Approximate models exist for certain configurations. What we look for instead is a general friction model for control applications, including friction phenomena observed in those systems.»

Static Models

Fortunately, nowadays friction models can already deal with the most crucial effects observed in experiments but the underlying mechanics of the surfaces' particles are far too manifold and non-linear to be captured by a single equation. The simplest form

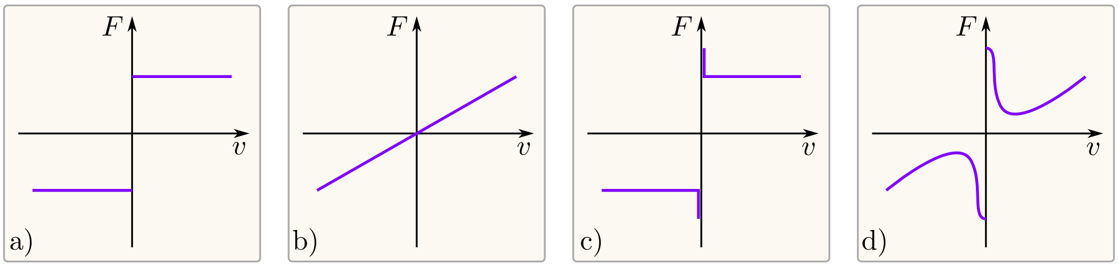


Figure 2.4: Static friction models: a) Coulomb (Dry), b) Fluid, c) Coulomb with stiction, and d) Stribeck's Curve: A combination of the effects of Coulomb, fluid and static friction.

of modeling friction is with static models. Such models do not have an internal state, instead, they are merely non-linear functions of velocity, $F(v)$. This is the relative velocity of the concerned bodies at the contact point. Static models are usually described by Stribeck's Curve as published by Stribeck (1902), and which is depicted in Figure 2.4 d). This experimental observation is a combination of the constant Coulomb friction (a), linear (to quadratic) fluid friction (b), and increased static friction (c) near zero velocity.

For use cases where we have to deal with steady velocities only, such static models are sufficient. But when using static models for dynamic motions, i.e., motions with a lot of changes in speed and direction, static models become inaccurate and numerically problematic. The problem is that we need to *detect* when the velocity is zero. Due to the discontinuity of the friction force at zero velocity, this can easily lead to oscillations. For near zero velocity, we need to know the external force F_{ext}, either measured or provided by the simulation. Consider the bodies are at rest, i.e., velocity is zero. Stribeck's curve is not defined in that point. But the friction model must react with a force F_R with the exact absolute value but the different sign in order to keep the forces equal, $F_{ext} = -F_R$. Otherwise, the bodies would never really rest. For some simulation environments, the external force may not be easily available. Approximating the external forces from the next time step's change in velocity is also not a good idea. A small time difference between action and reaction may produce unintended oscillations when dealing with larger restoring forces. Ideally, we would use models which do depend on the current velocity only.

Dynamic Models

Dynamic friction models have an internal state, or memory. They behave differently with respect to the current history of changes in velocity. A promising modern dynamic friction model is the LuGre model by Canudas De Wit et al. (1995); Olsson (1996) and Olsson et al. (1998), which covers most of the static and dynamic effects caused by friction, such as dry, fluid, and static friction, as well as the sudden break-away, and pre-sliding displacement.

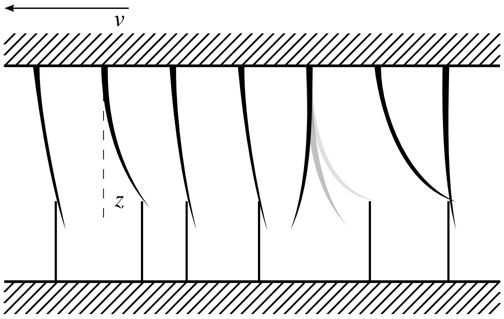


Figure 2.5: Simplified illustration of the LuGre friction model. Consider virtual bristles which are deflected when the surfaces move relative to each other with velocity v. An internal state, z, represents the averaged deflection of the bristles. Friction is conceived as a non-linear restoring force of that deflection.

The LuGre model consists of a set of equations and is defined as:

$$\frac{dz}{dt} = \dot{z} = v - \frac{|v|}{g(v)} z \tag{2.2}$$

$$g(v) = \frac{1}{\sigma_0} \left(F_C + (F_S - F_C) e^{-(v/v_S)^2} \right) \tag{2.3}$$

$$F = \sigma_0 z + \sigma_1(v) \dot{z} + \sigma_2(v) \tag{2.4}$$

with F denoting the friction force, v for velocity, and z for the internal state. The resulting friction force Equation 2.4 is a function of velocity and the internal state z (the averaged bristles' deflection) as well as its time derivative $\dot{z}$.

The function $g(v)$ in Equation 2.3 is a Stribeck kernel and regulates the transition from stiction to Coulomb friction and its parts are given as functions

$$\sigma_1(v) = \sigma_1 e^{-(v/v_d)^2} \qquad \sigma_2(v) = F_v |v|^{\delta_v} sgn(v) \tag{2.5}$$

are the stiffness $\sigma_1(v)$ and damping $\sigma_2(v)$ of the bristles. The differential Equation 2.2 forms the core of the model; it is an integrator which has the current velocity as input and accumulates an estimate of the deflection.

Problems with Dynamic Models

Admittedly, dynamic models such as LuGre bring other difficulties: They usually need a lot of parameters to be determined, or they need an additional discretization step when using these continuous time models in the discrete time domain. But most crucial, there are inherent problems with force-based approaches. High friction needs high restoring forces. In combination with lightweight objects and discrete time simulations, we have another potential oscillation problem here. In previous work, Kubisch (2008), I extensively studied this model and found it not trivial to apply for my application. Ideally, a friction model should not output a force but a target velocity and we should separate the problem by using a velocity controller which is stable for the given simulation step size.

For walking robots, the joints' velocities will change with each step the robot makes. Also when the robot is just standing and merely performing balance recovery, zero crossings and low velocities are the general cases and steady velocities are mostly exceptional. In short, we have the need for dynamic friction models here. In the next section I present a simple dynamic friction model for discrete time simulations with large step sizes which I use for the joint motor friction modeling within Simloid. The main reason for me to create my own model was, that I discovered oscillation problems with *force-based* approaches, when using higher restoring forces and lightweight objects. This problem further increases with larger simulation step sizes. So I considered a *target velocity* approach to circumvent stability issues. For the new model, I integrated the most important features learned from the LuGre dynamic model, but used a different approach to apply the friction force, in order to avoid stability problems right from the beginning.

2.2.2 A Discrete-Time Dynamic Friction Model

Inspired by the LuGre friction model as introduced in Canudas De Wit et al. (1995) and further discussed in Olsson (1996) and Olsson et al. (1998), I worked on an easy-to-implement and discrete-time dynamic friction model, which I called Brush. From the LuGre model, my interpretation inherited the idea to simulate the deflection of virtual bristles moving over an abrasive surface. These bristles are flexible and will bend when there is a relative movement of the objects in contact. If the bristles deflect, they apply a restoring force that works against the cause. The internal state variable, z, denotes the current average bristles' deflection. Bristles may belong to the same brush, therefore the name.

Dynamic friction models usually only take the relative velocity as input. In order to calculate the deflection of the bristles, we need to integrate that velocity. So, the internal state forms a discrete integrator and we get a relative position, i.e., the bristles' average deflection. This integrator has a hard limiter, so the bristles can only be deflected to a certain degree which is denoted as the maximal bristle deflection σ. The discrete time update equations for the Brush model are given as

$$z(t) = f_{lim}\left(z(t-1) - v(t), \sigma\right) \tag{2.6}$$

$$\overline{v}(t) = k_B z(t) \tag{2.7}$$

with the input velocity v and the output velocity $\overline{v}$ and f_{lim} being the limiting function which constrains z to the interval $[-\sigma, \sigma]$. The bristles behave like non-linear springs; when deflected they work against the cause of their deflection. A factor k_B defines their stiffness similar to Hooke's law. The Brush model is by design not based on forces. It outputs a desired target velocity. So rather than directly applying restoring forces, we instead use a velocity controller

$$F_{ctrl}(t) = C\left(\overline{v}(t) - v(t), S(v)\right) \tag{2.8}$$

with $C : \mathbb{R}^2 \rightarrow \mathbb{R}$. The controller takes the difference in velocities (i.e., the error signal) as its first argument and tries to bring that difference down to zero using at most the force defined by the function $S(v)$ as the second argument (refer to Equation 2.9).

Using a controller is in general equivalent to applying the force directly but allows a smoother application over several time steps with certain compliance in order to meet the stability affordances of discrete time simulations. When exerting spring reaction forces directly to bodies with small masses, we easily encounter numerical stability problems in simulations with comparably large step sizes, e.g., $dt = 10^{-2}$. The velocity controller helps to stabilize since it separates the problem of applying the force from the friction model towards well-understood structures such as PID controllers. Also, parameters for the controller could be adjusted separately. Furthermore, many physics engines already provide stable velocity controllers, so does the Open Dynamics Engine. So in some cases, we could already benefit from that. However, in cases where no controller can be taken out-of-the-box, we could implement such a controller as described in subsection 1.2.1.

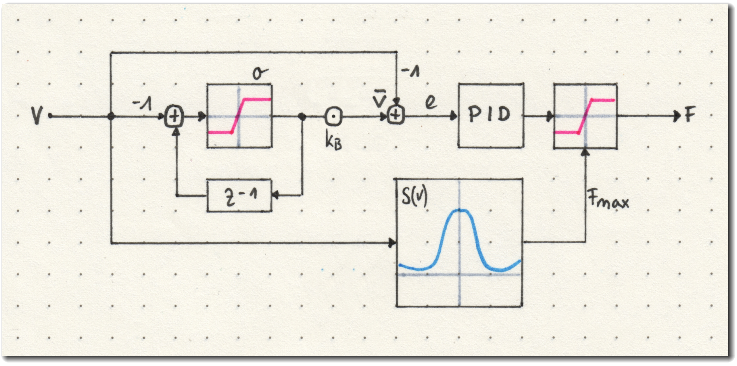


Figure 2.6: Schematics of the Brush friction model: The left part is the velocity integrator to create the bristle displacement. The right side comprises the velocity controller (in form of a PID) and the Stribeck-Generator to set the velocity dependent force limit for the controller.

Stribeck Generator

So far we saw that the deflection of the bristles invokes a restoring velocity. The maximal force which can be used by the controller to reach the set-point, i.e., the restoring velocity, is determined by the function

$$S(v) = F_c + (F_s - F_c)\, e^{-(v/v_s)^2} + F_v|v|^\delta \tag{2.9}$$

which has a similar shape as Stribeck's curve but is defined over the full velocity range. The only argument v is the relative velocity of the surfaces in contact. Stribeck's curve illustrates friction for constant, steady velocity. Independently of the parametrization, its characteristic shape is valid for a broad palette of materials. With respecting $F_s \geq F_c$, the Equation 2.9 can be read as: If the input velocity is in stiction range v_s, the *Stribeck velocity*, use stiction F_s, otherwise use Coulomb friction F_c. Nonetheless, the transition between sticking and sliding and vice versa is continuous. Independently from that, we always add a velocity dependent term, which mostly affects domains of higher velocity. The shape of the fluid friction term can be manipulated with the exponent δ to get an either linearly or non-linearly increasing friction term.

Parameters

Like for all friction models, the parameters needed to fit the hardware may be difficult to determine (for a complete list refer to Table 2.2). All parameters needed for the

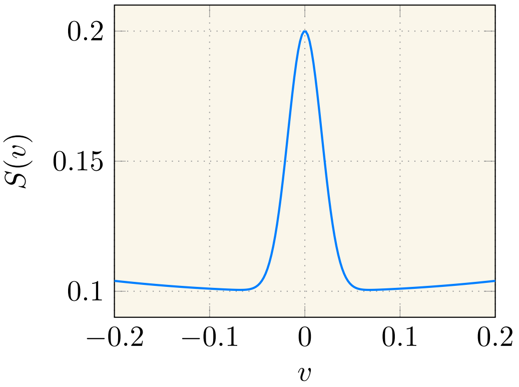


Figure 2.7: Characteristic shape of the Stribeck generator used for the Brush Friction Model. This function regulates the transition from stiction domain to Coulomb (dry) friction and fluid friction for higher velocities. In comparison to Stribeck's Curve it is symmetrical $S(v) = S(-v)$ and is defined over the full velocity range. Parameters for the figure are $F_c = 0.1$, $F_s = 0.2$, $v_s = 0.025$, $F_v = 0.1$, $\delta = 2$.

Stribeck function can be estimated by steady-state velocity experiments. However, carefully conducting these experiments may take time and need special equipment and some parameters as the maximal bristle deflection and the bristles' stiffness may be hard to measure at all. Alternatively, we could consider a machine learning approach: We collect enough representative data from hardware experiments and compare it to our simulation with a cost function. From experience, we bring the parameters in an already close starting distribution and define limits for each. After that a stochastic search, e.g., a genetic algorithm is applied in order to fit the parameters (see subsection 8.3.1).

σ	max. deflection	F_c	Coloumb friction coefficient
k_B	bristle stiffness	F_s	stiction coefficient
v_s	stiction range	F_v	fluid friction coefficient
δ	viscosity exponent		

Table 2.2: Overview of friction parameters for the Brush friction model including Stribeck's curve parameter. PID coefficients not included.

Experimental Results

To qualitatively demonstrate the capabilities of the model we need to have a proof mass which is assumed to be subject to friction and exert an external force. The results of the following experiment are depicted in Figure 2.8 and show the object's state, the external forces, the acting friction forces and the internal state of the bristles. The state of the object (proof mass) is (x, v) and it rests in the beginning. There is a continuously increasing external force exerted to the mass and, due to stiction, the object still rests, but the internal state z accumulates. Actually, the object moves a little (the pre-sliding displacement) but does not gain a higher velocity. After 500 ms the external force is removed and the object returns to its original position. Then at $t = 1s$ the force is increasing again and within the next second, the internal friction state (the bristle deflection) has reached its maximal value and cannot further *compensate* for the external force. So the proof mass slowly starts to move and since friction rapidly drops when velocity is increasing according to $S(v)$ the object begins to slip. During this transition, the friction applied drops quickly from the stiction to the Coulomb domain. Finally, when the external force is removed again the proof mass slows down accordingly, friction raises to the stiction level, the internal state z rewinds and eventually the object comes to rest. But although the external force does not vanish completely the objects still rests. The force is fully compensated by the bristles and the mass sticks.

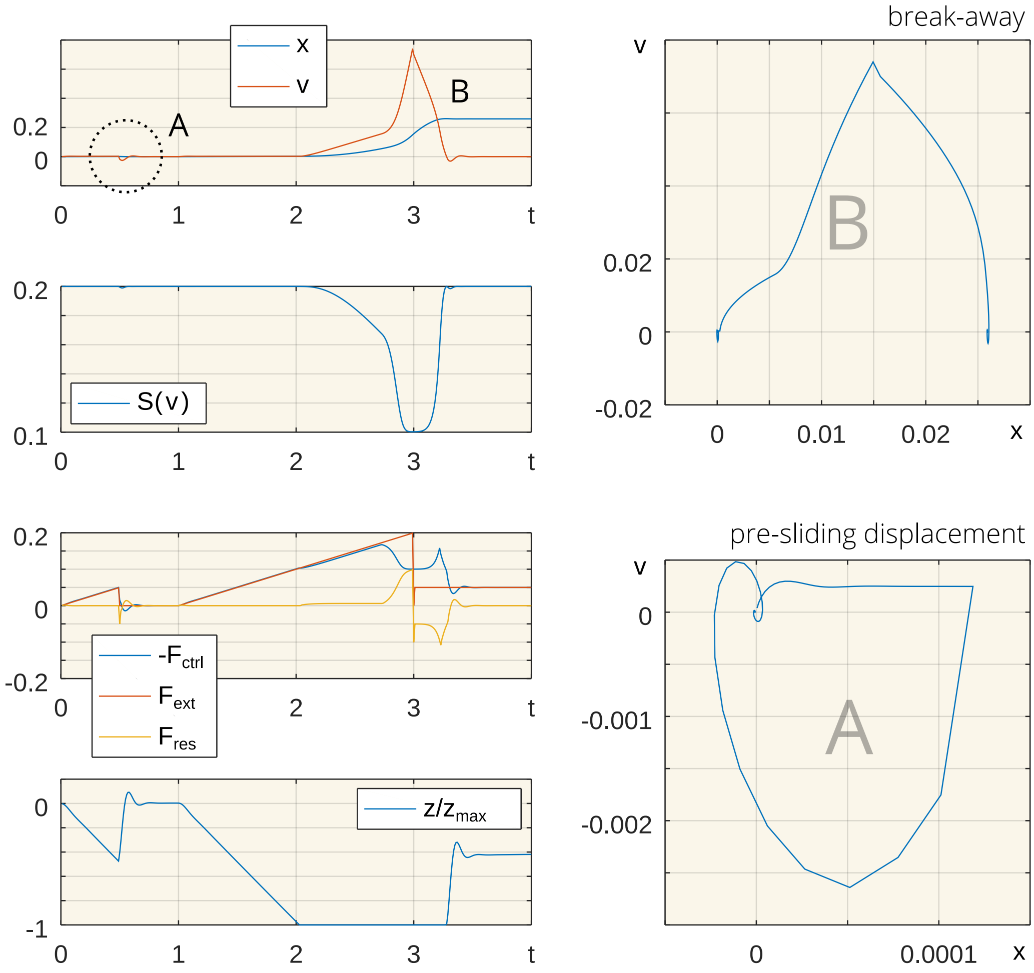


Figure 2.8: Data illustrating the effects of stiction generated by the Brush-Model: A lower force leads to pre-sliding displacement (A) and position x is restored after the force is removed again. A larger force leads to the break-away (B) resulting in a permanent displacement. Parameters of the model: $\sigma = 0.025$, $k_B = 1$, $F_c = 0.1$, $F_v = 0.05$, $F_s = 0.2$, $v_s = 0.025$, $\delta = 2$. Parameters are selected for demonstration purpose to better visualize the effects covered by the model. $F_{res} = F_{ext} - F_{ctrl}$ the resulting force that accelerates the proof mass.

Summary and Conclusion

To summarize the features of the here developed Brush friction model, it can be said, that with comparably little efforts in implementation we gain a convincing sticking behavior without the need to access the exact external forces from simulation and without the need to determine zero velocity. The model supports break-away force and non-leaky resting as well as the more subtle effects of pre-sliding displacement and returning to the original position if the virtual bristles did not fully reach their maximal deflection. As a bottom line, the friction force for steady velocities is identical as described by Stribeck (1902). We split up the model into two parts, the first is concerned with the friction and creates a target velocity and the second is a velocity controller part which takes this target velocity as set-point and tries to keep the difference between current velocity and target velocity as small as possible under the constraint of the maximal force given by a non-linear function of velocity. The controller can be developed with standard techniques such as PID. With this *trick*, we keep the simulation out of the loop of our friction model and it is now easier to implement a numerically stable system.

2.3 Simulated Robots

Within the scope of this work, I created a couple of simulated robots with different degrees of freedom. My goal was to have a variety of platforms for testing gait controllers and learning algorithms. The generalized controller defined in subsection 1.4.1 is independent of the number or orientation of the robot's joints, the lengths of the limbs, the mass distribution and so forth. My hypothesis was, that I will probably find suitable learning methods when I start testing algorithms on multiple morphologies right from the beginning. I wanted to avoid the unintended introduction of dependencies to specific construction details. The only assumption I made was that the created robots rely on actuated revolute joints and that their segments are made of rigid bodies only. And even though the algorithms may work fine with other actuation types or with soft body dynamics they were not explicitly tested within this thesis, and I will make no statement on this.

During the development of the robots, some of them were created with initial flaws, such as improper joint configurations that limited their ability to move. Systematic studies for optimizing robot morphology are not considered. All robots described here, were developed through a brief series of trials and errors to remove the most crucial design mistakes.

The following series of robots, different as they are, provided a good basis for successively increasing complexity. From one robot to the next, the number of joints increased. This is worth mentioning since most methods directly rely on the number of sensor inputs. Also, robots are able to move more versatile with increased degrees of freedom. The simplest systems are either fixed or can only crawl on the ground, so stability is not an issue. With increasing complexity, the robots managed to actually walk. Finally, the humanoid robot is expected to walk on only two legs, posing the problem of keeping balance during walking.

We begin with a simple pendulum describing a system for testing that intentionally can move but not walk to get a basic feeling for the interdependencies between the algorithmic components. We continue introducing robots with only a few joints but which are already able to crawl on the ground. The last two describe more elaborate robots with up to 22 joints that are able to walk and run with versatile gaits.

Pendulum

My simplest system under test is a physical pendulum as depicted in Figure 2.9. This system has only one degree of freedom is suitable for testing methods on a small scale and can be regarded as a single-joint robot. Indeed, for such a simple system, there are already rich dynamics and a lot of behaviors can be differentiated. I found using systems like this to start with crucial for studying the interdependencies of the algorithmic components and furthermore to manually adjust the free parameters which are needed for successfully applying optimizers.

Since for experiments with legged robots, behaviors such as walking in different directions and gaits or standing up and recovering balance are of major interest. For a simple pendulum, an equivalent set of expected behaviors must be defined to see, if, in principle, the algorithms work and produce the anticipated behaviors. Since a pendulum has only a single joint and actions are limited, I defined three different tasks for the experiments: *1)* swing-up with restricted torque, *2)* return to resting position, and *3)* intrinsically motivated learning. I regard them as the pendulum's equivalent to the walking tasks for the walking and crawling robots.

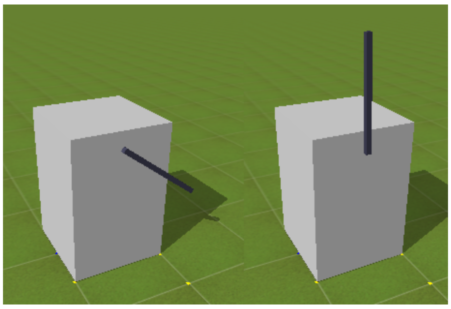


Figure 2.9: The physical pendulum.

The vector (φ, ω) defines the system's state with deflection angle φ and angular velocity $\omega = \dot{\varphi}$. The angle is not restricted by limits and so the lever can rotate freely, but the velocity is limited by friction and the maximal torque applied. The state changes according to the torque M applied to the system which is the sum of gravity acting on the lever, joint friction (according to subsection 2.2.2) and the torque exerted by the motor according to the control input u. The lever's mass is $m = 0.47\,kg$ and its dimensions are $l = 0.5\,m$, $b = 0.025$, and $d = 0.225m$ (distance between CoM and pivot). The pendulum's damped Eigenfrequency is $w_d =\approx 1.5\,Hz$.

Crawler

The *Crawler* is first of my systems actually worth calling it a robot. It is a cube-shaped torso equipped with a 2-DoF arm, with the help of which the robot can only *crawl*. Since the torso and the arm are made of boxes only, the movement abilities of that robot are strictly limited to crawling forward and backward. However, with applying large torques to the joints it is nonetheless possible that this robot performs small jumps. Furthermore, it can happen, that this robot performs actions that result in finally laying on the back or side. Also, it may happen that Crawler gets stuck with its arm clamped under the torso and is not able to move anymore. (In the conclusion of this chapter, I discuss how to handle or avoid such terminal states.) This robot actually slides over the ground with its body, so the friction for the ground-touching part of the arm must be higher than the friction for the torso to slide easily.

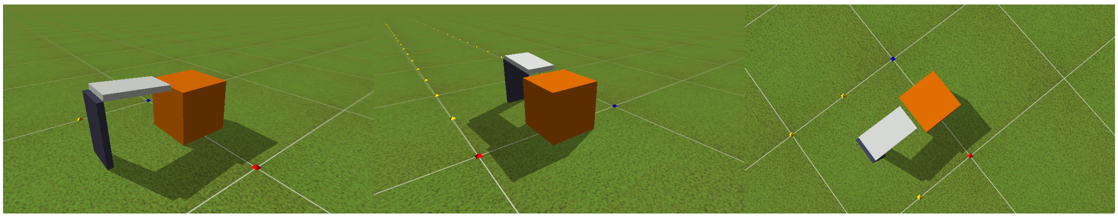


Figure 2.10: The Crawler robot from different perspectives.

Tadpole

This robot is called *Tadpole* and even though it is still kept simple with only four joints, it is already more robot-like. The shapes are rounded which gently introduces the problem of balance recovery. It has two arms with two joints each. The robot is in principle able to move everywhere if the ground is flat and if there are no obstacles. It can move forward, backward, and can turn. Furthermore, combinations of such behaviors such as moving in circles will be possible. Therefore we can expect from this robot to move forward, moving backward, turn left and turn right. Also, the Tadpole may accidentally fall on its back and will probably take a while to roll back to the initial position on its own. Likewise the Crawler, Tadpole also has increased friction on the arms and reduced friction on its torso to facilitate sliding over the ground.

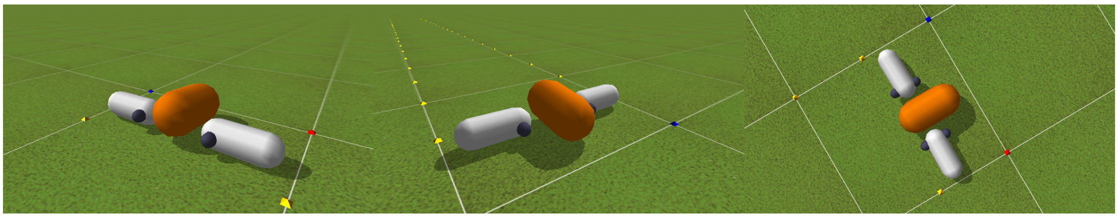


Figure 2.11: The Tadpole robot from different perspectives.

Quadruped

With its 12 degrees of freedom, the Quadruped robot is already complicated enough to demonstrate the capabilities of the methods but is not too complex for good access to the analysis of the results. Also, it is the simplest of the mentioned robots that actually walks on legs. This robot can move around freely with different gaits and speeds, and it is able to turn in place and move sidewards. It can run comparatively fast and make large steps with all four legs lifted from the ground simultaneously. All twelve joints of the Quadruped are arranged in the simple perpendicular scheme in order to keep the results comparable. As the Crawler and the Tadpole, the Quadruped robot can also fall on its back but is able to recover and turn back to the initial position. It can lay down, stand-up and roll over. Due to its good tradeoff between versatility and simplicity, it plays a central role in this thesis since most of the evolution experiments were conducted with the «four-legged» robot.

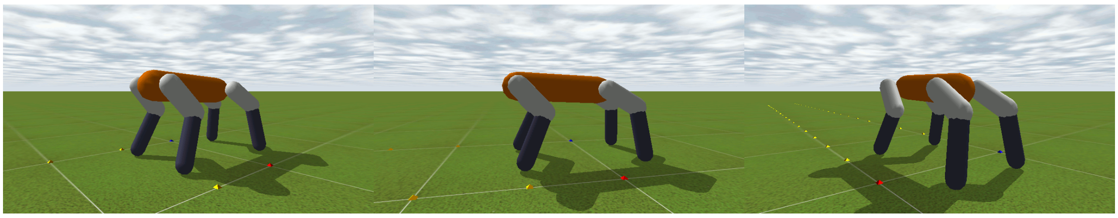


Figure 2.12: The Quadruped robot from different perspectives.

Humanoid

The Humanoid is the most sophisticated robot within these experiments. It has 24 joints and similar-to-human, child-like proportions. Since the joint count is significantly greater than with the aforementioned robots, it makes the control problem much harder and it is also an interesting case to see if the methods scale in a behaved fashion. Additionally, the problem of keeping or recovering balance will be way more difficult to solve, since the stability margin is reduced to only two feet and the relative height of the center of mass has increased. Thanks to the many degrees of freedom, this robot is basically able to show the most versatile movements, but finding them will also be a harder task for learning algorithms. The number of free parameters for the joint controller is already very high, that manually tuning good parameters for walking is hardly possible.

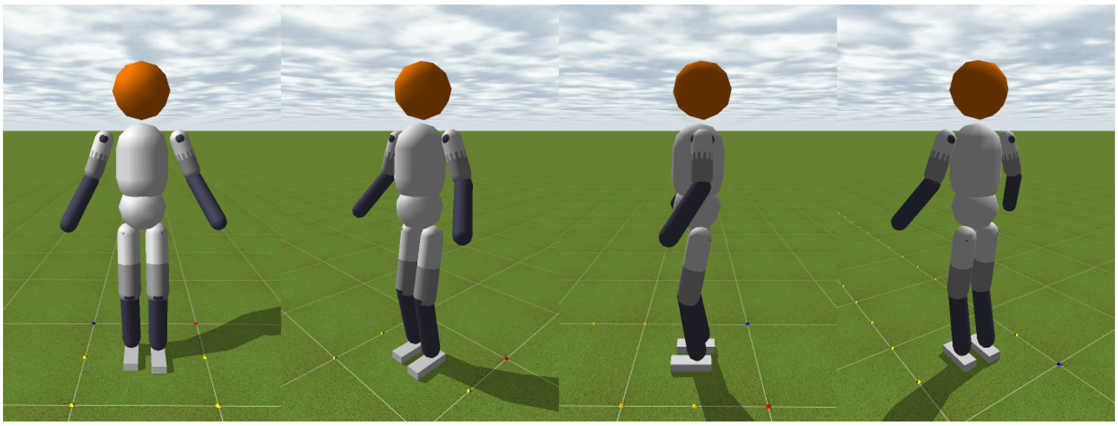


Figure 2.13: The Humanoid robot from different perspectives.

Overview

The robots all have a comparable size within the same magnitude. This constraint comes from having a single type of simulated servo motor which is identical for all robots. Each robot has exactly one motor per revolute joint. However, the robots are restricted in how much torque their motors can provide: For instance, the Tadpole is restricted to use 1/5 of the maximal available torque and the Humanoid has doubled the torque. Also, if the size of the robots is kept moderate, it is also easier to relate the results with real robots.

Name	DoF	Size	Weight	Height CoM	Torque
Pendulum	1	L:0.50 m	0.47 kg		2/5
Crawler	2	L:0.15 m	2.15 kg		3/5
Tadpole	4	L:0.20 m	1.77 kg		1/5
Quadruped	12	L:0.40 m	4.72 kg	21 cm	1
Humanoid	24	H:1.05 m	9.49 kg	60 cm	2

Table 2.3: Overview of the used robot platforms and their properties.

2.3.1 Improving the Walking Robustness

Flat ground is the simplest world for walking robots. There is nothing in the way, the robot could stumble over. Provided enough time, it could just move infinitely far in the same direction without ever reaching an end. So when the task of walking on even ground is solved with moderate speed and low chance of falling, we can subsequently raise the level of difficulty and provide a setting in which the robot can move under more complicated conditions.

For increased robustness of the learned gaits, we can add external disturbances. For instance, we could push or pull on the legs or torso while the robot is walking. We can place some obstacles in front of it, so it must successfully step over them. We could introduce slopes. Whatever robot we train, there must be sufficient «noise» in the setting, if we want the controller to be robust. Otherwise, evolution or learning algorithms will always adapt to the clean, flat and even ground conditions and the acquired behaviors will stay on the edge of stability. Why?

Adaptation towards critical stability must inevitably happen if we optimize for speed or efficiency only. Here is why: Stability has costs. It is costly to search for more robust parameters. The optimizer will find better solutions sooner if the robot does not stumble and fall often since the fitness function delivers similar values. While optimizing for speed, faster walking has an inherently increased risk of falling. But also if we optimize for efficiency. The essence of efficient behavior is a reduction to the minimum effort needed. Therefore, when trying to minimize the control output to save—let us say electrical power as a real-world equivalent—it will have more costs to react to the disturbance and recover balance. However, a simple controller like ours will probably be incapable of strictly minimizing control costs and be rock solid simultaneously. A trade-off must be found between stability on the one hand and efficiency or speed, respectively, on the other.

Conclusions and Discussion

Possible extensions to the robot models range from flexible body parts to ground contact sensors. But it appears highly interesting to me to find out the effects on learning when using (simulated) strain gauges, i.e., sensors for indicating the strain of a material or structure at the point of attachment. Either directly used as force feedback or they could be incorporated into objective functions in order to reduce mechanical stress. In

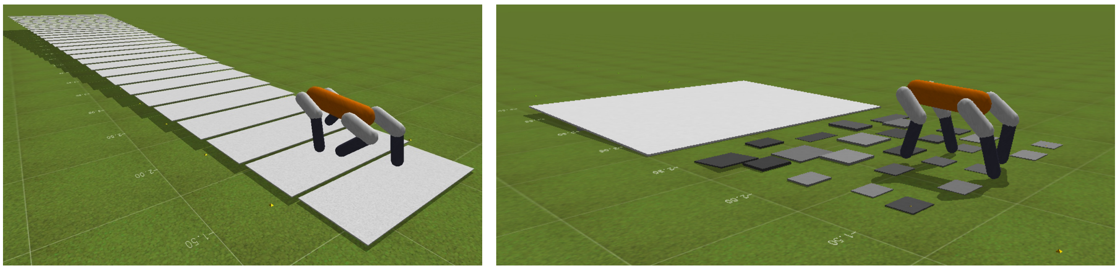


Figure 2.14: Increasing walking robustness through settings with disturbance: The robot walking up a ramp or over tiles and a seesaw.

the current simulation, the objects are not deformable, but at the joint connections, the solver must apply forces to meet the constraints given by the model. It must keep the body parts together when they are subjected to load and apply joint correction forces. So even in a rigid-body dynamic simulation like ODE used by Simloid, we potentially have the chance to get similar sensory information from the joint connection.

Just to give a brief note on «How to deal with undesired terminal states?» Each hardware design of mobile robots faces the problem of (undesired) terminal states. These are states of either body or environment in which the robot may get stuck and is henceforth not able to recover itself regardless which action it takes and needs external help. Terminal states can be partially avoided by hardware modifications or environmental changes but it is hard to avoid all terminal states or even foresee them. But the performance of learning algorithms can easily suffer from such situations if not handled at all. Fortunately, with an increased number of joints the risk of getting stuck often declines. But for simple robots used in unsupervised learning scenarios, terminal states can be a severe problem since it implies that we are able to detect them, and provide a mechanism to reset the robot to a safe position. However, it could be fun to think about not avoiding terminal states at all. Why not including them into learning in order to avoid them in the long run. We could allow the robot to send a *cry-for-help* signal which will call some external person to help the robot to reset it, as sort of an action which can be taken actively by the robot.

To close this chapter, allow me to give some historical remarks: The Simloid project was initially started by Daniel Hein for his diploma thesis in 2007. I used it as a student in my robotics courses. Since then, large parts were rewritten and extended by myself to match the needs of my thesis. Many new features were included, but also some were removed. My decision, not to use a highly developed simulator off-the-shelf, but instead take this project and develop it further to meet my requirements, was based on a simple argument: I wanted to be able to give it the most suitable form for my experiments. And honestly, I did not want to waste any time in struggling with the usual pitfalls of further developed projects. Over the years, simulation environments like Gazebo became quite popular, and by today I would probably think differently and it might be a good choice for similar ventures to start from.

3 Evolving Dynamic Robot Locomotion

Evolution has found countless ways for living things to move in their specific environment. Therefore, it is not surprising that the principle of evolution has inspired numerous researchers, such as Rechenberg (1994), to abstract the underlying mechanisms of evolution and make them applicable to optimization problems of almost any kind. Because of their generality, we can apply *Artificial Evolution* to very different problems. Often, we then see what we can learn from it and *how* it finds a particular solution. Afterwards, we can then adjust the optimization process itself, if necessary, to find solutions faster or more reliably with problem specific machine learning methods.

For quite some time, evolutionary algorithms have therefore been a common and valid approach to solve many different optimization problems in robotics research. For example, questions of robot morphology or control have been studied by Auerbach and Bongard (2010, 2009); Nolfi and Floreano (2000) and Komosinski (2000), among others. The method has proven particularly useful for research on controllers for locomotion of walking robots, as done by Bongard (2015, 2011); Rempis and Pasemann (2012) and Hein et al. (2007). Of particular note is the inspiring and groundbreaking work of Sims (1994), which deeply impressed me when I first saw it in a lecture at the university—so much that it gave me lasting motivation for my own research efforts in robotics.

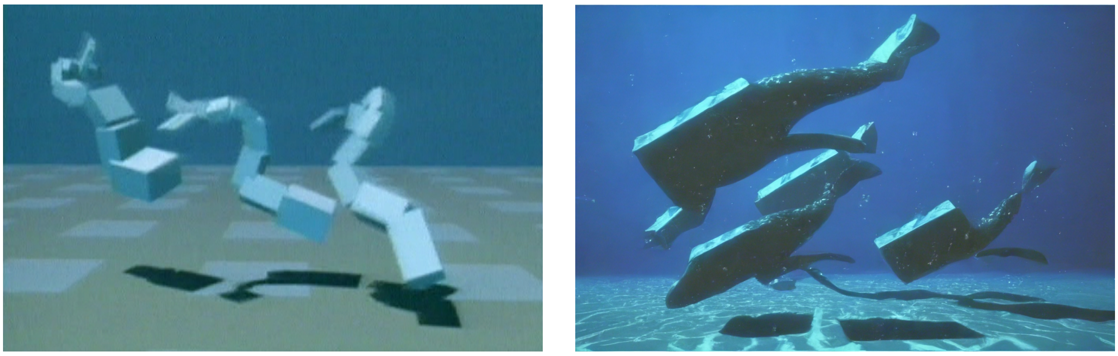


Figure 3.1: Scene from Karl Sims' *Evolved Virtual Creatures*, 1994 (left) and reinterpreted by Midjourney (right).

This chapter is dedicated to finding basic locomotion modes for the robots presented in the previous chapter. It aims to prove that different gaits exist for the given morphologies and controller structures, which together form a set of basic motions that can be worked with in the later chapters. The potential of the generalized controller of subsection 1.4.1 will be systematically explored, also considering different walking directions and ground conditions, in order to create the broadest possible picture of existing solutions.

The results of this chapter will be useful for inspiring learning algorithms for robot walking and provide insight into how similar gaits might be found through pure online learning. In doing so, the found set of motions might also be useful as an easier starting point for further learning without having to learn everything from scratch. The objective functions used in the evolutionary method will provide ideas for designing reward functions in reinforcement learning. One of the most important issues is to find out how to restrict the search space in order to reduce the search time.

This chapter is structured as follows: First, I describe the basics of the evolutionary algorithm used and define important terms. I also discuss the improvement of existing strategies in terms of parallelization. I continue with how this principle is applied to robot walking and gait control by describing my methods, in particular the fitness functions and necessary constraints. I then describe and discuss the results and give an overview of the behavioral space of the simulated robots that I have been able to explore so far using evolution.

3.1 Evolutionary Algorithms—Terms and Fundamentals

Evolutionary algorithms comprise a whole family of methods for solving optimization problems. It is a general method, which is not coupled to a certain type of problem. A given optimization task is formulated abstractly as a set of parameters, the genotype of a so-called *individual*. All individuals of one generation form the current *population*.

Each individual in the population is evaluated so that its phenotype is given the opportunity to demonstrate its capabilities in a defined experimental set-up. The optimization is guided by the return value of an objective function, called *fitness*, which assigns a scalar value to each individual, i.e., ultimately to each parameter set. All individuals are ranked according to their fitness values, and those with higher fitness are more likely to be selected and thus have a higher chance of evolving. Each evolutionary method repeats the same scheme: (1) reproduction with diversity, (2) evaluation of individuals by a fitness function, and (3) selection of the best individuals according to an appropriate strategy.

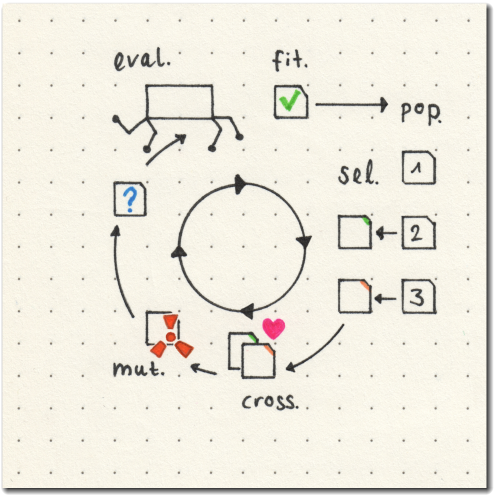


Figure 3.2: Illustration of the evolution cycle.

Optimization is driven by random injections of new parameters, referred to as *mutations*. This randomness expands the diversity of the parameter base and possibly produces new features that must be evaluated. The degree of randomness determines how quickly adaptation can occur, but also indicates how stable the population is. *Recombina-*

tion distributes pre-existing genetic information within the population by taking features from different individuals and creating new individuals with mixed features from them. Recombination thus causes the rearrangement of the genetic base and the spread of features within the population. After all individuals have been evaluated and ranked, *selection* is performed, which serves to contract the genetic base again. This keeps the population size constant and thus the effort of evaluation, and compressing the genetic base to the essential features that lead to increasing fitness. Recombination, mutation, evaluation, and selection is thus an ongoing process of expansion and contraction of the solution set, which moves targeted towards higher fitness.

Applicability of Evolutionary Algorithms for Robot Walking

Why is artificial evolution suitable for finding controls for walking robots? In general, for any sufficiently complex walking robot, it is generally assumed that there are several possible gaits that can be considered optimal in terms of direction, efficiency, or speed. In order to find even some of them, we also need a search strategy in addition to our parameterized controller. The parameter space of the chosen controller can grow quite large, so it is impractical and time-consuming to manually search for suitable controller parameters. Moreover, since the parameters inevitably depend on the morphology of the robot, we may have to start optimizing the parameters again after changing it. Thus, we inevitably need an automated method to find and refine these parameters.

Since pure random search is problem invariant, in principle any optimization problem can be solved by evolution for which we can define a function that assigns a unique fitness value to each set of parameters. In practice, however, we often deal with problems for which we can only specify stochastic fitness functions, so there is not necessarily a unique fitness value for each individual, but rather a particular distribution. Therefore, in order to approximate a suitable average fitness value, we need to evaluate the same individual multiple times in such a case.

The fitness function defines in an abstract way the target behavior. For example, if we want to find a control for walking forward as fast as possible, then the fitness for two evaluated individuals must reflect the relative differences in their performance in walking forward. We place the robot in a particular starting position and compare the behavior of the individuals. If all other characteristics are equal, the individual that walked farther must be rewarded with a higher fitness value. This can be easily achieved since we can simply measure the distance traveled by the robot since the beginning of the experiment.

Thus, to apply artificial evolution, we need a problem definable as free parameters, an evaluation procedure with a fitness function, and evolutionary operators. The search for gaits of walking robots, more precisely the search for parameters of the generalized controller structure, fulfills all these requirements to apply the evolutionary method, as I will demonstrate in the following sections.

Individuals, Population, and Generations

Free parameters are part of the individual. This will usually be a vector or matrix of real numbers, but in general it can be any kind of symbolic representation such as binary digits, characters or words. The total number of free parameters is usually referred to as the *size of the individual* and should be kept as small as possible, as it also affects the structure of the search space and the duration of the search.

When an individual's genotype is evaluated, the phenotype, i.e., the sum of the observable characteristics of the individual, is generated. The genotype is kept constant during the evaluation. However, the phenotype usually changes in response to the environment. For the task of developing control parameters for walking robots, the phenotype is the movement of the robot in relation to its own state and that of the environment, which may differ, for example, between flat and rocky ground. In the following, the term individual thus describes the robot system with a particular set of parameters, its performance measures in relation to a particular task, and other characteristics.

The genome must meet some requirements: (1) Each of its parameters should (ideally) represent only a single feature. Any dependencies between different parameters that are known should be resolved beforehand, e.g., symmetries. (2) For the genome, it must be possible to define mutation and recombination operators that modify it in a meaningful way. If the genome is altered by such an operation, this must also be reflected in a change in its performance. We should avoid indifferent features as much as possible, e.g., the color of the robot, and include only those parameters in the genome that affect the fitness of the individual. Sometimes, however, certain mutations may have no effect in one environment but cause significant change in others.

The size of a population is given by $P \in \mathbb{N}^+$ and each population is assigned a generation index, e.g., P_5, the population of the 5^{th} generation. Such a *generation* is a representation of the population at a particular time during the optimization. While the starting population P_0 is usually initialized randomly or with the help of already pre-evolved individuals (so-called *seeds*), subsequent generations usually build on the previous ones only by applying evolutionary operators. In the evaluation, ideally all individuals of the same generation should also be treated in the same way to ensure the comparability of their evaluation results. This is especially important if randomness is present in the evaluation or if the evaluation function is to change over the course of generations.

Fitness

The fitness function is the most important part of an evolutionary algorithm. It determines the *performance value* of each individual as a result of its evaluation. All individuals of each generation are directly compared with each other based on their fitness values. Therefore, to be fair, the fitness function must remain unchanged, at least for individuals of the current generation. However, it is quite common to change fitness functions during evolution in order to shape the optimization process across generations and thus avoid undesired local optima.

In favor of an optimal evolution run, it is common to track the minimum, average, and maximum fitness values of the individuals of each generation as a compact set of statistical characteristics. If all hyperparameters are appropriately well set, the average fitness values will hopefully be halfway between the minimum and maximum values. In other words, there should always be some diversity in the results. If there are too many individuals with poor performance, the mutation rate may be too high, and the optimization process is unlikely to converge. On the other hand, if there are too many results with fitness values close to the current maximum values, then too few new features are available, and evolution converges only slowly or possibly only to a local optimum.

We speak of *shaping* when we change the fitness function in successive steps during optimization to achieve a certain goal. In principle, the fitness function can be changed after each completed generation. For some tasks it is too complex or very costly to find a solution with only one fitness function. Therefore, fitness functions are changed either manually or automatically over the generations. Thus, an easier task is often chosen at the beginning, which offers a larger number of valid solutions, and then the difficulty level is increased step by step.

In complex problems, it can easily happen that the optimization process gets stuck in fitness niches, i.e., , local optima, which is usually considered as an inadequacy of the fitness function. Here is an example: In our robot, if we reward only the distance traveled, individuals tend to hop a little and then immediately fall forward to quickly make distance and achieve large fitness values. It is clear that this strategy is not beneficial for evolving forward walking, so we need to add more constraints to avoid such local optima. More on this in section 3.4.

Evolutionary Operators: Mutation, Recombination, and Selection

Mutation means to change the genome randomly. Mutation is defined as an operator which is applied to the genome of any individual and which moves the population away from the current actual state. Often, the mutation operation involves changing each parameter by drawing a new value from a normal distribution, where the mean is the previous value and the standard deviation is called the *mutation rate.* A very common approach is also to select a particular parameter and completely re-randomize it from a uniform distribution spanning a predefined allowable interval.

From a scalar fitness value, there is no immediate information to draw that tells us in which direction to change the genome. Some strategies from Rechenberg (1994) use the development of the genome over time to further constrain the random distribution of our mutation operator and roughly maintain the current direction or avoid unnecessary backwards steps. The sense is to speed up the optimization and thus corresponds to the idea of the momentum term in stochastic gradient descent.

The mutation rate is usually considered as a fixed hyperparameter, but there are methods for *coevolution* of this parameter. In other words, the rate of change is itself considered a free parameter and is part of the individual. This can help to leave local optima or plateaus of the fitness function.

The task of the *recombination* or *crossover* operator is to generate new individuals from the already existing genetic base. As mutation introduces new features, if they are useful they are distributed throughout the population by crossover. In general, recombination does not add new information to the genotype, but the phenotype might show completely new behaviors. Crossover is defined as a function of at least two input genomes, referred to as *parents*, yielding a *child* genome that is considered part of the next generation. Depending on the specific evolutionary strategy, the parent genomes are either discarded or carried over into the next generation.

A simple type of crossover is to iterate over all parameters of the parent genome and select randomly with a uniform distribution. However, if more knowledge about the dependencies of the parameters can be used or the genome is already structured in separable logical units, a crossover operator is often defined much more specifically and cuts out more or less large parts of the genome, e.g., functional building blocks or submodules.

The task of *selection* is the subsequent reduction of the feature space to the best evaluated according to fitness. Only when each individual of the population has been evaluated and has received its fitness value, we can compare them. In doing so, we assume that the evaluation function is fair, i.e., constant for each individual in the current generation. However, this is often only an idealization, since the evaluation can be stochastic. All individuals are then sorted by their fitness value, keeping those with higher fitness and discarding those with lower.

The simplest implementation of selection is to keep only a subset of the best individuals, e.g., the top 20%, and discard the rest. In this case, the *selection size*, $S \in \mathbb{N}^+$ with $S < P$ is another hyperparameter. Selected individuals are then mutated again and recombined to fill the population up its original size to form the next generation.

General Remarks

As in all optimization processes, a good choice of initialization and hyperparameters is crucial for success. For Evolutionary Algorithms with a large search space, one difficulty is to find suitable initial conditions or seeds. However, with the given controller structure for walking, it was shown that several known, albeit still non-optimal parameterizations can already be initialized by hand that have already shown useful behaviors, such as a simple PD-controlled standing robot. Even though these behaviors are far too simple, because hand-made, to be considered optimal in any respect, they are still essential as initializations for the start population.

Evolutionary algorithms differ in their operators, their parameterization, or their representation of the genotype. For example, we might always leave the best individual unchanged and not retested into the next generation. This strategy might work for constant fitness functions but likely fails for stochastic ones. So, depending on how well we know the problem, we can optimize our operators, choose a more appropriate representation, or automatically change our hyperparameters according to specific measurements. The basic principle always remains the same, with most approaches based on the concept of generations. Alternatively, I present in the next section a simple algorithmic modification that does not iterate over generations but uses a continuous population of constant size.

3.2 Proposal for Generation-free Evolutionary Algorithms

In general, evolutionary algorithms are based on the concept of generations. For each individual in the population, an evaluation is performed to obtain a fitness value. Then all individuals are ranked by their fitness values and a new population for the next generation is created using the operators selection, recombination and mutation. The population of the previous generation is mostly discarded.

But distributed computing is standard today. For time-consuming evaluation functions, it is therefore a good idea to use multiple processes or even cores on distributed machines. The application of the above-mentioned generation-based evolutionary method has some disadvantages for this kind of parallelization: Assuming we distribute the evaluation of single individuals to different computers, we have to wait until all individuals are evaluated before we can create a new generation. However, if the respective evaluations differ greatly in their computation time, then idle times grow and will inevitably slow down the overall process unnecessarily. For example, when learning to walk for simulated robots, tumbling happens, so an evaluation could be terminated early to save computation time. Idle times would be the consequence. The most serious disadvantage, however, is that newly created individuals cannot benefit from any recently found features until all individuals of the current generation have been evaluated. It will therefore take longer for good features to spread.

In general, parallelizable evolutionary algorithms are now of greater interest, as studied by Bandow and Hartke (2006); Krzywicki et al. (2014), or Krzywicki et al. (2015). They are usually referred to as *pool-based* or *evolutionary multi-agent systems.* While Bandow and Hartke adapted the algorithm specifically to their problem, Krzywicki et al. looked for a general solution and developed a massively competitive approach where agents compete with each other and the Erlang programming language is used due to its inherent support for parallel computation and communication. However, with a few minor adjustments to the general method above, we can reuse it to create a *generation-free* variant with very similar properties.

Method Description

Suppose there is already a fully evaluated population of P individuals. We randomly select two individuals, recombine their features to form a new offspring, apply mutation, and evaluate as usual. Since the offspring now also has a fitness value, we can determine its rank relative to the whole current population. For example, let's say the offspring has the j^{th} rank. This offspring immediately replaces the existing individual with rank j. All other individuals remain unchanged, only one individual is replaced per step. Since each evaluation is independent of the others, we can parallelize the process of evaluation onto multiple machines with a single host process managing the population and replacement request from the worker processes that perform the computationally intense evaluation work.

It is important to note that only exactly the individual with rank j is exchanged. A plain sorting and downward shifting of the worse ranks does not work and would have the consequence that the diversity of the population would be lost after a short time. As a byproduct from the fitness of the population we can see how well the mutation rate is adjusted. If only the best ranks are constantly replaced, too little mutation takes place. And the mutation rate is too high, if the offsprings have to be constantly discarded because their rank is so low that they cannot be returned to the population.

To rank each offspring in the population, we must first initialize the process by ensuring all individuals have been evaluated at least once. After this, we can proceed with the standard selection procedure, giving preference to those with higher ranks.

When using stochastic fitness functions, it is necessary to evaluate individuals more than once and thus improve their fitness estimates. Otherwise, individuals of better rank may be replaced too infrequently if they have erroneously overestimated fitness values. Thus, instead of only performing replacements where new offspring are produced, we also need to perform regular updates where we re-evaluate individuals and average their fitness values over the number of trials to obtain better estimates of their true fitness. I refer to the relationship between reevaluation and replacement as the *movement rate* of the population. The term was chosen with the idea that individuals in a population would move together to optima in the fitness landscape. When local optima prove unreliable though, continuous reevaluation helps to skip them in favor of stable optima into which the entire population can then move safely.

Summary of the Generation-free Method (Pool Strategy)

The proposed algorithm is as follows:

0. Given a population size $P \in \mathbb{N}^+$ and movement rate $\mu \in [0,1]$. Initialize a randomized or seeded starting population P_0 according to a given distribution. Evaluate each individual at least once and determine its ranking by sorting the population according to its fitness.

1. Repeat until a termination criterion is met:
   a) With probability $(1-\mu)$ select a randomly chosen individual for re-evaluation of fitness and respective rank (optional[1]).
   b) Choose at least two[2] individuals from the population with respect to a distribution that favors better-ranked individuals and generate an offspring from them using recombination and mutation. Evaluate the offspring and determine its fitness and respective rank j. If the rank is $j \leq P$, replace the j^{th} individual from the population with this new offspring. Discard it otherwise.

The proposed method is as straightforward to implement as the standard method, with a bit more computation required to keep the population sorted by rank after each fitness update or replacement. Therefore, this method is particularly suitable for optimization problems where the evaluation of an individual is computationally more intense compared to sorting a list of P real numbers—especially when the reduction of waiting times through parallelization and the immediate distribution of new features within the population have already resulted in larger time savings. For this project, that was definitely the case.

Closing Remark

In describing this method, I neither claim that the algorithm is superior to others nor will I recommend it as a general solution. Since I use this algorithm for my experiments,

[1] For non-stochastic evaluation $\mu = 1$.

[2] In principle, recombinations can also be generated from more than two individuals

I feel scientifically obligated to document my method, even though there has not yet been a systematic investigation with research questions other than my personal ones. Depending on the parameterization and underlying problem, both generation-based and generation-free methods can be well adapted to find walking controllers for legged robots, and that is what I have done. Although I have to examine a systematic comparison of their advantages and disadvantages, I generally prefer the generation-free method and intuitively found it more compelling overall for the experiments described here. The potential time savings in parallelization is probably the most decisive argument for this method. However, an exact numerical comparison with several different problems and parameters is not part of this work, so I can only give anecdotal evidence here.

3.3 Gait Controller as Subject to Evolution

Having described the general principles, I will now go into detail on how the building blocks of Evolution are applied to our problem. The general controller structure described in subsection 1.4.1 has numerous free parameters, which will be called *weights* in the following. The term was chosen by analogy with neural networks, since the structure can be considered as one. All controller weights taken together form the individual which is subjected to optimization.

Initial Values for Starting Populations

The set of all possible weight vectors forms the potential search space of our application. It is a vast space, and I found it challenging to explore. Depending on the problem, we can imagine the landscape of the fitness functions as a more or less rugged terrain with many local optima. Therefore, shaping the trajectory through this search space is crucial to speed up the optimization process and find the desired optima. Instead of the global optimum, I found it acceptable to settle for sufficiently good local optima coming close enough to my «experimenter's expectations» of what makes a good result. The search trajectory is adjustable by applying constraints and a reasonable initialization.

With a clever initialization, we can guide the process to favored results from the beginning, since experience shows that a purely random initialization often does not achieve good results or the search takes a lot more time. As starting parameters for each evolution run I therefore exclusively begin with slightly randomized already working controllers and refrain from purely random initialization. This means I initialize with hand-crafted controllers that use the weights of the inherent PD and CSL properties of the structure.

3.3.1 Symmetry Assumption

In order to obtain usable results in a reasonable time, we are not limited to setting appropriate initial values for the weights. We can also try to significantly compress the search space by reducing the number of free parameters in the controller structure. To achieve this, we look for possible dependencies between weights and make restrictions. In the best case, such a constraint does not prevent the optimization from finding solutions we are actually looking for, but speeds up the search. In other words, we use our specialized domain knowledge here to carefully constrain the free parameters. Of course, this also means that we introduce more assumptions and possibly lose some interesting solutions in favor of shorter search times.

When looking at most animals, we immediately notice their symmetry. Both their morphology and their locomotion in the main direction of movement, which we usually define as *forward*, is symmetrical in most cases. However, there are exceptions[3], such as crabs, which can move remarkably fast sideways. All robots created here also exhibit this symmetry. The torso and limbs are shaped and arranged symmetrically about the sagittal plane. Therefore, we can assume that the gaits we are looking for may also exhibit some degree of symmetry.

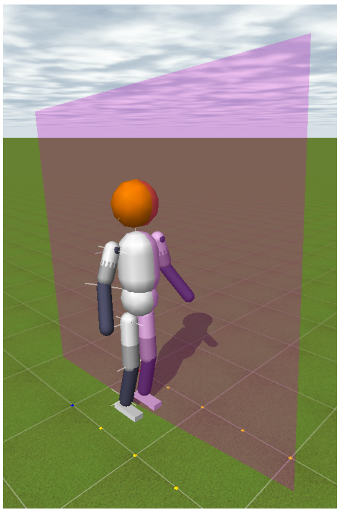


Figure 3.3: The sagittal plane of the humanoid robot.

Unlike shape, it is not immediately clear how we can define what makes a gait symmetric. As a first draft, we could base our definition on the design of the robot. We could require that joints always have a symmetric counterpart on the other side of the body. We could further specify that each symmetric counterpart has the same motion properties terms of torque, velocity, and range. Thus, these joints are identical in all their features except that their position and orientation are symmetrical about the sagittal plane. We call them *symmetrical joint pairs*. However, this does not apply to joints that originate exactly *in* the sagittal plane, such as a neck joint. Such joints do not require counterparts, but should have their axes aligned so that they are either within the sagittal plane or orthogonal to it.

Let us recall that the output of the generalized controller is a deterministic vector function of the current sensory inputs and past motor control values. The controller has no explicit memory (time invariance) and has a fairly large linear operating range due to the transfer function tangent hyperbolicus. Therefore, periodically changing sensory values must result in periodically changing motor control outputs. This does not change even if some of the weights have identical values. So, if pairs of symmetric joints have a common set of weights, we still get periodic motions.

We expect symmetric motions to emerge when symmetric pairs of joints go through the same sequence of motions, not necessarily at the same time, but most likely with a phase shift. Of course, noise from the sensors or interference with the environment will cause deviations, but the overall image should be symmetrical.

To constrain the controller to be symmetric in this way, we change its structure as

[3] Most animals are structured externally symmetrical but internally asymmetrical. In fact, there is also a range of interesting externally asymmetrical animals such as fiddler crab, narwhal, plaice and snails with shells.

follows: We constrain the controller outputs of the left and right body parts, $\boldsymbol{u}_l$ and $\boldsymbol{u}_r$, to use common weights. From the state vector $\boldsymbol{x}$, we create a *mirrored state vector* $\bar{\boldsymbol{x}}$ in which all left and right sensor inputs are swapped. The controller output is then split into

$$\begin{aligned}\boldsymbol{u}_l(t) &= f\left(\boldsymbol{W}\,\boldsymbol{x}(t)\right)\\ \boldsymbol{u}_r(t) &= f\left(\boldsymbol{W}\,\bar{\boldsymbol{x}}(t)\right)\end{aligned} \tag{3.1}$$

which are the output vectors for all left and right motors, respectively. The new *common* weight matrix is $\boldsymbol{W}$. For example, if the output of the left hip joint is partially composed of the velocity signal of the right knee joint (with a corresponding non-zero weight), then the output of the right hip joint must also be partially composed of the velocity of the left knee multiplied by the same weight.

The number of joints for a given robot is the sum of the number of symmetric joints D and the number of asymmetric joints J, so the number of sensor inputs is $N = 3(D+J)+3+1$, namely angles, velocities, last controls, three accelerations and the bias. If we are able to approximately halve the number of joints by making them symmetric, we can significantly reduce the number of weights (ideally also up to 50%) and obtain a reduced weight matrix $\boldsymbol{W} \in \mathbb{R}^{K' \times N}$ with $K' = D/2 + J$.

robot	D	J	N	KN	K'N
crawler	0	2	10	20	20
tadpole	4	0	16	64	32
quadruped	12	0	40	480	240
humanoid	20	3	73	1679	949

Table 3.1: The size of the controller weight matrix can be reduced by up to 50% for robots with predominantly symmetrical joints.

3.4 Fitness Functions for Robots with Limbs

The fitness of an individual shall be maximized. For controllers, this means that higher fitness must somehow reflect better performance in walking. What does this mean? So a measure of *individual A walks better than B* must be found. In fact, there are several possible approaches.

3.4.1 Running Fast

For each individual to be evaluated, we specify a maximum trial time T and measure its walked distance d_y, e.g., forward. This distance can be easily measured in simulation and is denoted by F_w. However, what we implicitly evaluate is the walking velocity of the individuals, since the distance per time is speed. We further constrain the direction of the robot's velocity vector by measuring only the forward component (denoted by y).

To be precise, we measure the distance as the minimum total positions $\boldsymbol{p} = (p_x, p_y, p_z)$ of the body parts, where $\boldsymbol{p}(0)$ is the starting position of the robot:

$$F_w = d_y = \min_i \left(p_y^i(T) - p_y^i(0) \right) \tag{3.2}$$

To develop walking modes for different directions, we can simply measure the distance backwards or sideways; and with only a little more effort, walking at certain angles or curvatures is also feasible. For forward and backward directions, we can apply the symmetry assumption as described in subsection 3.3.1 to reduce search time. The situation is somewhat different if we want to develop a controller for turning the robot on the spot. To create a fitness function for turning, we need to average the rotations of each body segment, and a symmetric controller is certainly not helpful here.

In its current form, this measure says nothing about what running should look like. In fact, it does not even say that we expect the robot to *run* at all. Any possible way of moving forward is valid under this objective function. Thus, for robots that are not already lying on the ground, e.g., quadrupeds or humanoids, this kind of simple fitness function does not yet work as desired. Assuming that evolution begins with the robot in a standing position, measuring only distance is likely to favor those individuals that simply fall forward quickly.

3.4.2 Constraints

We stuff the «holes», in our fitness functions with further constraints. We can fix the *early falling problem* mentioned above by measuring whether the robot has fallen (or at what point it will fall) and then terminating the experiment prematurely. Again, this is simple to realize in simulation. It is sufficient to monitor the height of the torso's center of gravity above the ground. If this slips below, say, 50% of its initial height, we abort the experiment, and only the distance traveled up to that point is given as fitness. To make this contraint more robust, we still need to subtract a small distance as *penalty*, e.g., twice the torso height, and do this every time the trial has to be aborted because of a fall. This helps prevent evolution from engaging in risky running controllers that normally tend to fall at the very end of their trial. This showed to easily happen when controllers evolve that cause the robot to reach a too high terminal velocity and hence become unstable. I refer to this constraint as *fall prevention.*

To further reduce the search time, we can also abort the trial when the robot goes off the path. We can simply define a (virtual) corridor of, say, 0.5 m around the preferred path, and if a robot is about to leave this corridor, the trial is aborted, similar to falling. I refer to this contraint as *track keeping.*

Target Velocity

The fitness function for fast running will produce individuals who outperform their previous generations in terms of average movement speed and repeatability. But sometimes we do not want to achieve behavior that gets faster and faster but want the robots to move reliably at a certain speed. Designing controllers that maintain certain target speeds is possible by limiting the rewarded distance. We simply need to stop rewarding for greater distances than can be achieved at the desired speed. This favors more stable walking. Let me give an example.

If we expect the robot to run at an average speed of 1 m/s, we reward only up to a maximum distance of 10 m for a fixed trial time of 10 s. When most individuals in the population have reached this limited fitness value, there is no need to run faster. But there is still room for improvement in reliability, since individuals who repeat their performance consistently are preferred on average to those who reach the fitness value only occasionally. In other words, at a certain point in evolution, the evolutionary pressure shifts from running faster to running more stably, as individuals that leave the corridor or fall achieve significantly lower fitness. Thus, by the end, when almost all individuals in the population have reached more or less the same speed, only those who avoid falls and stay on track better will be able to be more successful than others.

3.4.3 Walking Efficiently

So far, we have only looked at fast running, but have not yet looked at efficiency. Efficient locomotion is characterized by minimizing the amount of energy required for locomotion. Unlike fast running, efficient walking requires us to move only at a (presumably lower) speed suitable for reducing the total energy required for each attempt to the minimum necessary. The seemingly trivial solution of not expending any energy is not valid here, because it leads to no distance covered if the plane of the running track is horizontal on average.

Lets define the estimated Energy $\mathcal{U}(t)$ as a function of output motor voltages $\boldsymbol{u}$ of all robot joints since the beginning of the trial until time step t as

$$\mathcal{U}(t) = \sum_{t'=0}^{t} \|\boldsymbol{u}(t')\|_2^2 \tag{3.3}$$

where $\|\cdot\|_2^2 = \sum_i (\cdot)^2$ is the squared euclidian distance.

We can derive this simple measure from the fact that electrical power is defined as $P = UI = U^2/R$ the ratio of squared voltage and resistance. If we assume that the resistance (of the motor coils) is approximately constant[4], and when we add all squared motor voltages, we get the sum of the partial electrical powers. Integrating this total power over time, we get *work* (or energy), so summing the instantaneous power values of all time steps since the beginning of the experiment gives a simple approximation of the (expected) energy used.

Fitness is still the measured distance in the target direction. But in this case, we constrain individuals to use only a limited amount of energy. We terminate the trial when $\mathcal{U}(t) > \mathcal{U}_{max}$, with trial time $T = \infty$. Thus, we now change the behavior from fast to efficient, as we prefer individuals traveling longer distances with less energy. Here we notice that we maximize *distance per energy used* (in units of $[m/Ws]$), which is also *velocity per power* or simply the inverse of the forces used. There is no penalty if the robot's limbs move passively without being driven by the motors, e.g., in swing phases.

[4] The resistance of motor coils usually depends on temperature, but this is neglected in our simulation.

However, we also do not explicitly reward doing nothing, only minimizing the effort required to move. Rather, we prefer controllers that effectively use the robot body's momentum.

3.4.4 Slowing Down and Stopping

In contrast to crawling on the floor, there is no easy way to stop when running. This is especially true if the robot has already reached higher walking speeds. It must slow down before switching to a controller for stable standing. Otherwise there is a high probability that the robot will trip and fall over. But how can we create controller for slowing down and stopping when there are already ones for walking and running? The simple answer is to use the means already created and *evolve* an efficient deceleration controller by creating a suitable fitness function for it. Again, we can use the energy measure presented in subsection 3.4.3.

A fitness function for slowing down a running robot that has worked for me is

$$F_s = a\,(1 + \mathcal{U}_T)^{-1} \tag{3.4}$$

using the energy measure $\mathcal{U}_T = \mathcal{U}(T)$ (Equation 3.3) defined above. But this time we calculate for the total energy consumed up to the last time step T. We also add a correction factor $a \approx 10$ to balance this fitness function with the fall prevention and track keeping constraints.

The fitness for the deceleration controllers is higher for those that expend less energy in a fixed trial time T. Controllers with smaller absolute values of motor control outputs averaged over all time steps are higher rewarded.

So, for each walking controller, we can use this approach to evolve a corresponding decelerating controller that will slow down the robot, eventually stop it at the end, and keep it at a stable standstill. To do this, we need to start each trial with the original walking controller, and when the robot has reached sufficient speed, we switch to the controller to evolve and start the timer. We initialize the weights as a mutated variant of the corresponding walking controller we want to slow down.

So for a certain initial time (initialization phase), the robot is allowed to accelerate to a moderate speed before the actual trial starts and we measure its fitness in decelerating. If we are trying to minimize energy but do not care about the distance traveled, the best way for the controller to conserve energy is to stop quickly. After all, stopping consumes less energy than moving forward. However, the robot is also encouraged to stop carefully, since a fall is still penalized by the above mentioned fall prevention constraint and stopping too abruptly will lead to a stumble.

Therefore, the expected controllers must be damped oscillatory controllers because (1) they have inherited the basic properties of their moving counterparts, (2) they are rewarded for dissipating the energy of periodic motion, and (3) they must still preserve the periodicity of motion because the robot cannot brake too quickly without losing control of its equilibrium. We recall that the robot can only lose speed rapidly if it exploits ground contact in a controlled way without falling.

In terms of dynamical systems, the vector field of the controller should possess a stable fixed point or a small limit cycle, which can then be observed as a *standing robot*. And as a welcome side effect, these stabilizing controllers also respond to shocks or other disturbances with corresponding stabilizing movements.

3.4.5 Seeds

Initial parameter sets that go beyond random initialization are usually termed *seeds*. For the very beginning of our evolution of walking experiments we use a manually created parameter set which is able to fulfill the requirement of holding the robot's default position, i.e., standing and keeping balance, but at the same time performs slow, low amplitude oscillatory movements around the default position. This is mandatory since otherwise, our generalized controller will probably not move at all. Initial conditions that do not change the robot's joint positions will most likely evolve very slowly. Remember the controller can way more easily produce non-constant output when non-constant input is applied.

The initial parameters are created from a mixture of P-controller and CSL weights randomized around the hold-mode (referring to section 1.3). Hence, due to the P-controller, there is a strong restoring force towards the joints' default position and the CSL's behavioral mode will either tend to release (damping) or towards contraction (undamping), just depending on a little random fraction added to feedback gain $G_f = 1$. Such a controller will make the robot balance but will always create a certain portion of overshoot and delayed reaction, so it actually never stabilizes but instead changes the fixed point of standing into a repellor with for the moment unclear surrounding attractors. A little instability serves therefor as the basis for forming stable periodic attractors in the sensorimotor loop. Because of its kind of wobbling behavior, I called this seed controller *Unruhe* (German for restlessness or impatience).

But seeds must not necessarily be hand-tuned controller weights. After having evolved the first stable locomotion controllers, they can easily be utilized as seeds for further experiments. For instance, if we want to evolve running, we can start from results for walking slowly.

Summary

The fitness functions and constraints discussed so far allow for a broad spectrum of robot behaviors related to walking. With little modification, we can evolve walking or crawling in different directions, either fast or efficiently, and we are even able to evolve turning or slow-down controllers that eventually stop a gait motion.

The number of weights can be reduced by a symmetry-assumption in order to speed up the optimization, but this can only be applied to behaviors that are expected to be symmetrical, e.g., running forwards or backwards. Since we are interested in what is in general possible with the kind of controller, we refrain from evolving sparse weights, using the aforementioned L1-normalization. For now, we postpone further optimization on the structure itself and concentrate on the behavioral aspects.

3.5 Experiments and Results

In this section I want to give a summary of the most interesting behaviors that I was able to find with the aforementioned approach and techniques. The order of the results is arranged according to their increasing complexity. And because moving robots are difficult to show in pictures or graphs, there is a video file included with each example to better capture the results.

All experiments are conducted and repeated several times to demonstrate the stability of the results. Since both, the fitness functions and the optimization are stochastic in their very nature, there are acceptable differences in subsequent outcomes, but all hyperparameters were chosen accordingly to maximize the reproducibility of the results.

To outline this section, we begin with the most fundamental crawling behaviors of the robots Tadpole and Crawler and with the walking forwards motion of the legged robots. We continue with describing the change of the objective function from efficient towards fast movements and will see how the same approach scales to almost all styles of directional movements.

Crawling

We start with crawling because we might expect it to evolve easier than walking. After all, the torso is already close to the ground and needs less balance to be kept or recovered. Indeed, the robots *Crawler* and *Tadpole* can only move at ground level since their limbs are far too short to lift their torsos significantly. The first evolved behavior is moving forward. As the fitness function, we set up the forward criterion combined with the efficiency and track-keeping constraints. As initial parameters, we choose the manually created Unruhe controller as described in subsection 3.4.5.

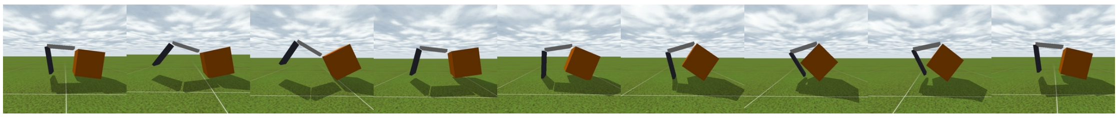


Figure 3.4: The Crawler robot moving efficiently on the ground. Crawler is sliding over the ground with a little lift-off moment.

Remember, the robot Crawler cannot actively move sideways, and the friction between the body and the limbs is low. Therefore, we might have expected the robot would pull itself over the floor in a periodic movement characterized by a phase shift between the joints. The shoulder joint exerts the bulk force in the motion while the elbow joint is more or less only thrown forward and then held rigidly after hitting down. A striking feature is that when moving, the body gets tilted to a considerable angle and pulled only on the edge. After reaching higher speeds, the robot's body slightly lifts off the ground and shortly remains lifted.

The crawling movement of Tadpole is entirely different. Tadpole does not slide on the floor. The limbs slightly lift the body in each stroke. There emerged two distinct strategies for this. One is comparable to the breaststroke, the other to the crawl swim. In the first, both limbs move forward and backward synchronously, and the other has a

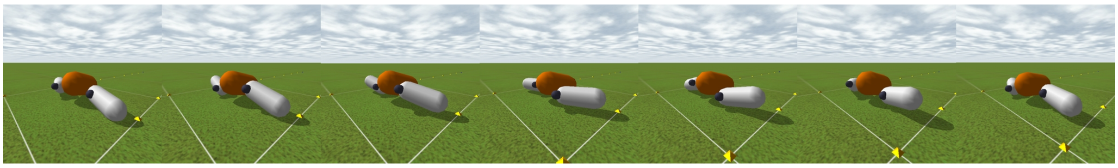


Figure 3.5: The Tadpole robot moving efficiently on the ground with its arms moved with a phase shift.

phase lag between shoulder joint movements.

Walking efficiently

When we apply the forward fitness function to the Quadruped and the Humanoid, we also require the fall-prevention constraint to avoid falling since both robots start from the standing position. For both legged robots, stable walking successfully evolved when initialized with the Unruhe controller.

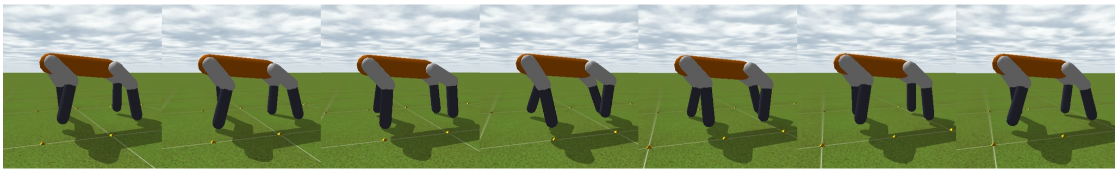
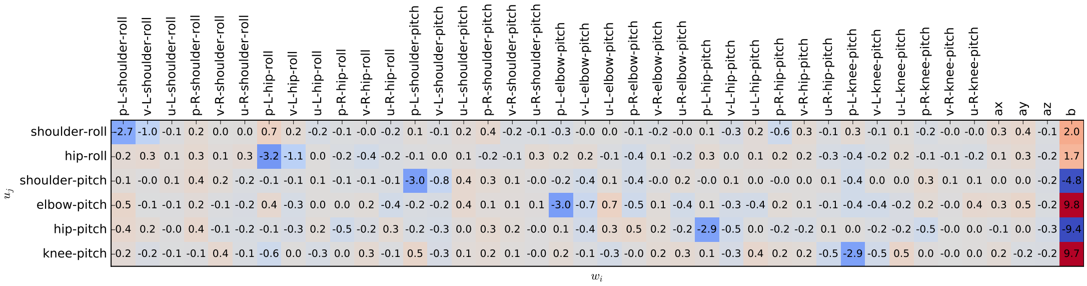


Figure 3.6: The quadruped robot walking efficiently making moderate steps.

As can be seen in the resulting weight matrix of Figure 3.7, the proportional-derivative controller structure is preserved, whereas the initial feedback weight near 1.0 disappeared.



u_j	p-L-shoulder-roll	v-L-shoulder-roll	u-L-shoulder-roll	p-R-shoulder-roll	v-R-shoulder-roll	u-R-shoulder-roll	p-L-hip-roll	v-L-hip-roll	u-L-hip-roll	p-R-hip-roll	v-R-hip-roll	u-R-hip-roll	p-L-shoulder-pitch	v-L-shoulder-pitch	u-L-shoulder-pitch	p-R-shoulder-pitch	v-R-shoulder-pitch	u-R-shoulder-pitch	p-L-elbow-pitch
shoulder-roll	-2.7	-1.0	-0.1	0.2	0.0	0.0	0.7	0.2	-0.2	-0.1	-0.0	-0.2	0.1	-0.1	0.2	0.4	-0.2	-0.1	-0.3
hip-roll	-0.2	0.3	0.1	0.3	0.1	0.3	-3.2	-1.1	0.0	-0.2	-0.4	-0.2	-0.1	0.0	0.1	-0.2	-0.1	0.3	0.2
shoulder-pitch	-0.1	-0.0	0.1	0.4	0.2	-0.2	-0.1	-0.1	0.1	-0.1	-0.1	-0.1	-3.0	-0.8	0.4	0.3	0.1	-0.0	-0.2
elbow-pitch	-0.5	-0.1	-0.1	0.2	-0.1	-0.2	0.4	-0.3	0.0	0.0	0.2	-0.4	-0.2	-0.2	0.4	0.1	0.1	0.1	-3.0
hip-pitch	-0.4	0.2	-0.0	0.4	-0.1	-0.2	-0.1	-0.3	0.2	-0.5	-0.2	0.3	-0.2	-0.3	0.0	0.3	0.2	-0.0	0.1
knee-pitch	-0.2	-0.2	-0.1	-0.1	0.4	-0.1	-0.6	0.0	-0.3	0.0	0.3	-0.1	0.5	-0.3	0.1	0.2	0.1	-0.0	0.2

u_j	v-L-elbow-pitch	u-L-elbow-pitch	p-R-elbow-pitch	v-R-elbow-pitch	u-R-elbow-pitch	p-L-hip-pitch	v-L-hip-pitch	u-L-hip-pitch	p-R-hip-pitch	v-R-hip-pitch	u-R-hip-pitch	p-L-knee-pitch	v-L-knee-pitch	u-L-knee-pitch	p-R-knee-pitch	v-R-knee-pitch	u-R-knee-pitch	ax	ay	az	b
shoulder-roll	-0.0	0.0	-0.1	-0.2	-0.0	0.1	-0.3	0.2	-0.6	0.3	-0.1	0.3	-0.1	0.1	-0.2	-0.0	-0.0	0.3	0.4	-0.1	2.0
hip-roll	0.2	-0.1	-0.4	0.1	-0.2	0.3	0.0	0.1	0.2	0.2	-0.3	-0.4	-0.2	0.2	-0.1	-0.1	-0.2	0.1	0.3	-0.2	1.7
shoulder-pitch	-0.4	0.1	-0.4	-0.0	0.2	-0.0	0.1	0.0	-0.0	0.0	0.1	-0.4	0.0	0.0	0.3	0.1	0.1	0.0	0.0	-0.2	-4.8
elbow-pitch	-0.7	0.7	-0.5	0.1	-0.4	0.1	-0.3	-0.4	0.2	0.1	-0.1	-0.4	-0.4	-0.2	0.2	-0.0	0.4	0.3	0.5	-0.2	9.8
hip-pitch	-0.4	0.3	0.5	0.2	-0.2	-2.9	-0.5	0.0	-0.2	-0.2	0.1	0.0	-0.2	-0.2	-0.5	-0.0	0.0	-0.1	0.0	-0.3	-9.4
knee-pitch	-0.1	-0.3	-0.0	0.2	0.3	0.1	-0.3	0.4	0.2	0.2	-0.5	-2.9	-0.5	0.5	0.0	-0.0	0.0	0.2	-0.2	-0.2	9.7

w_i

Figure 3.7: Weight matrix of the best efficiently forwards walking quadruped. The PD-controller structure is preserved, local weights dominate and interconnected weights create a rich and dense mixture of *state* information as setpoints for the local controller.

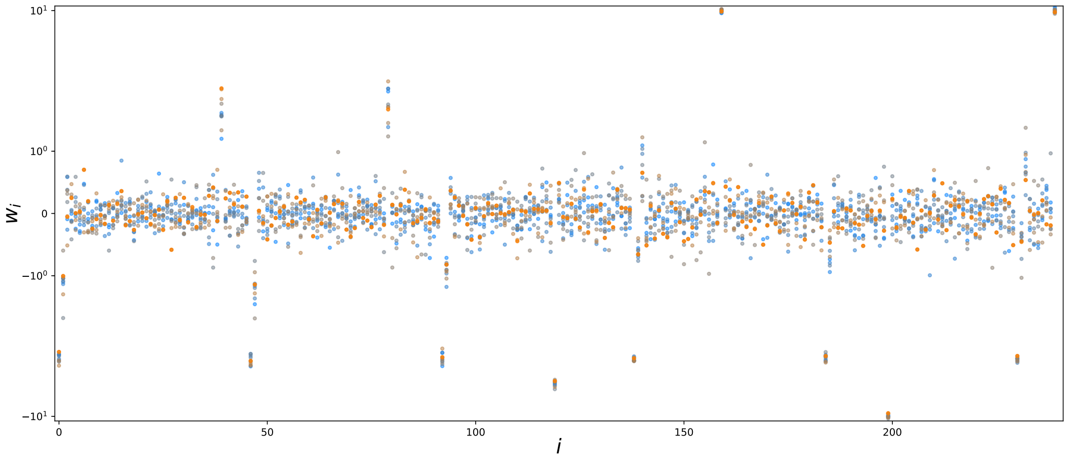


Figure 3.8: We see the weights for the quadruped forward efficient walking experiment displayed for the top-performing individuals from 10 distinct experimental runs. Although the already mentioned local weight clusters and the PD structures are present, the other weights primarily concentrate densely around the $[-1, +1]$ range across the entire matrix. The absence of clusters indicates the abundance of oscillatory sensor information available for the network to utilize in stabilizing the walking attractor, with hardly any additional visible similarities between controllers with very similar performance.

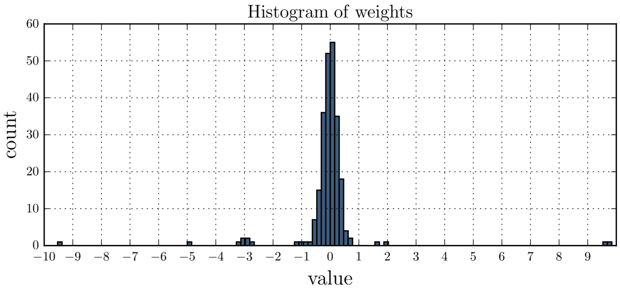


Figure 3.9: Distribution of the controller weights of the best efficiently forwards walking individual. The large weights ±10 are the bias weights due to the intentionally selected bias input of 0.1. The −3 weights are primarily the k_p part of the PD controller substructure.

Running fast

For evolving running fast, we can benefit from previous results. As a seed, I used the previously evolved gait controllers for efficient walking. This has shown to generally lead to more predictable outcomes and has a higher probability of creating comparable results over multiple repetitions of the same experiment. Starting from scratch the experience has shown, that the results tend to be quite different when running the same experiments several times. When we are interested in finding different gaits or hopping styles and the absolute velocity is of minor concern, then starting from scratch is the preferred option. The fitness landscape, apparently, has many more local optima with quite special gaits, when efficiency is not a requirement.

With the crawling robots, however, there was less difference between fast and efficient. I suspect that since the movements on the floor generally require more energy to push through against friction, there was less potential for optimization. The humanoid robot developed a particularly unique walking style, in which it strongly articulated its arms and leaned its back.

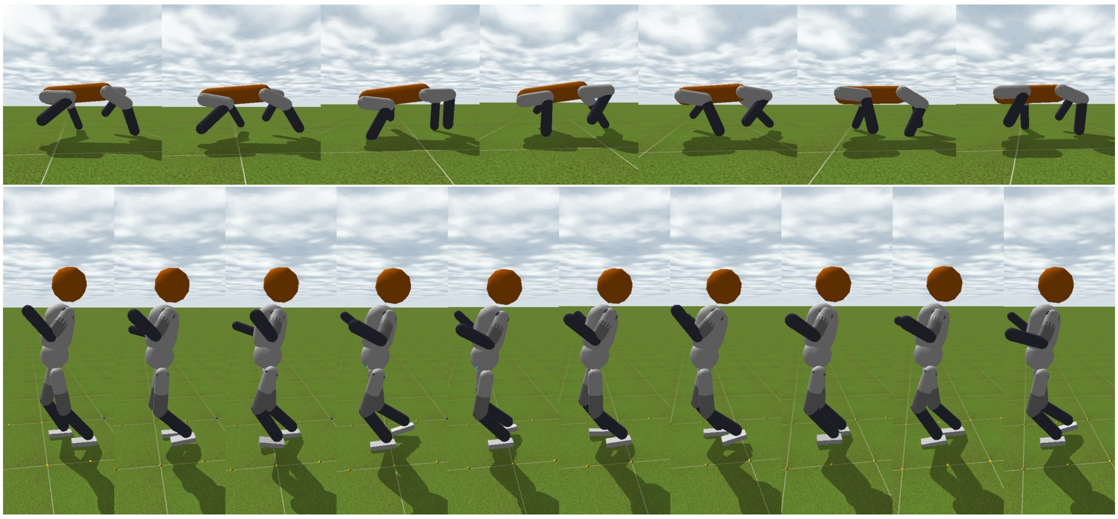


Figure 3.10: The Quadruped and Humanoid running fast: As can be seen from the pictures during quadruped running flight phases occur with all four legs lifted off the ground. I did not expect this result especially because the body segments of the robots are created from rigid bodies only. The Humanoid also walks forward in a very particular style, waving his arms and leaning very far back.

Directional Locomotion

Up to now, for each evolution experiment described above, I evaluated the moved distance forwards. By changing the sign, the robot is expected to evolve the respective backward gait. Same counts for sidesteps of the Quadruped or turning of Tadpole.

Conveniently, we only need to evolve left *or* right sideward/turning movements, since the robots are symmetrical and we can swap the parameters accordingly. Since there is no forward-backward symmetry, we should expect the backward gaits to look differently than the forward gaits.

Not quite obvious and hence even more interesting, when removing the fall prevention constraint for the quadruped moving sideward, the result changes to rolling over. Since the fall prevention constraint was used to avoid tumbling over at the beginning of the trials, we explicitly can make use of this effect now. The morphology of the quadruped allows the controller to keep the momentum when falling, so multiple turns are possible.

Starting and Stopping

There are at least three working methods for starting the neural controller. We assume the robot is already in a default position, such as standing stable. Since the gait controller connected to the robot's physics forms a quasi-periodic attractor in multiple dimensions, there is a good chance that the system independently starts oscillating if its state is already in the gait attractor's basin. If not, it might be at a fixed point nearby so that

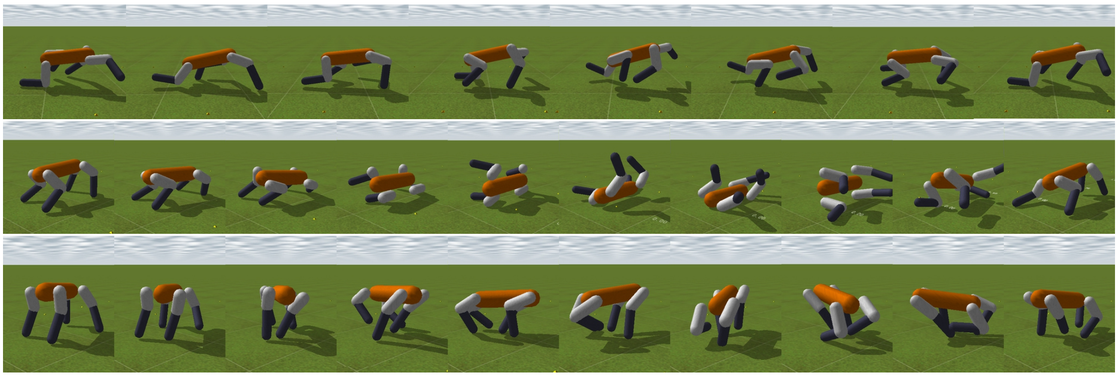


Figure 3.11: The quadruped running backwards, rolling sidewards, and spinning on the spot.

a tiny nudge would suffice to enter the gait's basin. So, if the system needs help getting into the attractor, we have two simple options. We can initialize the *last* motor values with some noise or explicitly insert a significant pulse to the motor outputs for a few time steps. The third option for getting into the target basin is causing a significant change in the robot's sensory inputs by physically nudging the robot with external perturbations. To increase the basin of attraction, we can randomize the robot's default initial position during evolution so that the network has to find ways to start walking from varying robot poses.

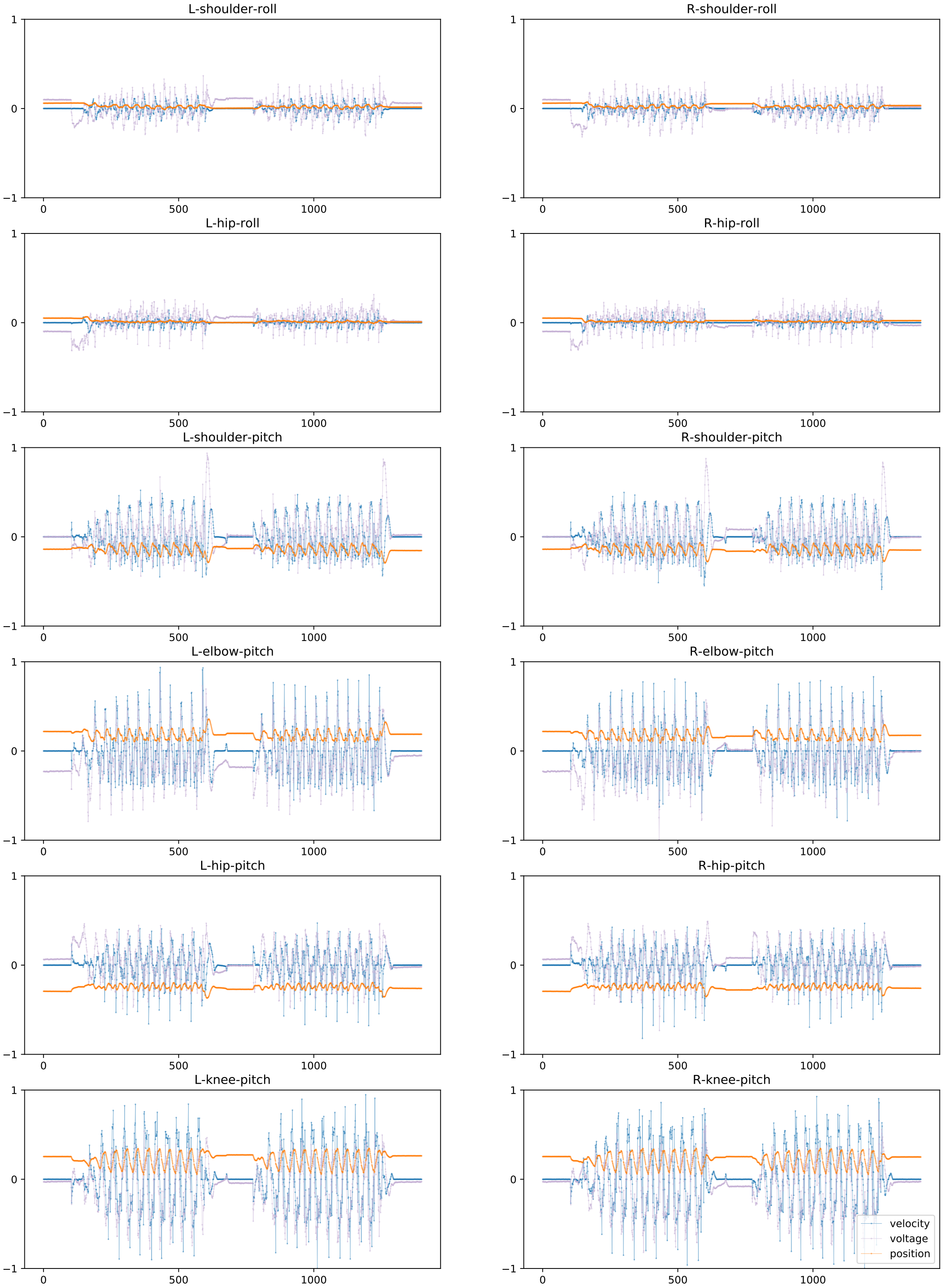


Figure 3.12: Two complete sequences for starting and stopping of the quadruped robot using a separately evolved controller for stopping. Position, velocity, and voltage vs. time steps are plotted for each of the 12 joints. Hip and shoulders' roll joints are merely holding the position whereas the pitch joints create the mechanical power for the forward motion. At each breakpoint, the joint position is slightly different. The controller still finds its way back to the attractor.

Moving under Disturbance

To help the robots evolve robustness in walking, I generally found it better to automatically nudge them randomly during training. It also supported them to overcome situations in early training where the controller could not reliably initiate the motion. Fortunately, it also turned out that these small forces did not significantly change the appearance of the gait. However, when the simulation pushed the robot too hard, it resulted in very conservative, almost tense-looking locomotion.

Another source of uncertainty is the occurrence of obstacles in the environment as described in subsection 2.3.1. To begin with, we put some randomly distributed and differently sized rocks in front of the robot. The setup is the same as described before but it is now a lot more difficult to avoid stumbling. Even though they are regularly distanced, the steps of successively increasing height cannot be predicted by any robot due to the lack of visual feedback or long-term memory in the controller. It is hence a source of uncertainty, increasing with every meter passed.

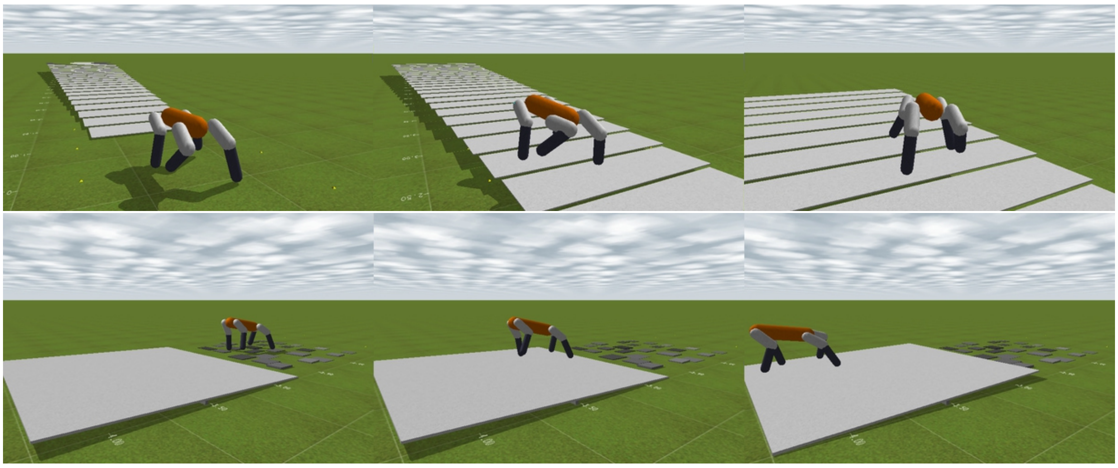


Figure 3.13: The quadruped walking with disturbances to increase the stability of the gait and to increase its capability to deal with moderate slopes.

Operators and Hyperparameters

For recombination two individuals are selected for crossover from the population with a selection bias towards higher fitness. Each parameter has 50% chance of originating from either parent. Mutation is applied by adding a small normally distributed random value to each parameter $w_{ij} \in \boldsymbol{W}$,

$$w_{ij} \leftarrow w_{ij} + \frac{\chi}{\sqrt{n}} \tag{3.5}$$

with $\chi = \mathcal{N}(0, \sigma)$, mutation rate σ with $\sigma(0) = \sigma_0$ and the individual's size n. The value is normalized by the number of parameters in order to have the same amount of mutation invariant to the specific robot controller.

Population size and mutation rate are among the most critical hyperparameters we must define in advance. For mutation rate, it is beneficial to use an adaptive approach such as proposed by Thierens (2002) or Hansen and Ostermeier (2001) for automatically adjusting a suitable value. We can regard the mutation rate as a property of the individual and leave it to evolution to find good values. So, the mutation rate can be specific for each individual and can mutate itself. To do this in a controlled way, we chose a constant *meta-mutation rate*, $\dot{\sigma}$. We draw random values from a limited normal distribution, $\xi = \mathcal{N}(0, \dot{\sigma}) \in [-\lambda, \lambda]$, with $\lambda = \ln 2$, then change the mutation rate by

$$\sigma \leftarrow \sigma e^{\xi}, \tag{3.6}$$

and further bound it by the interval $[10^{-4}, 1]$. So the mutation rate can adapt quickly, since it is able to change exponentially, but can maximally be doubled or halved in each step. It can neither become too small to be ineffective nor too large to destabilize the optimization process.

A self-changing mutation rate implicates that individuals only survive who find appropriate values for their mutation rate depending on their current position in the fitness landscape. This can be used to either increase mutation, e.g., when being on a plateau or to decrease it for climbing up to narrow optima. So the problem of finding suitable mutation rates is reduced to specifying a less critical meta-mutation rate $\dot{\sigma}$ and to finding a good initial value σ_0 since it directly influences the features present in the starting population.

moving rate	μ	0.99
meta-mutation	$\dot{\sigma}$	0.5
max. trials	$\mathcal{T}$	$5 \cdot 10^5$

Table 3.2: Hyperparameters used for most of the experiments.

A method for determining good hyperparameters is to perform parameter sweeps. So to find a suitable pair (P, σ_0) a series of experiments were performed under variation of population size and initial mutation. The results are depicted in Figure 3.14 and demonstrate that the parameters can be selected from a rather broad range without significant loss of success.

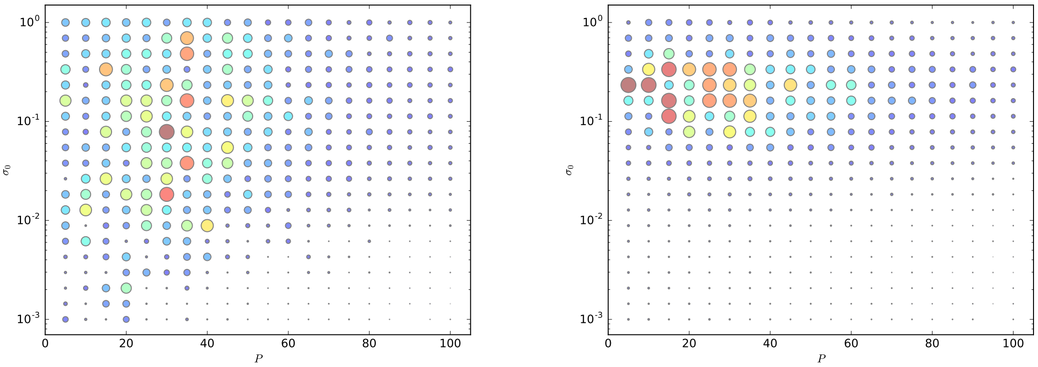
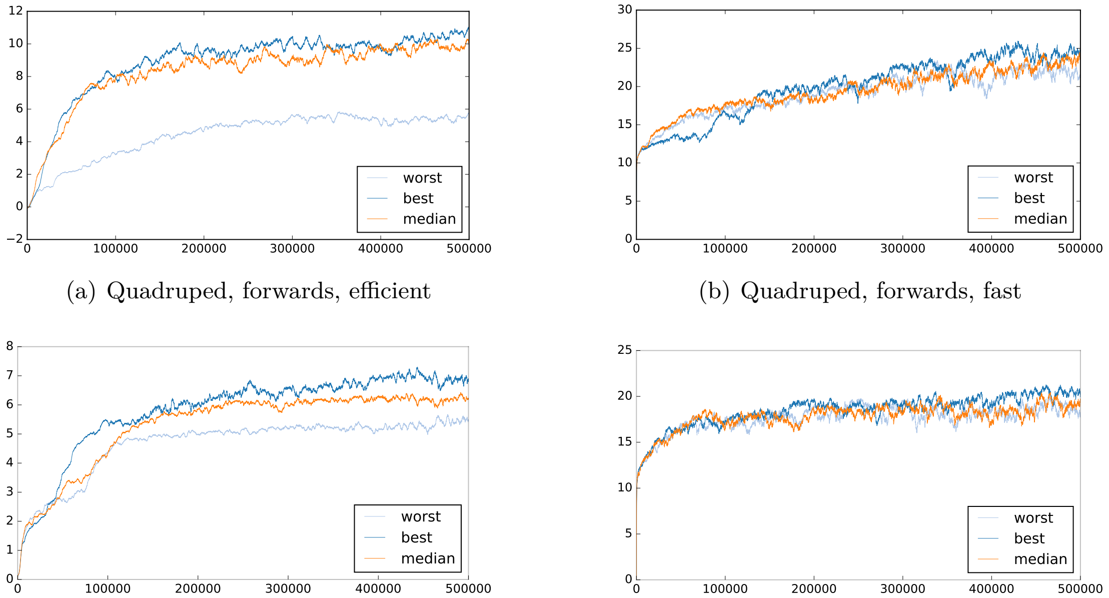


Figure 3.14: We see the fitness after the first 10^4 trials depending on population size P and initial mutation rate σ_0. We can select the hyperparameters for further experimentation from a comparably broad epicenter of sufficiently high fitness values, e.g., $P \in [20, 40]$, $\sigma_0 \in [0.01, 0.1]$. The left figure shows a higher meta-mutation rate of $\dot{\sigma} = 0.5$ while the right one has $\dot{\sigma} = 0.001$. The auto-adjustment of mutation rate helps to broaden the area of best results because it allows for a more coarse initialization of σ_0. A higher meta-mutation rate $\dot{\sigma}$ increases the area of higher fitness but also introduces the chance of suboptimal results.



(a) Quadruped, forwards, efficient

(b) Quadruped, forwards, fast

(c) Quadruped, backwards, efficient

(d) Quadruped, backwards, fast

Figure 3.15: Exemplary plots of fitness for forward and backward walking evolution of quadruped: Figures showing the worst, best, and median of 10 experimental runs per gait. Fast locomotion was seeded with converged results of evolution runs from efficient locomotion.

Conclusion

When looking at the average fitness outcome of the evolutions for forward and backward movements for multiple idendtical runs of each experiment there is a certain stability in the results indicating the solid path to the optima. In contrast to Crawler for Tadpole there is more ripple on the fitness development mainly because the capsule-shaped body introduces balance issues while moving. Also, local fitness maxima are more pronounced resulting in different (but totally valid) styles of locomotion. For both: There is no big difference between the fast and efficient results because both robots consume much power while sliding over the ground due to the fact that they must pull their bodies over the ground. When walking on legs the difference between fast and efficient motions gets more significant. Starting from scratch mostly results in a variety of motion styles while starting from efficient gaits for fast gait evolution results in more or less the same final gait.

Although the outcomes sometimes differ in their appearance, they are mostly valid gaits. Despite the fact that local optima in the fitness landscape seem to be various they are meaningful in the majority of cases. The usage of the constraints for symmetry and fall prevention and a proper initialization play a big role in this. The found solutions can mostly be optically identified with solutions we would also expect to be found by nature, in terms of the recognized gait patterns for instance. By simply looking at them we can accept them as «correct».

It has to be noted that, when not using the mentioned constraints or using random initialization then plenty of silly gaits can be found easily and the outcome of repeated trials are mostly unpredictable. Even with longer trials, a once found local optimum is rarely left again. This probably indicates that the optima are quite distant and narrow in the weight space.

It was found that the specific morphology of the robot, in particular, the number of joints is moderate enough for the proposed structure to create reasonable results. The number of parameters grows quadratically with the number of motor outputs. However, if we aim to increase the degrees of freedom significantly, we may need to leverage the modularity of the system to manage complexity, as suggested by Clune et al. (2013). Currently, with around two dozen joints, this is not a pressing concern.

Summary of Part One

Part one outlined my way from manually created single-joint motions to machine-learned highly dynamic full-body robot locomotion with hundreds of parameters and many hours of automated training. Based on existing controls and dynamical systems theory, I reasoned a generalized control structure and intensively explored its effectiveness and capabilities. I found plenty of robot behaviors for a spectrum of morphologies with the help of evolutionary algorithms and a particular set of fitness functions. Within the given control framework, we can undoubtedly discover many more exciting and meaningful behaviors for these or similar robots.

The examined controller uses inputs available at the raw proprioceptive sensor level. So, the simplicity of its structure in implementation and computational effort for training and real-time execution is attractive for running on embedded systems and is energetically beneficial for mobile robots. A subtle outcome of this first part is the quiet hope that the applicability to real robots may be within reach through the consistent avoidance of robot-specific details in the controls and the use of more realistic friction and actuator models.

While related work often focused on characteristic locomotion for a specific morphology, I can say that a primary contribution from the first part is the systematic study of diverse behaviors for a meaningful range of morphologies that grows in complexity. I concentrated on a model-free and purely sensor-driven controller. Other approaches frequently use pattern generators or directly incorporate models of the robot. Furthermore, I created minimalist fitness functions for running fast or efficiently in various directions and styles, complemented by reasonable constraints to improve the results. Finally, in contrast to current trends making the control networks ever deeper by adding numerous layers, I see another relevant contribution by restricting controllers to a sufficiently low complexity—*shallow learning*—so to speak. With this comes the hope of broader applicability to real-world low-cost robots with limited computing and energy resources.

When the search space is vast, researchers sometimes consider the results of evolution experiments *lucky*. But even though it needed some persistence and time to find the appropriate setup, I can generally comment that the given controller, fitness functions, constraints, and initial conditions described here reliably produce the documented results. Concluding this part, the most important outcome is the confidence I gained about my expectations of what kind of behaviors I could generally find given a particular robot and the proposed controller structure. We might consider the results generated with the evolutionary approach as a benchmark for other methods.

Part II

Towards Self-Organized Learning Machines

4 Elements of Embodied Self-Learning

This chapter presents the basic learning methods used in this part. Although our robots should ideally learn independently without supervision, the particular components equally stem from other areas like supervised or reinforcement learning. Together in their combination, only the source of the learning signals matters for the overall system, whether they have to be specified or whether the system generates them itself. This chapter can only briefly introduce the used methods and their classification without claiming to be complete. We can find a broad overview of popular learning algorithms in Haykin (2008), Goodfellow et al. (2016), or Sutton and Barto (2018).

The chapter follows this organization: First, we will delve into supervised learning of neural networks and examine their training procedures. Moving on, we will discuss methods designed to identify structures in data independently. Subsequently, I will review a technique for reinforcing machine behavior by providing rewards. And lastly, we will explore a unique type of reward that originates from the robot itself and encourages autonomous learning in the machine.

Figure 4.1: Illustration of the text prompt: *«A neural network trained with stochastic gradient descent»* generated by a neural network trained with stochastic gradient descent.

4.1 Supervised Learning

Supervised learning is training from examples. Supervised learning algorithms are based on the idea that the problem to be solved is described in terms of an objective function and a data set which contains labeled training samples. We consider the data points given as examples of an unknown process whose distribution is to be approximated.

The training examples comprise pairs of inputs and desired outputs. The objective is now to make the learner, e.g., a neural network, to associate the inputs to the presented desired outputs. At the beginning of learning the learner's outputs will largely deviate from the desired values. We measure this with the help of an error function, with the objective to minimize that error. And since the solution is already presented, there is the possibility to look at the learner's error and identify what caused the error and what must be changed in order to decrease it. We now adjust the learner's parameters, i.e., the network weights, to successively decrease the error from one example to the next.

Frequent tasks for supervised learning are, e.g., classification and prediction. The desired outputs in classification tasks are termed *labels* whereas in prediction tasks we often find the term *target values*. For classification tasks, the learner is trained to associate labels to the given inputs, for instance, if it is either a donkey, yak, or unicorn which is depicted on the input image. The objective is to give probabilities and maximize the detection probability for the entity which is actually on the picture. In prediction tasks the learner is trained to make assumptions on certain target values, e.g., to predict the next value of a time series, given the samples of the past. Applications are numerous but all have in common, that there is an objective and a collection of training examples. What is needed now is an algorithm which takes the objective function and the data and adjusts the learner to fulfill the objective.

4.1.1 Stochastic Gradient Descent

Consider the differentiable *error function*, E. In order to fulfill the objective of minimizing E, we can derive the error function with respect to its parameters. In general the error depends on the parameter vector $\boldsymbol{\theta} = (\theta_0, \theta_1, \dots)^\top$, for instance the synaptic weights of a neural network. So we can differentiate the error w.r.t. each θ_i. Thus, the gradient $\nabla_\theta E$ gives us information about where we can decrease the error the most. We now follow the gradient where it descents the most by adjusting the parameters $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\Delta\theta}$ with

$$\boldsymbol{\Delta\theta} = -\eta \nabla_\theta E \qquad \text{or the equivalent form for each weight} \qquad \Delta\theta_i = -\eta \frac{\partial E}{\partial \theta_i} \tag{4.1}$$

and $0 < \eta \ll 1$ as the stepsize or learning rate. So each parameter is adjusted by its partial derivative of the error. This is known as *gradient descent.*

In common gradient descent, the data set is once fully processed and the changes are accumulated before applying the changes to the parameters, which can get slow when using larger collections of training examples. In *stochastic* gradient descent (SGD), we process samples individually and apply changes immediately. Samples are usually presented in a randomized order, to reduce spatial or temporal dependencies between them.

However, since processing single samples can bring high variance, we usually process them in *mini-batches*, small randomized subsets of the full training data set. This, in general, reduces variance and is still computationally favorable.

We need to repeat the cycle of processing mini-batches and adjusting parameters until the error got sufficiently small and the algorithm converges. The learning rate is usually kept small for the training process to be stable since the true gradient is only approximated on a subset of the data. Depending on the step size and the data, SGD will approach a local minimum. There is no guarantee to find the globally best solution using SGD. So the collected training examples must cover all relevant aspects of the distribution to be learned. Also, because we present the training data in randomized order, several runs may yield different results with respect to initialization and learning rate.

We can further distinguish *online* learning, where a process generates the training examples while training simultaneously, and *offline* learning, where we collect all samples already in advance. The online variant usually brings the problem of statistical (or temporal) dependence between consecutive samples. We avoid this by mini-batch-like training, keeping previous process samples in an *experience buffer* and replaying them in randomized order from that buffer.

4.1.2 Backpropagation

Applying (stochastic) gradient descent to multi-layer neural networks is done with a technique called *backwards propagtion of error* or just *backpropagation* as introduced by Rumelhart et al. (1986). The network's output $\boldsymbol{y} = f(\boldsymbol{x}) = (y_0, y_1, \dots)^\top$ is compared to the desired output $\boldsymbol{d} = (d_0, d_1, \dots)^\top$ of the training example $(\boldsymbol{x}, \boldsymbol{d})$ with the help of an error function. This could for instance be the commonly used squared error

$$E = \frac{1}{2}||\boldsymbol{d} - \boldsymbol{y}||_2 = \frac{1}{2}\sum_j (d_j - y_j)^2 \tag{4.2}$$

where the elementwise distances of desired and actual output are accumulated.

Assume we have a feedforward neural network (for now with a single layer)

$$\boldsymbol{y} = f(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x}) \tag{4.3}$$

where $w_{ji} \in \boldsymbol{W}$ are the *weights* of the network, now in the more convenient form of a matrix. If we now calculate the gradient of the error w.r.t. each weight, we need to apply the chain rule to get the partial derivative

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{4.4}$$

with $y_j = \sigma(a_j)$ and $a_j = \sum_i w_{ji}x_i$. The first factor of the right hand side product is $\partial E/\partial y_j = -(d_j - y_j)$. The second term is the derivative of the transfer function $\partial y_j/\partial a_j = y'_j = \sigma'(a_j)$ and the third term is $\partial a_j/\partial w_{ji} = x_i$.

When we put this together to calculate $\Delta w_{ji} = -\eta \partial E / \partial w_{ji}$, the result is the backpropagation learning rule

$$\Delta w_{ji} = \eta (d_j - y_j) y'_j x_i \tag{4.5}$$

which expands to

$$\Delta w_{ji} = \eta (d_j - y_j)(1 + y_j)(1 - y_j) x_i \tag{4.6}$$

when using the transfer function $\sigma(\cdot) = tanh(\cdot)$ and its first derivative $tanh' = 1 - tanh^2$. We see that the weight from neuron i to j changes only in dependence on the neuron's inputs x_i, its output y_j and its respective part of the desired output d_j. For multi-layer networks, for instance with a single hidden layer $\boldsymbol{h}$, defined as

$$y_j = \sigma\left(\sum_k w^y_{jk} h_k\right) \quad \text{with} \quad h_k = \sigma\left(\sum_i w^h_{ki} x_i\right), \tag{4.7}$$

the learning rule needs to be extended, because for neurons from hidden layers there exists no direct training signal. We have to derive it from the above layers. Considering neuron j from a hidden layer, we determine the indirect error as the weighted sum of errors from the above layer:

$$e_j = \sum_k \delta_k w_{kj} \tag{4.8}$$

If we rewrite the learning rule as

$$\Delta w_{ji} = \eta \delta_j x_i \tag{4.9}$$

then for each neuron in the feedforward layered structure we have to decide whether it is a hidden or output neuron.

$$\delta_j = \begin{cases} (1 + y_j)(1 - y_j)(d_j - y_j) & \text{if } j \text{ is output neuron,} \\ (1 + y_j)(1 - y_j) \sum_k \delta_k w_{kj} & \text{if } j \text{ is hidden neuron.} \end{cases} \tag{4.10}$$

So, for each training step, we apply the inputs to the network and forward propagate the information to the outputs. Then, we evaluate the error between network output and desired output and adjust the weights according to the partial derivatives of the error for each neuron, backpropagating the error layer by layer.

Best Practices for Training

If the network's model capacity is too high or if there are too few or too similar training examples we see the undesired effect of overfitting where instead of generalizing, the network memorizes individual samples. So in constantly sized (non-growing) neural structures we predefine a fixed number of layers and their sizes and hence a static model capacity. So we need to carefully adjust these structure parameters to have the network's capacity close to the actually estimated demand.

Usually, the collection of data is divided into a training set and a test set. So when regularly checking the performance of the network against data, which the network has not been trained on (the test set), we get an estimate of how well the network generalizes. When learning is successful the error measured on the training data decreases and proportionally also the error of the test data set. At some point, the error curve of the test set has reached a minimum and would begin to increase again. This is the point where we would usually stop further learning since the network starts to learn the training data *by heart* and starts losing again its ability to generalize.

Initialization of the weight matrix $\boldsymbol{W}$ is commonly done with small random values. To avoid early saturation of units, the weights should be initialized from a distribution with zero mean and standard deviation $\sigma_w \propto 1/\sqrt{N}$ which depends on the number of inputs N to each neuron (LeCun et al. (1998)). If domain knowledge is available, there is the possibility to initialize the network to specific values. Recent methods try to pre-train parts of the network with unsupervised learning techniques (see section 4.2) followed by a supervised learning phase.

From the learning rules, we saw that the gradients can be transported through the layers of the network via backpropagtion which is simply derived using the chain rule from calculus. But gradients tend to vanish when transported through too many layers which can significantly slow down learning. This can be seen from the derivative of the transfer function when we consider neurons whose activation is close to saturation the weight updates get very small. This can happen if the error landscape has wide plateaus where no gradient information in any direction can be detected.

There were many attempts to tackle the *vanishing gradient problem* such as using *momentum terms*, which effectively low-pass filters the weight changes. Another approach was changing the transfer functions to *rectified linear units (ReLU)*, a kind of non-linearity that is less prone to vanishing gradients. Other attempts focus on not transporting gradient values but only the sign in which the individual weight should be changed.

4.1.3 Adam

To bring us even deeper to the heart of neural learning, the last part of this section on supervised learning describes a popular further development of the backpropagation learning method, called *Adam* published by Kingma and Ba (2014). When training with Adam, for each weight w_{ji}, there is a tuple of statistical information $(m, v)_{ji}$ comprising the approximate mean and variance of the local gradient information. The statistics are estimated by a simple low-pass filter approach given the equations

$$m \leftarrow \beta_1 m + (1 - \beta_1) g \tag{4.11}$$

$$v \leftarrow \beta_2 v + (1 - \beta_2) g^2 \tag{4.12}$$

where $\beta_{1,2} \in (0, 1)$ and with $g = (\nabla_w E)_{ji}$ being the local gradient, i.e., the partial derivative of the error w.r.t. to the weight w_{ji}.

From these estimates, we compute a tuple of intermediate values $(\hat{m}, \hat{v})_{ji}$ with

$$\hat{m} = \frac{m}{1 - \beta_1^t} \qquad \hat{v} = \frac{v}{1 - \beta_2^t} \tag{4.13}$$

which compensate for the strong bias towards the initial value of the averaging filter and where $t > 0$ denotes the increasing time step. So the normalizing factors $1/(1 - \beta_{1,2}^t)$ decrease starting from $1/(1 - \beta_{1,2})$ towards 1 when $\beta^\infty \to 0$.

Finally, the update rule for each individual weight update with Adam is

$$\Delta w = -\eta \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon_a} \tag{4.14}$$

using the bias-corrected values $(\hat{m}, \hat{v})_{ji}$ where $0 < \varepsilon_a \ll 1$ is a small constant to avoid numerical problems since the variance estimate might become very small or even zero.

This update rule is interpreted as, that the weight updates are not processed with each training example's individual gradient but with the average gradient of the recent history of samples. And further, the update is normalized by the local gradients' standard deviation. Hence if the gradient is *stable* and has little variance then the update is stronger. However when the gradient is noisy, i.e., when it has high variance, the weight change is weaker in order to stabilize the learning. Commonly used hyperparameter values for a stable optimization are given by the authors as $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

4.2 Unsupervised Learning

In this section, I will give an introduction to unsupervised learning describing the basic functioning and nature of problems that could be addressed with it.

Unsupervised learning is learning structure from data. It is about transforming the input stream into a more suitable representation in which the inherent structure of the data is in a certain way more accessible, for instance in terms of a lower dimension, less memory usage, or with a focus on the most important aspects. Possible applications of unsupervised learning techniques are, e.g., feature extraction pre-training, clustering, and dimensionality reduction. It is sometimes also referred to as predictive learning and used for time-series prediction or reconstruction.

In contrast to supervised learning, where we provide training examples comprising the input *and* the desired output, in unsupervised learning, there is only input data. Learning happens implicitly by trying to optimize an objective that does not depend on prepared desired outputs.

Unsupervised learning, therefore, allows for endless data sets, i.e., for being connected to continuous data streams. We can use unsupervised methods with a walking robot's sensor data streams for instance. In Figure 4.2 I tried to draw a qualitative comparison of how much data is available for different types of learning, that resembles Yann LeCun's «Cake Analogy» or «LeCake»[1]. Since unsupervised learning could be applied without the need to costly prepare training examples but instead applied to data streams more or less directly as raw data, the amount of data which is available is many times greater. The possibility to let the algorithm process data streams with unsupervised methods can help to reduce biases introduced in training data during manual data collection.

[1] The Cake Analogy was presented by Y.LeCun in an invited talk on predictive learning at NIPS 2016.

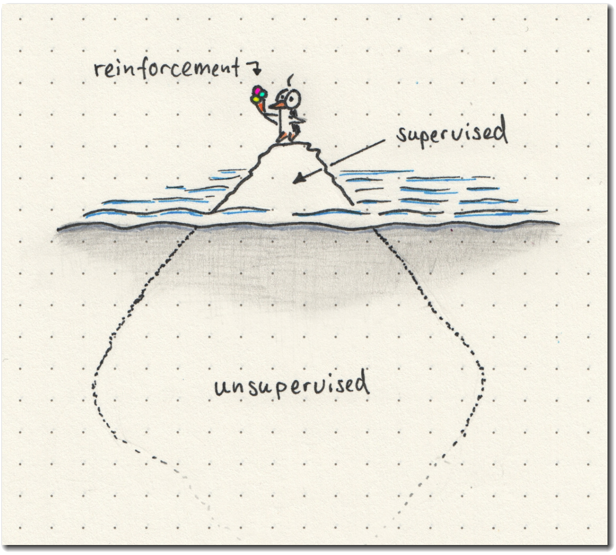


Figure 4.2: Analogy for the amount of data available for different types of learning represented by the volume of ice in this picture: Labeled training data created for supervised learning is just the visible tip of the iceberg. For reinforcement learning agents, feedback is even scarcer, occasionally even limited to a simple binary win or lose outcome at the end of a game. But the structure of the world is to be found in vast quantities of unlabeled data present in recordings or unending streams of robot sensor data.

Unsupervised learning algorithms usually rely on the principle of self-prediction. The algorithm makes a prediction $\hat{x}$ of the provided inputs x, using only those inputs to calculate the error signals in order to adjust the prediction. One can remark that predicting values which are available already is not very useful and, correct, this is in general not what we want to achieve since the best prediction would be just to copy the values. Instead, we introduce an intermediate representation h, a hidden layer in between so to say, and we actually use the input data to train a mapping $h \leftarrow x$ and also a mapping $\hat{x} \leftarrow h$. These functions are usually termed encoder and decoder.

Hence, we train the pair of encoder and decoder which forms the intermediate representation and decouple the prediction from using the inputs directly but instead trying to reconstruct the prediction from h. Our focus is now on this intermediate layer, depending on the data provided, this layer creates structure from the data and must find a hidden representation that is suitable to best reconstruct the prediction. One can think of searching for basis functions of an unknown system. In classification tasks, such basis functions are frequently termed *features*. Ideally, these basis functions or features should only have minimal overlap and represent the system in an optimal way. However, optimality can mean here, for instance, minimizing the number of features needed given an acceptable prediction error.

Another possibility for a hidden representation is *time*. We can attempt to predict future data samples from the present ones. Alternatively, we could forecast the most recent inputs from past values. For this, we build networks to store a lot of past data given an input data stream and use this as their basis functions. Frequently used networks for representing time are, for instance, tapped delay line multi-layer perceptrons or recurrent networks such as Long Short Term Memory (LSTM) by Hochreiter and Schmidhuber (1997).

Two important approaches for unsupervised learning shall be presented here since they illustrate the mechanics in a clear and tangible way. We start with the technique of enforcing a compact representation which is used by *autoencoder* neural networks. Secondly, we will discuss self-organizing networks with highlighting the *growing neural gas* algorithm, that is mainly driven by competition between subunits.

4.2.1 Compact Representation in Autoencoders

Autoencoder networks (Baldi and Hornik (1989); Hinton (1989)) comprise subnets for encoding and decoding. The encoding makes a transformation resulting in what is usually termed the *code* and which is to be reversed (or reconstructed) by the decoder in order to match the inputs again. One can say that the decoder tries to predict the inputs given only the code.

These kinds of networks usually come with a symmetrical structure comprising as many decode layers as there are encode layers. In the simplest case, there is only one hidden layer, which is then the code. A simple autoencoder with a single hidden layer can be written as:

$$\underbrace{\hat{\boldsymbol{x}} = g(\boldsymbol{h}, \boldsymbol{\theta}_2)}_{decoder} \qquad \underbrace{\boldsymbol{h} = f(\boldsymbol{x}, \boldsymbol{\theta}_1)}_{encoder} \tag{4.15}$$

with f being the encoder, g the decoder and $\boldsymbol{h}$ the intermediate or hidden representation. Autoencoders neural networks can for instance be trained to minimize the reconstruction error $E = ||\boldsymbol{x} - \hat{\boldsymbol{x}}||_2$ using gradient descent as defined above.

Since the hidden layers' dimension is usually larger than the input/output dimension, it happens that without any additional constraints, the network just learns the identity function, which should be avoided. But if we simply reduce the number of neurons in the most hidden layer we have built an artificial bottleneck. This bottleneck creates the compactness in the code representation and enforces the optimization to concentrate on the most important features.

When the dimension of hidden layers shall be larger than the input dimension, e.g., when using non-linear expansion, we need other constraints to avoid that the network just learns to copy the inputs. For example, we can constrain the weight matrix of the decoding part to be dependent on the encoding weight matrix in the way that

$$\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1^\top, \tag{4.16}$$

which is commonly referred to as *tied weights*. With this, we decrease both computation time and memory usage, since we have halved the number of weights to be learned and stored. Additionally, we also avoid imbalanced weights: Imagine using the introduced non-linear transfer functions $tanh(\cdot)$ with the center part of it being approximate linear. A pure linear reconstruction would require small weights in the encoder to keep the effects of the non-linearity low and larger weights, in the decoder to compensate for now smaller values. The tied weights restriction using the same matrix for both just transposed prevents this.

Autoencoders are also often utilized for pretraining in classification tasks to reduce the number of training examples in supervised learning. When plenty of unlabeled data is available but labeled data is rare, we can pre-train classification networks by training autoencoders and finally use only the encoder part to initialize a classification network. So, the network has already learned the basic features of the provided data, and it remains to learn how to classify them given a good set of features but with fewer labeled samples needed.

A special form of autoencoders are de-noising autoencoders where we present a noisy or distorted signal $\boldsymbol{x}+\boldsymbol{\xi}$ as input to the network but we train it to predict the original signal $\boldsymbol{x}$. The network then is trained to reconstruct the signal and hence to remove the noise or distortion.

But what makes the autoencoder unsupervised is the self-organization of the code neurons. They rearrange during training, trying to reconstruct the input in the best possible way. They learn the features of the input data independently of the task. They become experts for specific regimes of the input data. In this way, they do a form of data compression, having the code layer compactly representing the data.

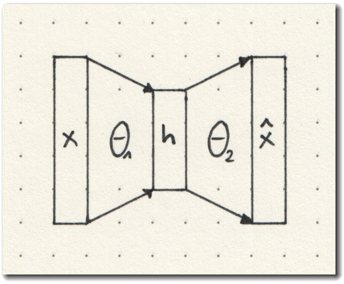


Figure 4.3: Autoencoder with a single hidden layer.

Relation to Dimensionality Reduction

When high dimensional data is to be visualized, we often encounter problems to focus on the relevant dimensions or areas of interest. Unsupervised learning algorithms are hence often used to reduce the dimensionality of the inputs to two or three dimensions in order to be able to use conventional cartesian plots or other common diagrams of low dimension. This mostly comes with the assumption, that even though the data dimension is high, the dimension of the underlying problem's manifold is not.

So if we transform the data appropriately we are able to visualize it in let us say three dimensions without losing too much of its structure. However, this is a lossy process which may hide some detail. So there is a broad evolution of techniques trying to solve this problem with quite a variety of approaches.

In the simplest and linear form, dimensionality reduction is done by principal component analysis (PCA) which rotates and scales the coordinate system in order to maximize the variance of the data. The result has the same dimension but their components are ordered by the variance, so the assumption is that, by truncating to the two or three dimensions with the highest variance, we could reduce the dimensionality to the most important dimensions. PCA can be in a way regarded as a linear transfer function autoencoder. Hence, autoencoders are a practical solution for the purpose of dimensionality reduction, especially when we need a computational efficient online (incremental) algorithm.

But there are shortcomings in precision compared to more sophisticated methods like *t-distributed Stochastic Neighbor Embedding* (t-SNE) as introduced by van der Maaten and Hinton (2008), where the samples' relations are taken into account to keep the most relevant neighborhoods close together. The approach creates probability distributions in high (original) and low (target) dimensional space and minimizes their relative entropy to have related samples also close in the reduced dimension.

4.2.2 Self-Organization, Competition, and Growing Neural Gas

Unsupervised learning is also referred to as self-organized learning, mostly in the context of neural networks that can grow the number of neurons and re-organize their connections according to an optimization criterion. A famous candidate algorithm for this technique is Growing Neural Gas that has been adopted in various ways and will be discussed later in this section.

So, this brings us to a second unsupervised learning approach: *competition*. We can employ competition between subunits following the winner-takes-all paradigm to encourage specialization. The main idea behind this approach is the assumption that the optimization problem naturally divides into easier subareas and that we can, therefore, also split the model into simpler but sufficiently many submodels. So, we create a competition between submodels for the subsequent data sample (concurrent hypotheses) and let the best-performing model win. The compound model will then cover all essential aspects. The goal still is prediction. We only allow the already best predicting submodel to adapt to the current sample. In the result, we get submodels that have specialized, that are *experts* for their niche.

The pattern to be implemented this is quite simple, let all experts make a prediction of the current sample, estimate and compare their prediction error and select the one with the minimum prediction error, adapt only this expert. The next time a similar sample is presented, the adapted expert has an even lower prediction error and the representation stabilizes. Specialization only works as expected when the submodel's «capacity» is low enough so that it is only capable of winning a fraction of the samples.

But why does competition lead to specialization? Under the winner-takes-all mechanics, units that are subject to competition must quickly find their niche because redundancies and overlapping competence of experts will decrease over time. When no redundancy can be removed anymore the process converges and results in a group of highly specialized units that cover more or less equally sized parts of the sample set. This specialization is advantageous for identifying the function of subunits and hence a probate means for introspection and a possibility for a better understanding of the underlying principle. Introspection is still one of the major drawbacks in many modern supervised methods, especially true in deep neural architectures. Their function is by design highly redundant and individual features are distributed over a vast number of neurons with overlap, which makes it hard to impossible to identify or remove a specific function and even worse to introspect why the network is doing what it is doing.

We often find lower dimensional systems covered in higher dimensional data. For instance, a video stream showing a pendulum swinging. The dynamics of the pendulum can be predicted to a good degree with the help of low dimensional equations covering position, velocity, and simple friction terms. However, if we just look at the raw pixel streams, the original system's low dimension is non-linearly distributed in lots of partially redundant input channels and are most likely also covered by noise. This picture holds for many real-world problems, where the underlying dynamics or the manifold of the data is of a far lower dimension than the input data. And our aim is to discover the important aspects and filter away everything unrelated.

Unsupervised methods help us finding interesting areas in data. Not only can we identify clusters in the data, but we are also enabled to reveal spatiotemporal dependencies between clusters if e.g., the samples from multiple clusters are often following the same cluster index sequence. Or we can decouple the data from the frequency of recurring samples, and highlight the rare ones. If we for instance track a legged robot, samples belonging to walking, running or standing will be frequently occurring, but exceptional states as tumbling or falling might occur more rarely.

As we have seen, unsupervised learning techniques also relate to data compression. When we are able to identify clusters in the data, we could for example only transmit the identified cluster index instead of the raw data. Alternatively, when it comes to state identification, we could just report that the robot has changed state from walking to running. We can use these techniques to transform hours of sensorimotor data into very compact representations, such as behavior graphs, or sensorimotor maps.

GNG-Algorithm

Self-Organizing Neural Networks allow for a successive construction of topological structures or *maps* that can represent the sensorimotor input and output space. The Growing Neural Gas (GNG) invented by Fritzke (1995) and improved as GNG-U by Fritzke (1997) is a specific technique which should be described here exemplarily for a family of algorithms that exploit the competition principle mentioned above.

In GNG, input spaces are modeled as probability densities and the algorithm places units (think of neurons) in the multi-dimensional input space so that they optimally cover the underlying density. Each unit represents a set of input data points. High-dimensional input data is thus mapped onto a smaller number of representatives. Thus the input space is discretized if we assign each data point to a single representative. During learning, the topological structure of the input data is worked out by inserting edges between adjacent units. These edges are created for units which have the smallest distance to the current data sample and can thus be regarded as *Hebbian* synapses.

Units are tuples of $(\boldsymbol{w}, E, U)$ comprising a weight vector $\boldsymbol{w}$ an accumulated error measure E, and a utility measure U. Input samples of a certain distribution are presented randomly to the network. For each input vector, the squared distance to all units is calculated and the unit whose weight vector is closest to the input vector is announced the *winner*. The winning unit is adapted in direction of the input vector, together with units of its immediate neighborhood which are defined over existing edges. So GNG uses a softer variant of the winner-takes-all-approach, since the local group is adapted towards the current sample as well, however with a smaller learning rate.

With each input sample presented an edge is created (or refreshed) between the winner and the runner-up. Edges have an age and recently refreshed edges rejuvenate and aging edges eventually die. The procedure starts with exactly two units and grows successively as new units are inserted on demand. This demand is located with the help of the accumulated error. New units are inserted between the unit with the highest error and its neighbor with the next to highest error. By inserting units in high-error-density

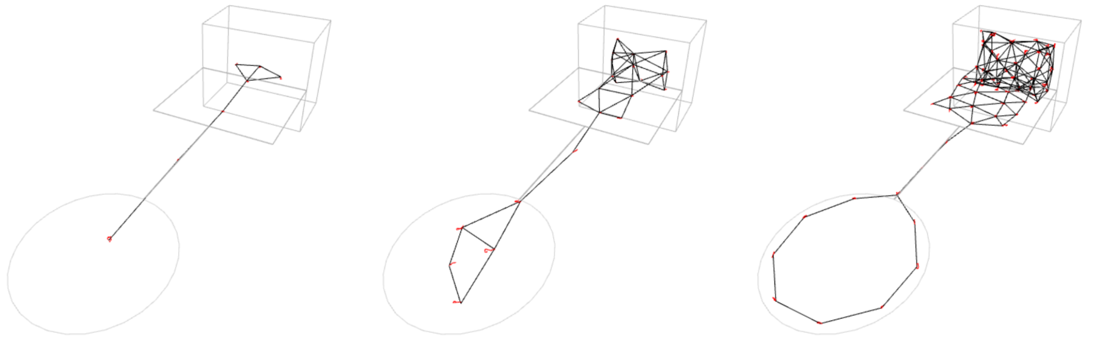


Figure 4.4: The figure showcases the Growing Neural Gas (GNG) algorithm by Fritzke (1995), depicting its initiation, progression, and convergence. The marked regions denote uniformly randomized sample inputs. The number of neurons is growing from 2 to 64. Edge connections indicate the neighborhood relationships between neurons. The GNG can model distributions of arbitrary dimension, with this illustration showcasing 1D, 2D, and 3D distributions combined. Standard hyperparameters from the original paper were used. Red traces show the units' recent paths.

locations the aim is to minimize the overall error of the system.

When having a non-stationary distribution of the input data, it is necessary to remove ineffective units and create new ones where they are more useful. This is necessary to cope with bounded resources in the general case of limited computation time and storage capacity. The utility measure of the Growing Neural Gas with Utility (GNG-U) helps us identifying useful units. The utility is measured as the accumulated local differences of the current error, so a unit is considered more useful when it is the winner and its runner-up would make a significantly larger error. Units are removed when all their edges aged out or the unit with lowest utility is removed when the ratio between the largest accumulated error and the lowest utility has reached a certain threshold.

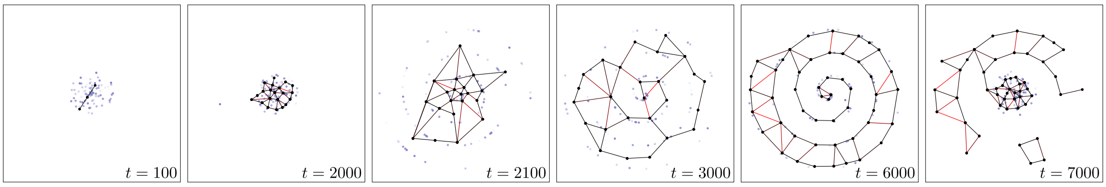


Figure 4.5: The Growing Neural Gas with Utility Criterion (GNG-U) algorithm by Fritzke (1997). The random distribution alternates twice between a small uniform circle and a spiral, which causes the network to adjust. Small dots represent individual samples from the distribution. Edges colored red are close to aging.

Limitations of GNG

The designated task of Growing Neural Gas is to model distributions. We must present input samples to the GNG randomly ordered to break temporal dependencies. In my application—robot locomotion learning—I am dealing with continuous dynamic processes that generate the input data online and whose samples highly correlate in time. We would have to store them in a sufficiently large replay buffer and provide them randomized. It would introduce a stochastic delay before the current sample is processed and before the network can adapt to non-stationary changes in the samples' distribution.

Also, what if we decide to use more elaborate prediction units? We can regard the GNG as a multi-expert method, with each expert estimating the average of a sample subset, cluster, or part of a trajectory. By incorporating more knowledge about our input space, we could build specific units that can predict and represent the spatio-temporal structure of the data. We could think of having a mix of differently typed features instead of a uniform set. Randomizing samples, as required by Growing Neural Gas, however, might bring us into trouble if our features exploit temporal dependencies.

Furthermore, GNG needs many hyperparameters, bringing difficulties in an already complex learning apparatus, such as a walking robot. With so many components, each having a set of parameters to tune, we would greatly benefit from reducing the hyperparameters' number and their overlap in meaning.

Edges in GNG mean connections between units that have a similar distance to the input sample, whether or not these samples can ever occur after each other. In dynamic systems like walking robots, we would profit from having edges carrying more details like probabilities of successive state transitions, putting temporal information on edges in addition to the spatial one.

Closing this, I want to remark that my attempt to apply Growing Neural Gas to sensor spaces of continuous systems in online learning setups resulted in sub-optimal results of network growth, and it was not easy to find suitable parameters. Fortunately, this led me to create a new GNG-inspired algorithm that better suits my target application. A comprehensive presentation and discussion of this algorithm follows later in Chapter 5.

4.2.3 Principle of Homeokinesis

In the last part of this section on unsupervised learning, we will shift our focus to the motor side. We will discuss an idea that tries to answer the questions: «Where do motor actions come from?» and «How can we enable the robot to create its own motor actions?»

Usually the actions that a machine takes are predefined by us humans, and our main concern is to think about letting the machine choose the right action in a given situation. However, our picture about motor actions could be partially wrong or incomplete. How can motor actions evolve purely from within the machine? These are questions that motivated the *Principle of Homeokinesis.*

When we do research on self-learning robots, we look for the robots' inherent modes of behavior. We may already be a bit tired of manually programming behavior into the machine when we have understood that all the work that goes into it is not yet sustainable. Today's robotic systems are becoming more and more complex. The hardware is becoming ever more affordable and available to everyone. But the time and knowledge required to program useful things into robots in the classical way does not yet match the availability of robotic hardware and the speed of prototyping and manufacturing. For systems with many degrees of freedom, the creation of meaningful robot movements is not yet fully thought out and definitely not trivial.

Self-learning or self-exploration means finding and recognizing one's own characteristics. What are the modes of a specific robotic systems? What types of movement are inherent to the machine, that can be learned easily by the machine in a reasonable time and with low energy consumption? Dynamical systems generally have resonance properties. A rod pendulum, for instance, as one of the simplest robotic systems, can be driven by a motor located in the joint and can already move in many different ways. However, the energy it would take to force an arbitrary movement on the pendulum can be enormous. On the other hand, the energy is minimal when the pendulum is swung up at its resonant frequency. These movements, the resonants, the modes, the inherent types of movement, they are elementary building blocks—motor primitives, so to say—from which more complex behaviors can be derived later. These behaviors are strongly coupled with the robot's individual shape, mass distribution, materials, controller dynamics, sensor placement, and so on. For each new robotic system, these properties can be very different. So we are searching for general methods with little assumptions about the systems to which they are applied.

A General Drive for Activity

Homeokinesis according to Der and Martius (2012); Der et al. (1999) is a promising approach for identifying the robot's very own movements. Despite using gradient descent, homeokinesis is an unsupervised process since it only uses data the system generates itself and makes surprisingly few assumptions about the robot's topology. It can therefore be used for a large number of systems. The authors have already demonstrated its performance for a variety of differently shaped robots (please refer to Martius et al. (2008); Der et al. (2005)).

Homoekinesis may be better understood as an exploration strategy, as it is not a learning process in the classic sense that follows a user-defined objective function. Neither is it expected to converge to a stable solution. The objective, if one wants to formulate it that way at all, is much more implicit. Rather than following gradients of external reward or fitness functions, follow instead local gradients of the system's dynamics to meanfully maximize activity. The homeokinetic system continuously minimizes an inner measure called the *Time-Loop-Error* which drives it to activity and exploration. In that sense it is far more effective than random exploration. Also controller parameters are expected to intentionally change quite rapidly so the learning rules for the controller together with the robot form a combined dynamical system.

The Inner Mechanics of Homeokinesis

The principle of homeokinesis is implemented as a complementary system of a model and a controller. Based only on sensor data, $\boldsymbol{x}_t$, the controller, C, generates continuous motor values, $\boldsymbol{y}_t$, applied to the robot. The motor outputs also form the basis for an estimate of the future sensor values, $\hat{\boldsymbol{x}}_{t+1}$, calculated by the forward model M. These estimates are compared with the values that are read by the sensors in the next time step, $\boldsymbol{x}_{t+1}$, and the difference between them forms the *prediction error*, E_{Pr}. The model is then learned in order to minimize this error over time.

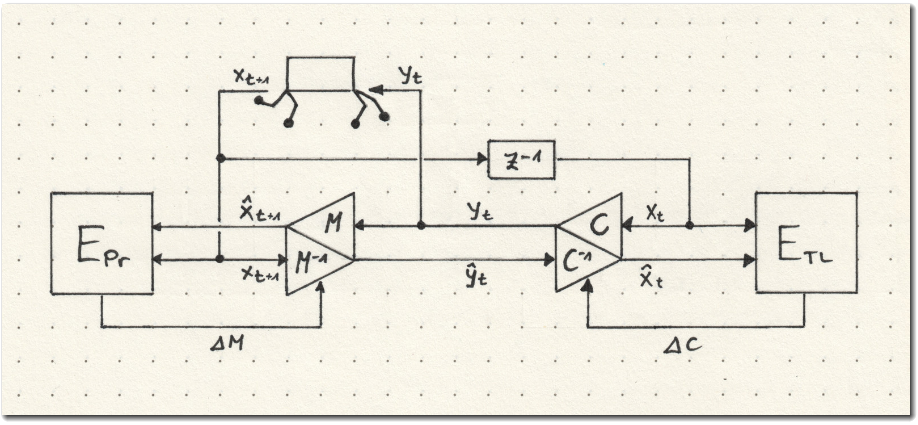


Figure 4.6: Illustration of the principle of homeokinesis. The robot is part of the closed sensorimotor loop $\boldsymbol{y}_t = C(\boldsymbol{x}_t)$. The model, M, estimates the next sensor state given the control output, $\hat{\boldsymbol{x}}_{t+1} = M(\boldsymbol{y}_t)$. The prediction error E_{Pr} is used to adjust the model of the robot and the controller weights are adjusted to minimize the time-loop-error E_{TL} for which the model and the controller need to be inverted to achieve the virtual state, $\hat{\boldsymbol{x}}_t$ of the robot.

On the other side, to adapt the controller, a reconstruction of the previous sensor values, $\hat{\boldsymbol{x}}_{t-1}$, is calculated with the help of an inversion of the model, M^{-1}, and the controller, C^{-1}, respectively. The reconstruction compared to the actual values forms the time loop error, E_{TL}, for which the controller is now modified to minimize it. Figure 4.6 summarizes the homeokinetic system.

When the model and the controller are both adapting they are changing each others situation. When the E_{TL} is minimized, i.e., the projection into the past is stabilized to be better predictable, it leads to a destabilization into the future. The inversion of the model and controller to reconstruct the past leads to leaving of precisely this state in the future. This is based on the property of dynamical systems that a system's attractor backward in time is a repellor forward in time (refer to Figure 4.7). However, by predicting the future, the system is destabilized in a gentle way that remains sufficiently predictable by the model.

Through their adaptation, both, the model and the controller form two opponents who depend on each other. One could note that a form *competition principle* as we learned for other unsupervised methods is also effective here to achieve a common goal. This principle is often exploited in machine learning, as for instance in Actor-Critic methods or Generative Adversarial Networks. Alternatively, we see it as the common search for the compromise between a driving force, in this case the very sensitive controller, and a regulating force, the prediction model, which demands that the actions remain predictable to a certain extent.

Limitations

Homeokinesis drives a robot for continuous activity. So there remains the need for homeostatic behaviour, too. Homeokinesis avoids stabilizing situations. However, these form a major part of usefull beaviors such as balance recovery. Maybe accompanied by the more

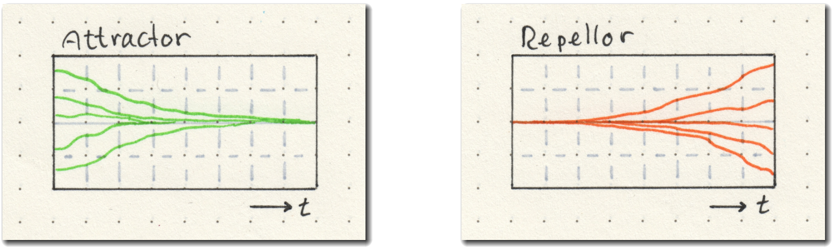


Figure 4.7: Attractor (left), repellor (right). When time is inverted, attractors change into repellors and vice versa. For instance, stable fixed points turn into unstable ones. Hence, stabilizing the controller dynamics in the past leads to destabilizing them in the future. The Homeokinesis exploits this to drive the robot to activity.

homeostatic character of CSL behavioral modes we are coming closer to a fundamental set of motor base functions.

When implementing homeokinesis on real-existing robots, just as with CSL, a number of measures must be taken to protect the machine. The homeokinesis is primarily an exploration mechanism and initially has no built-in protection functions. The controller is persistently destabilized. However, the resulting motions do not care for the limits of the robot. Similar to other controllers, e.g., Position-PID or CSL, the protection of the components against overtemperature or mechnical overload is not covered within the pure theory and has to be implemented additionally.

Continuous use of the motors usually causes them to heat up. The thermal losses, caused by the voltage drop at the ohmic elements of the motor and the H-bridge increase quadratically with the current, $P = I^2R$. In particular, very powerful movements or load changes heat up the motors, so that the robot has to take regular breaks.

Please note that homeokinesis uses servo target positions as motor outputs $\boldsymbol{y}_t$, so we need a PID position controller at the output and cannot apply motor voltages directly here as before. By using target positions, the motor is already quite sensitive to changes around a controller output close to zero. This must be taken into account. Because, in contrast, when motor voltages are applied around zero, nothing happens below static friction.

Closing Remarks

Homeokinesis is a valuable approach for robot exploration and a method for stimulating activity, regardless of the type of robot being used. It effectively investigates the dynamic modes of a robot that are neither excessively predictable nor completely unpredictable. This method fully integrates the physical capabilities of the robot, creating a closed loop between the sensors and the motor actions. Homeokinesis has the ability to complement unsupervised techniques for motor control, and it is anticipated to facilitate exploration to discover the fundamental locomotion modes of the robot more rapidly than evolutionary methods or by following reward function gradients.

4.3 Reinforcement Learning

«Why did the reinforcement learning agent cross the road?
To maximize its expected reward!» (by ChatGPT)

Reinforcement learning (RL) is a subfield of machine learning that focuses on training agents to make decisions based on feedback from their environment. In RL, an agent learns by interacting with an environment and receiving feedback in the form of numerical rewards for its actions. RL has applications in a wide range of fields, including robotics, game playing, and optimization. In this section on fundamentals, I will introduce only the methods that I use. My aim is to provide the reader with a good understanding of the technique, rather than being formally complete. As RL has become a vast family of algorithms, a comprehensive overview is provided by the books of Sutton and Barto (1998, 2018), in which all terms are formally defined and theoretically grounded.

Here is the outline of this chapter: First, I will introduce the reader to the basic terms and concepts of reinforcement learning. Then, we will delve into Temporal Difference (TD) Learning with emphasis on its model-free on-policy form, Sarsa. I will explore Boltzmann-Softargmax action selection as a means to solve the exploration vs. exploitation trade-off and discuss its practical implementation. Finally, I will focus on the application of RL in the context of robot locomotion learning.

The Reinforcement Learning Problem

RL is based on learning through trial and error, where an agent interacts with an environment. The agent performs actions, and the environment responds with new state information and a reward signal. The RL model is depicted in Figure 4.8. The agent does not necessarily have to be the robot but is rather the learning submodule. This implies that the state information s can be any derived or higher-level status values, or even outputs of other components of the learning system, and the actions a may not necessarily mean motoric actions but could also be used to trigger changes in hyperparameters of other subsystems. The term environment describes everything that is external to the

RL agent including the robot's body dynamics. To differentiate the symbols from our notation so far, I use the RL-specific notation for states, s, and actions, a. The time index of states and actions is mostly omitted, with the next state denoted as $s(t+1) = s'$ and action $a(t+1) = a'$.

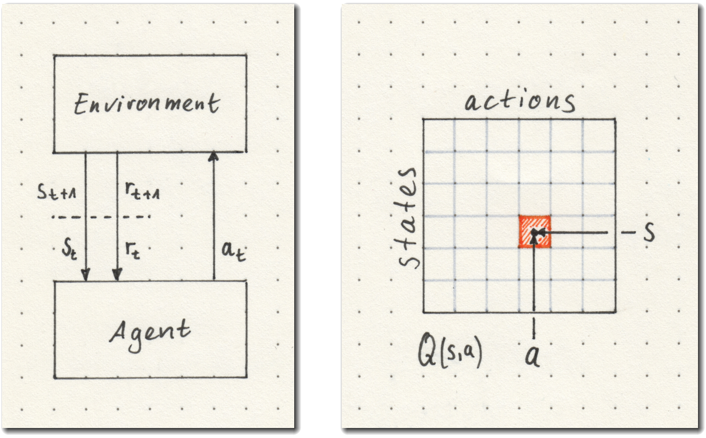


Figure 4.8: The reinforcement learning model (left), figure derived from Sutton and Barto (1998). Selecting an action a according to the current state s in discrete reinforcement learning (right). Q-values, $Q(s,a)$, have a tabular form when states and actions are discrete sets.

States and Actions

The agent's states and actions can either be discrete or continuous. In discrete models, we receive a certain unique state s from a set $\mathcal{S}$ of possible states and select an action a from a set $\mathcal{A}$ of possible actions. In the continuous case we receive a real-valued state vector $\boldsymbol{s} \in \mathcal{S} \subseteq \mathbb{R}^N$ as input and write a real-valued action vector $\boldsymbol{a} \in \mathcal{A} \subseteq \mathbb{R}^M$ as output. Nonetheless, hybrid forms, e.g., continuous state, discrete actions are also widely used. Without loss of generality, we will further assume that each action $a \in \mathcal{A}$ can be performed in each state.

Reward

To determine what action is the best to be executed in each state, we must utilize a problem-specific cost function known as the reward, R. The result of an action in a particular state is generally unknown in advance. So we have to try out different actions to eventually know what they do and how reproducible their outcome is. After performing actions the agent receives reward and the result of an action may be a new state of the system. However, the reward function is also unknown, and furthermore for stochastic systems, we generally get a distribution of rewards for every state transition and action. So the agent must make estimates of the expected rewards.

Markov Assumption

Part of the theoretical basis of reinforcement learning are *Markov Decision Processes* (MDP). Using reinforcement learning, we start from the Markov assumption that all relevant information is contained in the agent's state. Everything depends solely on the current state only, and not on previous states. An MDP is defined as the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ with states $\mathcal{S}$, actions $\mathcal{A}$, rewards $\mathcal{R}$, and $\mathcal{T} = T_a(s, s')$ the stochastic state-transition model, which tells us how likely it is to be in successor state s' given the current state s and performing the action a. For most real robots $\mathcal{T}$ is unknown and must be modelled or approximated. The reward function $\mathcal{R} = R_a(s, s')$ is human-defined specifically for the task, but the agent has only access to samples $r = R_a(s, s')$ of this function when actions are actually performed. Nonetheless, experience samples in the form of (s, a, s', r) which were already acquired can be saved in a buffer for later usage.

In real-world robotics, usually, we have no immediate knowledge about the overall state of the world but perceive their environment by means of sensors. So here, the assumption is explicitly made that the world behaves *markov*, but the robot can only observe the world partially through its sensors. This means that the actual state of the world is hidden to the robot and it can only approximate a model of it.

Value Function

The reward function, R, is used to approximate the state-value function, $V(s)$. By accumulating rewards for each state, the agent learns which states are more desirable. If a state transition model exists, the effects of actions can be estimated by propagating the transition model using the value function, V.

If for our robot a transition model is not given, we can instead of V approximate the action-value function $Q(s, a)$, which assigns a value to each *pair* of corresponding state and action. With discrete states and actions, this function has simply the tabular form as depicted in Figure 4.8. To identify the best action, we have to iterate over all available actions of the current state and find the one with the highest *Q-value*.

Policy

Selecting optimal actions based on the estimated Q-values is the task of the policy π. In general the policy is stochastic and non-stationary, because all entries in the value function are only estimates, and these are mostly wrong in the beginning of learning. Therefore the agent must try a lot and the estimated will change necessarily. Two common strategies for selecting meaningful actions from a discrete Q function are discussed later in subsection 4.3.2.

Remark: The intermediate step of approximating a value function can also be bypassed by optimizing the policy directly, such as in *policy gradient* methods. We can achieve this by estimating the gradient of the expected reward w.r.t. the policy parameters, and using the gradient to update the policy in the direction that maximizes the expected reward.

Learning Rate and Discounting

The learning rate $\alpha \in (0, 1]$ is a hyperparameter that can be adjusted to influence the speed of the agent's learning. However, this presents the stability-plasticity dilemma, where we must balance the rate of acquiring new knowledge with the rate of forgetting old knowledge. The learning rate can also be state-dependent or decrease over time.

In reinforcement learning, the agent's *return* is the infinite sequence of rewards received, $r_{t+1} + r_{t+2} + \dots$, which grows indefinitely for non-episodic tasks. This brings us to the idea of *discounting*, which determines the weight given to potential future rewards versus immediate rewards. A higher weight on immediate rewards leads to more conservative behavior, while a higher weight on future rewards results in more farsighted behavior. Discounting comes as the second RL hyperparameter $\gamma \in (0, 1)$ with $0 \ll \gamma < 1$

and the agent's expected *discounted* return is defined as

$$G_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{4.17}$$

so that the formal problem definition of estimating value functions $V(s)$ or $Q(s,a)$ while following a policy π and using the expected value $\mathbb{E}(\cdot)$ becomes

$$V_\pi(s) = \mathbb{E}\left[G_t | s_t = s\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right] \tag{4.18}$$

$$Q_\pi(s,a) = \mathbb{E}\left[G_t | s_t = s, a_t = a\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right] \tag{4.19}$$

4.3.1 Temporal Difference Learning and SARSA

Temporal Difference Learning is a type of reinforcement learning algorithm that updates the value function of an agent after every action, rather than waiting until the end of an episode. This update is based on an estimate of the expected reward associated with the current state and the next state. TD-Learning does not rely on a complete sequence of actions, but rather on the difference between the expected reward obtained from two consecutive samples. The TD-Learning update rule as described in Sutton and Barto (1998) in its usual form is

$$V(s) \leftarrow V(s) + \alpha\Big(r + \gamma V(s') - V(s)\Big) \tag{4.20}$$

with learning rate α and discount factor γ. We can rearrange the update rule to get the equivalent but in my view more instructive form

$$V(s) \leftarrow \overbrace{(1-\alpha)V(s)}^{fading\ memory} + \underbrace{\alpha\Big(r + \gamma V(s')\Big)}_{new\ value} \tag{4.21}$$

where we can clearly separate the parts of old fading memory which is kept with factor $(1-\alpha)$ and the new value $r+\gamma V(s')$ to be updated with α. In this form we see the update rule's averaging character (to calculate the expected value $\mathbb{E}(\cdot)$) since it is identical to a simple IIR low-pass filter. The *Temporal Difference Error* usually denoted as

$$\delta_V = \underbrace{r + \gamma V(s')}_{new} - \overbrace{V(s)}^{old} \tag{4.22}$$

signals if the last update was better or worse than the current estimate and hence positive δ_V means positive feedback for the agent.

SARSA

The action value equivalent to TD-Learning is Sarsa, an on-policy TD control learning method. The name derives from the form of the tuple used for updating which is (s, a, r, s', a'). This update rule is structurally equivalent to Equation 4.21 and reads the same way with the only difference that we update individual cells from a tabular function $Q(s, a)$, a value for every state and action pair. The Sarsa update rule for is defined as

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma Q(s',a')) \tag{4.23}$$

or in its original form with TD-Error

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_Q \qquad \delta_Q = r + \gamma Q(s',a') - Q(s,a) \tag{4.24}$$

The Equation 4.23 is to be interpreted as: When we are in state s and we take action a, we then receive a reward of r and get the next state s'. Following our policy, we select the next action a' and update the value for $Q(s, a)$.

Sarsa is an *on-policy* algorithm which means, we update the policy that is currently followed. This distinguishes Sarsa from Q-Learning, which is an off-policy procedure and allows one policy to be followed while learning another policy. Learning with Sarsa is *model free* since it does not require a state-transition model. Rather the transition model is implicitly approximated in Q.

Eligibility Traces

A common extension to TD methods are *eligibility traces* which allow for incorporating the contribution of previous state-action-pairs and paying them back their share of the currently received reward. The method tries to solve the *credit assignment problem* and keeps track of the system's recent states and actions taken and assigns them an exponentially decaying share of the reward. We tag each currently updated state-action-pair with an eligibility value $e(s, a) = 1$, called *resetting traces*, and iterate over all states and actions $\boldsymbol{e} \leftarrow \gamma\lambda\boldsymbol{e}$ to fade out their values over time using the decay parameter $0 \leq \lambda < 1$. Using eligibility traces the method is denoted as Sarsa(λ).

Experience Replay

To speed up learning and weaken undesired temporal correlations between samples, we can use *experience replay* by reusing samples for multiple update steps instead of learning each one only once. The agent's experience is recorded as a set of samples (s, a, s', r). For each step and update, the most recent sample is added to an experience buffer and a minibatch is usually sampled from it for updates. However, for real-world problems with large state and action spaces and stochastic rewards, it is impractical to keep all samples, so we need a strategy to decide which to keep and which to drop. Simple solutions include using a fixed-sized FIFO buffer or randomly replacing samples. More sophisticated methods evaluate each sample to determine its novelty and optimal complement to existing samples in the buffer. The size of the replay buffer should be adequate for the specific problem to avoid slowing down learning. The rate at which the non-stationary system changes can determine the necessary size of the buffer.

4.3.2 Self-Regulated Action Selection

When using Sarsa it is necessary to select actions based on the approximated action-value function. So by trial and error we gained estimates on how «beneficial» the actions are for the agent. We could let the agent always pick the one with the maximal estimate, but how could we be for sure that these estimates are true? Especially in the beginning of learning, the estimates are mostly wrong. So the agent must try out several actions, even if they do not appear to be the best ones at first glance. The trade-off in action selection is also referred to as the *Exploration-Exploitation-Dilemma.* Two common solutions of this problem are presented here, and I propose a useful extension of the second one afterward.

Epsilon Greedy

The probably most widely used method for selecting actions from a given set of Q-values is the *Epsilon Greedy* action selection for which we define an exploration rate $\epsilon \in \mathbb{R}$ with $0 < \epsilon \leq 1$. With the probability of $1 - \epsilon$ we pick the currently most promising, so-called *greedy* action

$$k_q = \underset{i \in [1,M]}{\arg\max}\, q_i \tag{4.25}$$

i.e., the action k_q with the highest action value available in $\boldsymbol{q} = (q_0, q_1, \dots,)^\top \in Q(s, \cdot)$ given the current state s. Every other action is selected with a non-zero probability of $\epsilon/(M-1)$. When $\epsilon \to 1$ we increase the exploration rate of non-greedy actions to the maximum, while $\epsilon \to 0$ will only exploit the currently most promising candidate for the *best action.* With $\epsilon = 0.1$ we usually have a good starting point for further adjustment of this hyperparameter. If we aim for convergence of stationary problems, we usually use a large exploration rate in the beginning and decrease it over the trials. For non-stationary problems and open-ended learning tasks, we must maintain a non-zero probability of exploring the non-greedy actions to keep the agent adaptive to changes.

Epsilon Greedy can be implemented easily and computational efficient with $\mathcal{O}(M)$. However it has one major drawback: All non-greedy actions are selected with the same probability independent of their action values. The method makes no difference between the second to best action (with a probably good value) or the worst one, already disclosed to bring no reward at all. Thus, we will now discuss an alternative action selection method that finds a better balance between exploration and exploitation and preserves the relative differences between the estimated action values so far.

Softargmax

The *softargmax*[2] or *Boltzmann* action selection utilizes the properties of the exponential function to map the vector of Q-values, $\boldsymbol{q} \in \mathbb{R}^M$ to a vector of probabilities $\boldsymbol{p} \in [0,1]^M$, i.e., a discrete probability distribution which has the same dimension M and all elements bounded to $0 \leq p_i \leq 1$ and with $\sum_{i=0}^{M-1} p_i = 1$. The softargmax function for selecting

[2] After Goodfellow et al. (2016), pp. 183–184, the conventional name softmax is misleading and we better use the term Soft*arg*max instead. The corresponding soft version of a pure *maximum* function is defined $\mathrm{softmax}(\boldsymbol{z}) = \mathrm{softargmax}(\boldsymbol{z})^\top \boldsymbol{z}$.

actions from given Q-values $\boldsymbol{q}$ is defined as

$$p_i = \mathrm{s}(\boldsymbol{q})_i = \mathrm{softargmax}(\boldsymbol{q})_i = \frac{e^{q_i/\tau}}{\sum_k e^{q_k/\tau}} \tag{4.26}$$

with $p_i = P(a_i|s)$ and $q_i = Q(s, a_i)$. The hyperparameter τ is called the *temperature* and adjusts the «greediness» in selecting actions. With $\tau \to \infty$ the selection is completely random, while $\tau \to 0$ will favor only the greedy action as illustrated in Figure 4.9.

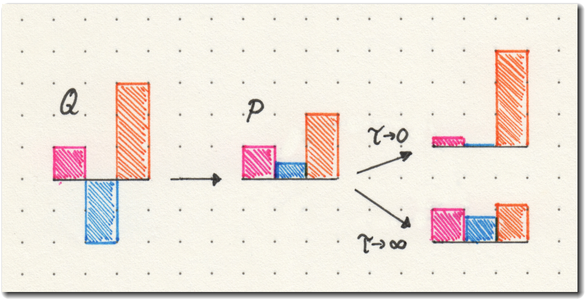


Figure 4.9: Softargmax action selection: arbitrary Q-values are transformed into probabilities and can be either regulated towards a greedy selection with $\tau \to 0$ or towards a uniform distribution with $\tau \to \infty$ making the selection merely random.

Exponential functions with large arguments can easily result in numerical overflow problems. Exponents of 10^2 already to break the limits of a single-precision 32 bit floating point variable while exponents of 10^3 could not even be represented anymore by a common 64 bit double-precision float with a maximum of $1.79769 \cdot 10^{308}$. So we commonly employ a trick to circumvent floating point overflows and transform them into less problematic underflows. Adding constant terms in all exponents of the softargmax

$$\frac{e^{x_i+c}}{\sum_k e^{x_k+c}} = \frac{e^c}{e^c}\frac{e^{x_i}}{\sum_k e^{x_k}} = \frac{e^{x_i}}{\sum_k e^{x_k}} \tag{4.27}$$

does not change the result. So we «normalize» our vector $\boldsymbol{q}$ by element-wise subtracting the maximum Q-value before passing it through the softargmax function. Using the tempered Q-values $\tilde{\boldsymbol{q}} = \boldsymbol{q} - \max(\boldsymbol{q})$ the arithmetics are save from overflows.

Self-Adjusting Exploration Rate

With the simpler Epsilon Greedy method we had an exploration rate hyperparameter ϵ in a well-behaved interval that was independent of the underlying Q-values. With softargmax we now have a more sensible parameter, the temperature τ. If the statistics of the Q-values are unknown or change during learning, it is complicated to correctly adjust τ beforehand. For non-stationary problems a fixed value of τ will be even harder to find and is probably inadequately set for the entire experiment. Due to the steepness of the exponentials the useful interval can get quite narrow and we might end up in situations where softargmax action selection starts degenerating and slowing down the learning success. The selections are then either too random with excessive exploration or too greedy with insufficient exploration. For non-stationary problems softargmax selection thus only makes sense insofar as the temperature τ is regulated.

So when I found that the window for good values of τ that allows for a balance between exploration and exploitation can get small and might be moving, I developed an extension to softargmax that helped me to regulate the temperature in a homeostatic way as published in Kubisch et al. (2010). The idea I came up with is to regulate the variance of the probability vector, $\boldsymbol{p}$, in form of a self-regulating update rule as defined in Equation 4.28. If the Q-values are almost uniformly distributed, the variance is small. On the other hand, if only few values poke out, the variance is high. Possible values for the (unbiased) variance $v = \mathrm{Var}(\boldsymbol{p})$ are limited to the interval $[0, \frac{1}{M}]$. A regulation towards the target value $v^* = \frac{1}{2M}$ will keep the variance balanced between the two extremes.

Let $\beta_s = 1/\tau$ be the *inverse self-regulated temperature.* We can apply a simple homeostatic learning rule each time we update $\boldsymbol{p}$

$$\beta_s \leftarrow \beta_s \cdot \left(\frac{3}{2} - vM\right) \qquad v = \frac{1}{M-1}\sum_{i=0}^{M-1}(p_i - \frac{1}{M})^2 \tag{4.28}$$

The product vM can only reach values from the interval $[0, 1]$. Thus β can be halved or increased by half at most. The regulation converges within a few steps when $v \to v^*$. The learning rule is unmistakably homeostatic when regulating the variance of action probabilities $\boldsymbol{p}$ towards v^*, but in its effect it is also forcing the system to amplify the differences in action values and breaks symmetries. As noted by Doya (2002), using regulated β as output of another process, the randomness in action selection can be controlled to regulate the robot behavior according to specific situations. In Figure 4.10 I show a simplified experimental run conducted to illustrate the mechanism of the β-regulated softargmax.

4.3.3 RL in the Context of Locomotion Learning

Closing the section about reinforcement learning I want to make suggestions on how the setting for walking robots and their task of locomotion learning can be phrased in the context of the methods described here.

Episodic vs. Continuous Learning

Episodic learning means we always have a starting and a terminal state, usually with high rewards at the end and multiple penalties on the path to encourage minimizing its length or duration. However, episodic learning can sometimes be challenging to transfer into the real world because it requires defined initial states, similar to evolutionary experiments. Frequent resets require additional effort and might be impractical. For example, training real running robots on a grass court has high efforts to bring the robot back to a defined location and body position after each attempt.

On the other hand, continuous learning struggles with the effective formulation of reward functions and sometimes creates local optima, from which the agent might be incapable to escape. This defect results, among others, from the frequent updates of the Q-function while simultaneously used for the followed policy.

But continuous learning naturally separates into quasi-episodes of independent learning trials. If an episode usually ends with reaching its terminal state, we start the next episode. For this, the terminal state of the previous episode must be equal to the starting state of the next episode in the new trial. If we design complementary goals, it might be

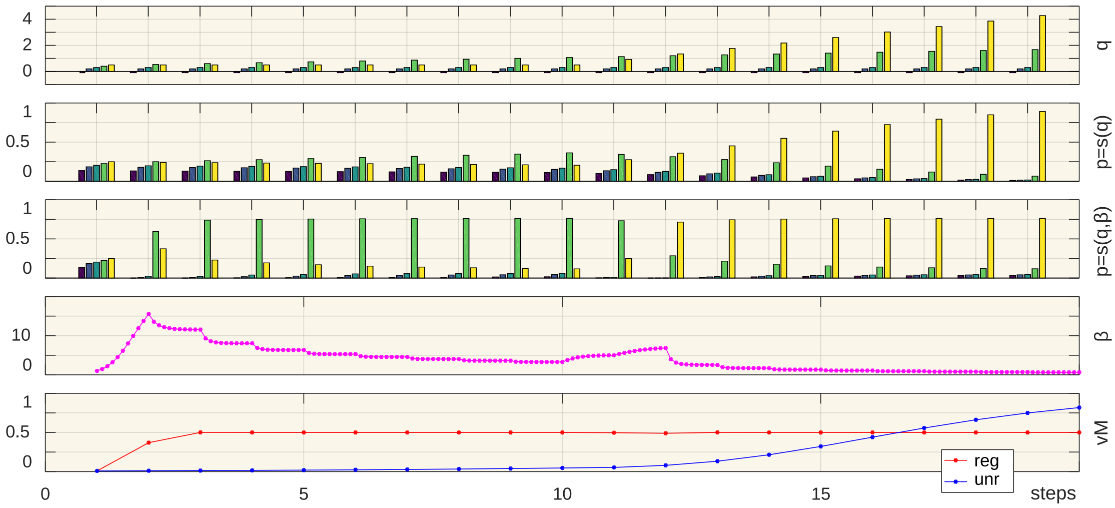


Figure 4.10: Experimental test run of the β-regulated softargmax: The system's Q-values start from small initial random values, the green action grows slowly since the beginning, and the yellow action starts later at $t > 10$ but grows faster. Both softargmax and β-regulated softargmax start suboptimally with $\beta_0 = 1$ from almost uniform action probabilities with their variances being very low. As soon as β-regulated softargmax starts the self-regulation, the variance grows and stabilizes at around the target value. As a result of regulation, the green action gets amplified at first. After the yellow action's Q-value outgrows it, a switch-over takes place for the system to now favor the yellow action. Finally, β regulates downwards to keep the target variance, even when the yellow action's Q-value continues to grow. Meanwhile, the unregulated softargmax is suboptimally setup with a constant value $\beta = \beta_0$. It is either too uniformly distributed with the variance too low or greedy only with the variance too high. As can be seen, to keep a target variance, β needs to make a progression from 15 times above its initial value b_0 to even slightly below it.

feasible to connect the order of goals appropriately, for instance, a robot standing up, running forward, stopping, walking backward, stopping, and eventually sitting down. Probably, it is not necessary to always reach the goal. Especially in the beginning of learning, achieving goals is rare. It might suffice to switch targets after a defined trial time.

So to improve open-ended learning in robotics, I moved away from episode-based algorithms. Instead, I allow the robot to practice for a specific duration, irrespective of achieving the specified goal state and subsequently switch to follow another policy. The task for the robot is to follow different policies that are sufficiently complementary, and I switch between them at random. I assume that different goals will positively influence each other due to the RL agent perceiving variations in its starting positions for the subsequent policy. I expect this to make the results more robust by providing the agent more data on the transients between behaviors.

Learning Multiple Goals Simultaneously

What if there are several reward functions that all simultaneously pay out their reward? The actions performed in a particular state then generate more than one scalar reward value, but a reward vector $\boldsymbol{r} = (r_0, r_1, \dots)^\top$ in which each element represents a different learning objective, e.g., different directions to run.

Let us assume that the robot is following a policy π_i that is designed to move forward. Now, if we consider another policy π_j where $i \neq j$, which may involve sideways-running, then the greedy actions of π_i may appear to be mostly exploratory actions that follow a non-greedy policy. However, the learning samples collected during forward walking can also help to improve the value function for sideways running, albeit indirectly.

Furthermore, tracking different policies results in more variance for initial states for each other policy. This means that each trial begins with the given position, regardless of what it is, as the starting point for the next attempt. Therefore, we avoid performing clean resets when switching trials, which offers the chance for more robust behavior.

By changing the policy frequently and ensuring that each learning goal is given enough time to train, a well-balanced mix of exploratory and greedy actions can be achieved. In cases where one of the learning goals is intrinsically motivated learning (as described in Section section 4.4), traditional exploration can be mostly eliminated except for a minor noise component to prevent unintended symmetries. Since the system already has enough exploration for each policy that comes from following other policies in between, additional exploration may not be necessary.

Reward Functions for Walking Robots

In Section section 3.4, I discussed fitness functions for walking robots. However, to use these functions in reinforcement learning, we need to convert them into reward functions. In evolution, the fitness value is given at the end of a trial. Similarly, in reinforcement learning, it makes sense to think in terms of learning trials where the robot practices a task for a while, receives a reward, and then moves on to the next task. Although the learning process continues, it can be broken down into natural episodes as discussed in previous section, and an accumulated reward can be given at the end of each episode. However, this also means that the update of the behavior only takes place at the end of each episode, and not necessarily the fastest method for learning.

Since we want to make steps towards methods for learning on the real robot platform, we need techniques that learn fast, robust, and directly from sensor data. So what we are looking for is reward functions that efficiently use the robot's sensor values, that need no external (human controlled) reward, and provide good feedback on a sample basis. But deriving reward values from raw sensor data potentially brings a lot of noise into the learning system, which we must handle.

When I optimized directional walking for speed, the fitness function was calculated by dividing the distance covered by a fixed trial time, resulting effectively in a velocity measurement. This method worked well in simulations, where distance could be easily measured. However, with walking robots, measuring the distance covered is not as straightforward. One possibility is to use supervised methods to train an odometry module to estimate the walked distance. Since our goal is to develop methods applicable to real robots, we need to rely on measurements that can be derived directly from the robot's sensors. Otherwise, relying on external rewards, such as measuring distance from camera observations, would require additional effort and may not be feasible.

The robot's built-in acceleration sensors can be used inexpensively to measure the robot's speed. To estimate the speed, we numerically integrate the sensor values, which are measured at the center of the robot's body. This integration provides an instantaneous or averaged speed per time step, used as a continuous reward signal, denoted as $r_v(t) = \bar{v}(t)$. We calculate the robot's average velocity, denoted by the scalar $\bar{v}$, using a low-pass integrator $\bar{v} \leftarrow b\bar{a} + (1-b)\bar{v}$, where $\bar{a} = \sum_{i=0}^{2} c_i a_i$ is the acceleration calculated as a linear combination of the acceleration sensor's individual axes a_i. The vector $\boldsymbol{c}$ sets the preferred direction, and the coefficient $b \in (0,1)$ adjusts the filter and determines the extent of rewarding instantaneous or averaged velocity.

But how to transfer our efficiency measure? In evolution experiments, we used another class of fitness function that accumulated the squared values of the robot's motor outputs, $\boldsymbol{u}$, and terminated the trial, when a fixed maximal sum was reached. This led to energy-efficient behaviors. We can approximate this by

$$r_u(t) = \frac{\bar{v}(t)}{1 + ||\boldsymbol{u}||_2^2} \tag{4.29}$$

which is the ratio of cause (force/torque) and effect (velocity), thus a simple estimate for efficiency.

Given a set of reward functions derived from sensor data, we can now attempt to train a robot for directed locomotion. But whose goals are this? It is ours! In the next section, we will delve into reward functions inherent to the agent and open a view of a completely different type of robot reward.

4.4 Intrinsic Motivation

In higher developed species, individuals show explorative behavior that produces new sensory perceptions in a systematic way. Playfully they improve their skills. Exploration and play is thus generally regarded as central aspects for successfully coping with life and its constantly changing requirements. In reinforcement learning, exploration typically involves random actions. However, when using softmax action selection, exploration is not entirely random but is biased towards more rewarding actions. Nonetheless, exploration is conventionally based on pseudo-random values and human-crafted reward functions. But exploration plays a pivotal role in RL methods, primarily and most importantly to fill the value function with credible reward estimates. However, purely random exploration can get stuck in attractors, as demonstrated by the underpowered pendulum scenario, where swinging back and forth multiple times is necessary for it to swing up. With random actions alone, the likelihood of reaching the upper position and receiving the reward is considerably low.

But we might already have a method to avoid pure randomness in exploration. As we have already seen with homeokinesis, the inner measurements of a robot can be a source of activity. Let us assume we can derive another source of motivation directly from the system's internal state, which is changing sufficiently often and systematically to create enough intended exploratory actions for random actions to become unnecessary.

Exploratory individuals actively dive into situations they are unfamiliar with. They possess a source of *curiosity* or *inner drive* to explore their environment. Random exploration is undirected and might even slow down learning when acting randomly in already well-explored situations. But we can now apply a method to systematically locate states where the agent needs to explore and focus on visiting them more likely.

Figure 4.11: Animals that play around.

Intrinsic motivation refers to the inherent drive and curiosity to engage in activities for the sake of personal enjoyment, learning, or mastery, rather than for external rewards or pressures. This type of motivation arises from within an individual and can lead to higher levels of engagement, creativity, and persistence. Intrinsic motivation has been studied extensively in psychology and is becoming increasingly relevant in the development of reinforcement learning algorithms and autonomous agents that can learn and explore their environments autonomously.

A promising formal implementation of intrinsic motivation is to estimate the learning progress achieved by the individual and employs it as a reward to select future actions. This reward is self-generated by the learner and therefore purely intrinsic. The theory behind is intensively discussed in Schmidhuber (2006); Oudeyer and Kaplan (2008) and Schmidhuber (2010) and reinforced with empirical data of real robot experiments conducted in Oudeyer et al. (2007) and Kaplan and Oudeyer (2007). When the individual is capable to measure its own learning progress, i.e., has a certain ability of introspection, and if it relates this progress to the current sensorimotor situation, it can seek out similar situations to which it has *learned to learn successfully.*

The method of intrinsic motivation with rewards derived from the learning progress of the agent, can replace pure random exploration in RL settings, as I examined in simplified experiments conducted in previous work (Kubisch et al. (2010); Kubisch (2017)). In low-dimensional dynamical systems I could show, that intrinsically motivated exploration is superior to pure random exploration, in a way that it enabled the agent to systematically escape fixed points in state space and not only explores but even regularly revisits different domains in state space.

Definition of Intrinsic Reward

Intrinsic motivation, in a nutshell, can be defined as self-rewarding one's own learning progress. To make this tangible, we need to define what it means to make progress in learning. Generally, in an agent, there are several things to be adapted, self-adjusted, optimized or learned. Whatever we call it, there is always *before* and *after* the adaptation step. Some inputs were processed and then some weights were changed to be more adapted than before. If we manage to quantify the outcome of the adaptation, i.e., if we can measure how much better adapted is the agent now, we can consider this measure as learning progress.

A popular definition of learning progress is the negative time derivate of the *prediction error*. The agent is constantly predicting the future sensory inputs with the help of a model, thus predicting the environmental changes and the outcomes of its own actions. The quality of that prediction can be verified by comparing the predicted sensory values $\hat{\boldsymbol{x}}$ with the measured values $\boldsymbol{x}$. So we can define the prediction error as

$$E = ||\boldsymbol{x} - \hat{\boldsymbol{x}}||_2 = \sum_i (x_i - \hat{x}_i)^2 \tag{4.30}$$

which is lower when the system's predictions got better. The prediction error can be measured on different time scales: instantaneous, averaged, or long-term, which is creating different meanings. For each new system finding the right time scale to measure the learning progress may need some manual tuning.

What is now important is how the prediction error evolves over time. If the system is actually learning something, then the error in prediction should decrease. A positive rewards is thus a decrease in prediction error, i.e.,

$$r_\star = -\dot{E} \tag{4.31}$$

where $\dot{E} \; = dE/dt$ is the error's derivative w.r.t. time. Of course, learning progress can be zero too, when the prediction was perfect, and nothing got learned. Or it could even be temporally negative when we trained with stochastic gradient descent. A simple differentiator as depicted in Figure 4.12 will work for most cases. However if a low-noise smooth differentiation filter is needed, refer to Hamming (1989) and Selesnick (2006).

Since there are other introspective sources of learning progress as well, this definition should not exclude the possibility that other intrinsic rewards can be formulated similarly. For instance, in growing structures like GNG the insertion of units is mostly done in reaction to some internal limit or threshold, signaling a demand for more capacity, which is an implicit reaction to some new stimuli from reaching new sensorimotor domains. Thus, rewarding the growth of the model means rewarding the learning progress caused by exploring the limits of what the robot knows so far.

In other words, if the individual would identify states and corresponding actions that generate learning progress, it could compare the return and decide for actions that maximize that progress in learning. Especially in systems with growing state and action spaces, this implies a growth in the agent's sensorimotor capabilities. Hence, maximizing learning progress by intrinsic rewards can be considered another form of *empowerment* in the sense of «Act to increase your options» (Klyubin et al. (2008)), i.e., that more potential options are generated of what the agent can in principle do or perceive. For walking robot we can say: Everything else being equal, robots should seek higher mobility.

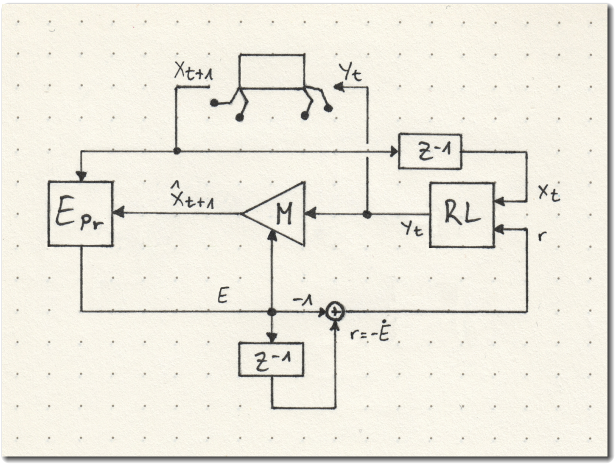


Figure 4.12: Possible structure for an intrinsically motivated robot: The model generates predictions of future sensor data, which, when compared to the actual sensor data, result in the prediction error. This error signal is used both to improve the model and to generate a reward signal being the negative time derivative of the error. Given a model that learns successfully, the error is expected to decrease over time. The reward then reinforces the actions that helped improving the model and hence further reduce the prediction error.

Exploratory behaviors guided by intrinsic motivation is thus a basic building block for task-independent learning. Exploration generates, by chance, new situations and sensory stimuli while intrinsic motivation sets the direction.

Learning progress as a reward creates, in any case, highly exploratory behavior and leads to continuous changes in the respective value function. Let us assume the agent performs an action in a certain state which yields a specific intrinsic reward $r_\star(t)$. If the agent is again in this state and performs the same action again, the reward must be on average less than before when learning was successful. So the reward function changes over time w.r.t. the states visited and actions performed. This is different from goal-directed learning where the reward function is usually unchanged (even though it might be stochastic).

So generally, due to its non-stationary nature, the policy which is based on intrinsic rewards will change during learning and it is not likely to gracefully converge. However, we probably can follow the intrinsic policy greedily without the need for frequently executing non-greedy actions, since the greedily selected actions are per definition exploratory and will change as soon as the learning progress fades for the current greedy action.

Closing Remark

Let me close this section with a thought experiment. When thinking about reinforcement learning as a possible model of animal learning, we eventually come to the question: «Where do all the rewards come from?» Here we saw that rewards can come from intrinsic measurements only but in the majority of cases rewards are defined externally by humans. But is it possible that a robot can be rewarded for directional walking from the same intrinsic motivation? Consider an agent having an incomplete map of its environment, where its state is the global position, orientation, and velocity in that map. There would be the need for exploring this internal representation of the world and mapping the new findings. Improving this world model (in addition to the self-model) creates learning progress, too. So there is actually a motivation to get somewhere, to push the edges of the world, and to add new states. Hence, if the robot is not able to walk in the beginning, the reward it would get for adding new landmarks in the world model must imperatively reinforce the development of its locomotion.

Conclusion and Discussion

In this chapter, I introduced the fundamental building blocks for the conducted learning experiments to follow and the developed algorithm in the next chapter. Supervised and unsupervised techniques, reinforcement learning, and intrinsic motivation are the elements I will put together in the context of self-organized robot locomotion learning.

The conclusion I can draw so far from this chapter is that the available classes of learning algorithms only reflect partial aspects of what is supposed to become a fully independent and self-learning machine, that is, the goal of epigenetic robotics. However, a corresponding superstructure is missing so far. At present, we compensated for this with a modular framework. We will see whether a generalization of the available algorithms will be possible in the future, with each so far known approach being a special case. Alternatively, we see whether a kind of meta-learning algebra will emerge in which the algorithms now used are represented by symbols and operators and with which we can formulate completely new learning architectures.

5 A Growing Multi-Expert Structure

In this chapter, I describe in detail the workings and implementation of an algorithm which I call the *Growing Multi-Expert Structure* (GMES) as a solution to the task of unsupervised segmentation of continuous sensory state spaces (Kubisch (2017)). I adapted this algorithm for open-ended learning tasks on systems with limited computational resources, allowing robots to learn how to move autonomously. The algorithm converts a high-dimensional continuous state space into a collection of sub-models, known as *experts*, each representing a distinct region of the state space. These experts can function as discrete states, with only one active at a time, or as adaptable basis functions, each with its individual activity level. This state representation is particularly advantageous for reinforcement learning control tasks.

The algorithm continuously processes sensory inputs and handles raw, non-stationary sequential data. The number of experts increases as needed to accommodate the complexity and inherent structure of the underlying data. The algorithm is well-suited for online learning with limited computational resources, as it has low computational requirements. Furthermore, we only need to select three intuitive hyperparameters without requiring explicit domain knowledge of the sensory input space manifold.

The structure's growth is modulated with the measured learning progress and allows for quite harsh context switches and hence does not suffer from so-called *catastrophic forgetting*. Its modularity is particularly noteworthy, as we can select the method used for the experts' prediction modules depending on the problem.

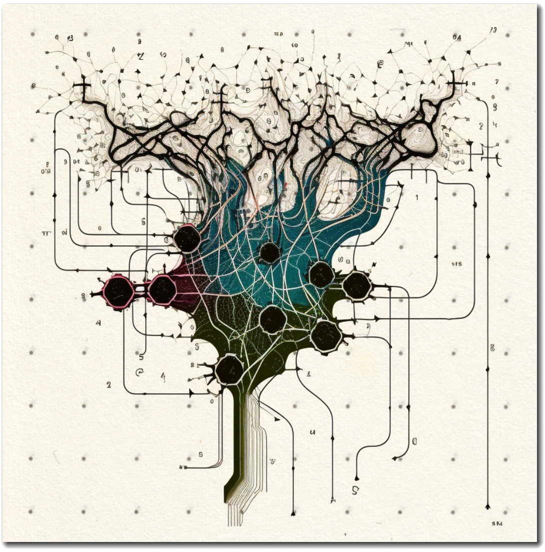


Figure 5.1: Art illustration of a growing multi-expert structure: Sensor inputs (top) get compressed into state-representing neurons (middle) and provide state outputs (bottom) for subsequent learning modules.

Introduction

Inspired by the stunning adaptivity of living beings and their learning capabilities, I generally believe that the artificial systems we want to build need to create their model of body and environment on their own and adapt it continuously during their *lifetime*. Ideally, they start building their world model from scratch, referred to as the *tabula* *rasa* situation, i.e., from as little prior information as possible. Unsupervised learning algorithms that take the data directly from experience and exploration seem to me to be most qualified to create and maintain such *world models* or *sensorimotor maps* as exemplarily well illustrated in the inspiring work of Toussaint (2006).

Classic reinforcement learning algorithms rely on the concept of discrete states when they fill tabular value functions. Their continuous counterparts need carefully crafted basis functions when using value function approximation with neural networks and gradient descent. However, the question of how to create these states or *features* in a self-organizing way, automatically and straight from raw sensor data, remains partially unanswered. Modern approaches skip the intermediate step of building value functions and instead learn the policies end-to-end directly from the data using deep neural networks such as in Schulman et al. (2017). Despite their popularity and effectiveness, these approaches often suffer from requiring many training examples and presently seem not suited for online learning on bounded computational resources. Also, end-to-end learning highly cloaks the inner mechanism and creates black-box systems that are hard to interpret.

In low-dimensional systems with only a few sensors, features can be manually defined, even using the inefficient method of partitioning the state space into equally sized and uniformly distributed segments. But for higher dimensional systems, this approach does not scale anymore. Hand-crafting an appropriately sized state space representation of a four-legged robot needs a lot of domain knowledge and will likely not be optimal.

Handcrafted state spaces can present several issues, including the presence of too many indifferent states that can slow down learning. Partitioning the state space into equally sized segments is inefficient in two ways and is subject to the *curse of dimensionality*. First, the underlying state manifold for many real-world systems does not necessarily fill the complete space, resulting in numerous unvisited states that may waste memory or processing power. Second, a uniform distribution of states may lead to problems, as areas of the state space may differ in their relative importance with respect to the system's performance. We may require more fine-grained control strategies in some areas of the state space, while other larger areas could be represented by a single state with a large receptive field. Domain knowledge is necessary to address these issues, which can be challenging to obtain for large-scale state spaces.

But the most crucial issue with handcrafted state spaces is that they are static. Robots change, be it due to payload, maintenance, or wear. The robot's behavior changes during learning, from uncoordinated waving to complex motions. So a state space that is not constructed statically but can adapt to the current needs and changes will be advantageous in speeding up learning and reducing the need for manual adjustments.

The subsequent sections particularize an unsupervised learning algorithm that leverages competitive Hebbian learning and gradient descent. This algorithm segments continuous data into discrete states using a collection of competing prediction modules known as *experts*. The algorithm is based on the Growing Neural Gas with Utility Criterion (GNG-U) introduced by Fritzke (1997) and follows the *winner-takes-all* paradigm. The resultant structure constitutes an expert ensemble or a network where nodes and edges represent states and transitions. It can grow on demand and can follow non-stationary data distributions. It inserts experts one after another as learning progresses until a pre-defined maximum number. After that, less valuable units get relocated to more advantageous positions in the state space. Refer to Figure 5.2 for a first glimpse.

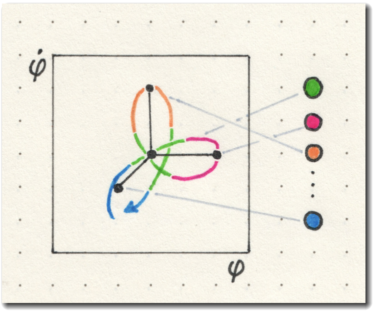


Figure 5.2: Simplified visualization of a state space segmentation (or discretization) using a Growing Multi-Expert Structure, Experts, and Predictions: The diagram shows the state space of a single robot joint comprising angle φ and angular velocity $\dot{\varphi}$. The nodes in the graph, called experts, represent the states of the system, whereas edges denote state transitions. The colored samples help identify the part of the trajectory that belongs to the closest (i.e., best predicting) expert.

The described algorithm runs continuously and learns incrementally. There is explicitly no notion of a training nor a terminal condition. Nonetheless, it converges to a locally optimal solution when processing stationary data. In any case, the structure keeps being adaptive. When the underlying manifold changes, the structure will note quickly and, if needed, will utilize its available experts to adapt to the new conditions. The presented Growing Multi-Expert Structure can alternatively be viewed as a self-organizing processing layer connecting continuous input streams to systems that build upon sets of discrete states or basis functions.

Let me now detailedly describe the algorithm and its parametrization. Subsequently, its functioning is first demonstrated on simple examples and afterward applied to the more application-oriented and particularly higher-dimensional sensor data of the robots presented in the previous parts of this thesis.

5.1 Algorithm Details

Let us define the sensory state space consisting of $D \in \mathbb{N}^+$ input channels describing the state manifold $\mathcal{X} \subseteq \mathbb{R}^D$. This manifold $\mathcal{X}$ usually covers only a subset of $\mathbb{R}^D$ since, on real systems like robots, not all positions in this space can be reached. We can think of specific robot limb positions that would collide with the trunk or other limbs. Sensory inputs of the current time step $t \in \mathbb{N}$ form the state vector $\boldsymbol{x}(t) \in \mathcal{X}$ which we call the *stimulus*. Raw sensory data is processed directly and sequentially. Without loss of generality, we normalize and limit each sensory input channel to the interval $[-1, +1]$.

Experts, Adaptation, and State Estimation

The structure consists of maximal $N \in \mathbb{N}^+$ *experts* competing for the best prediction of the next stimulus. The expert with the lowest prediction error is called the *winner* and is the only one allowed to adapt. This principle is known as winner-takes-all (WTA), with one winning expert per time step. We denote the index of the winner as k. WTA drives the group of experts towards specialization to get exclusive detectors for different system states. However, specialization only emerges if the data has more complexity

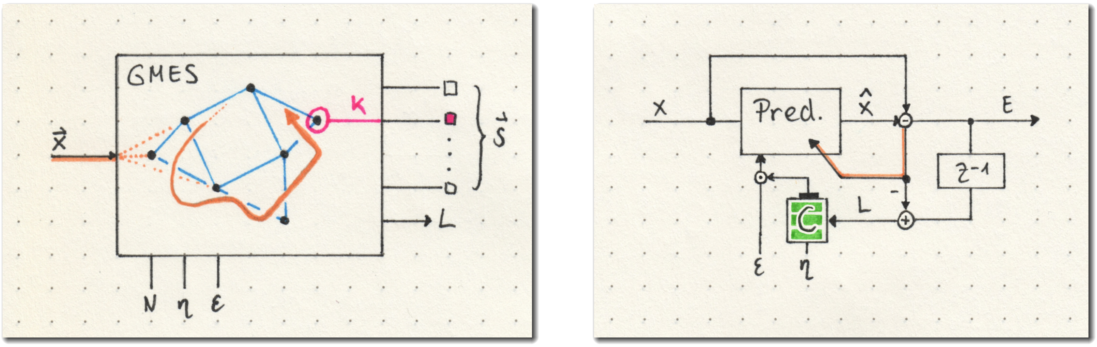


Figure 5.3: Schematics of GMES on the left: Sensor data gets compressed into a set of states, $s_0, s_1, \ldots$ while emitting learning progress, L. Schematics of a single expert on the right: The prediction error improves the predictor and is derived w.r.t. time modulating the learning capacity, $\mathcal{C}$ (denoted with a battery symbol). A too-low learning capacity halts any modification to the predictor.

than a single expert can cover. Otherwise, when a single expert can approximate too much, there is no need to incorporate other units.

Each expert $i \in [1..N]$ computes a prediction $\hat{\boldsymbol{x}}^i \in \mathbb{R}^D$ of the current stimulus $\boldsymbol{x}$ given past sensory data, $\hat{\boldsymbol{x}}^i(t) = \text{pred}(\boldsymbol{x}(t-1), \boldsymbol{x}(t-2), \ldots)$. Such a prediction module can have any suitable form. However, I prefer neural networks that adapt via gradient descent as introduced in subsection 4.1.1 and that I will specifically describe later in the section.

The prediction error $E^i \in \mathbb{R}_0^+$ for expert i is defined as $E^i = ||\boldsymbol{x} - \hat{\boldsymbol{x}}^i||^2$ as introduced in Equation 4.30, i.e., the sum of squared prediction errors for each input channel. To decrease its prediction error for future samples, each expert can change the weights of its prediction module. If the module is allowed to adapt, the weights are changed towards a better prediction using the *expert learning rate* $\epsilon \in \mathbb{R}$, a hyperparameter that must be selected manually depending on the system's properties and the type of prediction module. The learning rate is held constant and equal for all experts. The weights of the experts' prediction modules get updated proportionally to the prediction error.

Expert $k = \arg\min_{i \in [1,N]} E^i$ with the best prediction is called the winner. Only the winning expert is allowed to adapt to the stimulus, amplifying specialization. The others stay unchanged for the current time step. Therefore GMES is termed a *competitive* learning algorithm. Simultaneously, the winner defines the current system's state, k. So it works as a state detector, indicator, or receptive field. The state information is a possible interface to discrete reinforcement learning methods.

Learning Progress

We defined the learning progress as the decrease in prediction error with $L = -\dot{E}$ in section 4.4. Thus GMES can be used as a source of reward for intrinsically motivated learning systems. Since we deal with a discrete-time WTA system, the learning progress of the whole structure is generated by the winning expert and is approximated by

$$L \approx E^k - E^k_*, \tag{5.1}$$

which is the difference between the prediction error before and after adaptation. The measured prediction error regarding the current stimulus after adaptation must be lower than before, so for GMES learning progress is always $L \geq 0$.

Learning Capacity

The workings described so far we commonly find in self-organizing systems. The unique feature of GMES is that experts carry a property called *Learning Capacity* that is denoted as $\mathcal{C}^i$ and indicates the amount of allowed adaption per expert i. Subsequently, after each adaption step, the learning capacity of the winning expert is decreased with respect to the measured learning progress. Experts with exhausted learning capacity will not get adapted further.

Learning capacity gets initialized with a value of 1 for each expert. To keep the sum of all experts' available learning capacity constant, every consumption of learning capacity gets immediately redistributed to the group of N experts, allowing for a focus shift of learning to other areas while preserving the overall adaptivity. Learning capacity can be used as an indicator, too. A low learning capacity implicitly expresses the experts' relative utility, whereas a high capacity reveals untapped potential. Also, it indicates the need for further adaptation and can guide learning systematically.

Every adaptation decreases the learning capacity of the winner in proportion to its learning progress by

$$\mathcal{C}^k \leftarrow \mathcal{C}^k \, e^{-\eta L} \tag{5.2}$$

where $\eta \in \mathbb{R}^+$ is called the *structure growth rate.* The consumed learning capacity $\Delta\mathcal{C}^k$ gets redistributed to the group by randomly selecting a recipient.

Prediction Methods, Time-Embedding, and Non-linear Expansion

A prediction module takes the stimulus as input and outputs the prediction of the next stimulus as a vector of the same size. GMES is agnostic to the specific type of predictor used. We can perform simple predictions with only a moving average or use sophisticated neural networks. The simplest way to predict a time series is by computing an average over past samples, assuming that the signal will have roughly the same value as our average in the next time step. For our moving average predictor we have use a constant weight vector $\hat{\boldsymbol{x}}^i(t) = \boldsymbol{w}^i$ and learn the weights by

$$\Delta\boldsymbol{w}^i = \epsilon\,(\boldsymbol{x} - \hat{\boldsymbol{x}}^i) \tag{5.3}$$

$$\boldsymbol{w}^i \leftarrow \boldsymbol{w}^i + \Delta\boldsymbol{w}^i \tag{5.4}$$

with expert learning rate $0 < \epsilon \ll 1$. This can be an effective method for simple systems mainly having states of almost constant sensor inputs.

A more sophisticated method for time-series prediction is using time-delay multi-layer perceptrons. Their architecture is better suited for sensor signals emitted by our walking robots since these signals usually change more frequently. The network's inputs get shifted through tapped delay lines that delay each sample by one time step. Such delay line inputs can be seen as FIR filter synapses and compute a linear time expansion of the input signal. These tappings now constitute a higher dimensional input and are fed

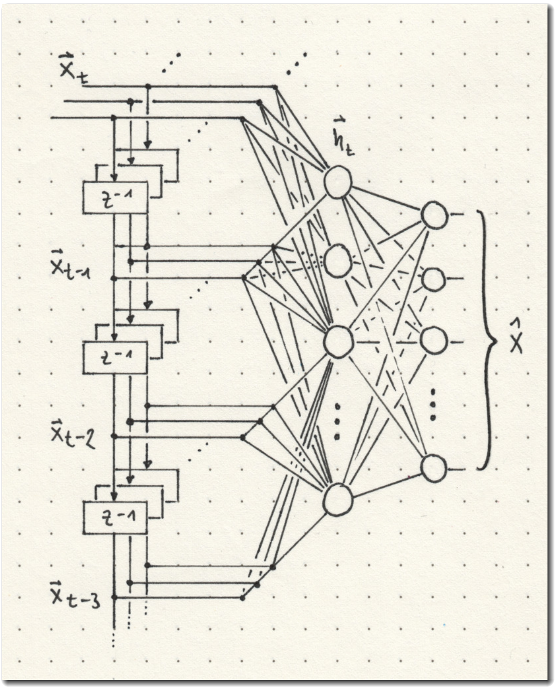


Figure 5.4: Schematic of a Time-Delay Neural Network (TDNN) with tapped delay lines and hidden layer: The network's inputs get delayed to provide taps, denoted as z^{-1}, to the past samples of the time series. The hidden layer works as a non-linear stage that either expands or compresses the information flow dependent on the number of hidden neurons. The output stage for the network as predictor must have the same dimension as the input. Since such a TDNN is qualitatively a network with FIR filter synapses, each input sample passes through a tapped delay line, so the experts can prefilter their inputs, e.g., smooth them (low-pass), react to sudden input changes (high pass) or delay them.

into a neural hidden layer, which is non-linear. The hidden neurons form a simple way of either non-linear expansion or compression, depending on the number of hidden neurons. Finally, the output layer serves as the prediction signal, having the same dimension as the input. Figure 5.4 shows a schematic for the Tapped-Delay-Line MLP predictor. The training of such a network is usually done with backpropagation as described in section 4.1. Whatever method we use for the predictors, the vector $\boldsymbol{w}$ represents its weights or parameters.

State Transitions

In a network, an edge connects two nodes. Here, an edge represents a directed state transition from one expert to another. So for each pair of experts, there exists a tuple of *linkages* $\{\mathcal{T}^{ji}, \mathcal{T}^{ij}\} \in [0, 1]$. Transitions get refreshed in the sense of Hebbian learning—*'Cells that fire together wire together'*, that is, the transition of subsequently winning experts i and j is validated ($\mathcal{T}^{ji} \leftarrow 1$) whereas learning progress on the other side causes the invalidation of all transitions of the winner k. Their linkages are losing guarantee after an adaptation that moved the expert towards a new stimulus when a transition happened some time ago. The invalidation is calculated by

$$\mathcal{T}^{ki} \leftarrow \mathcal{T}^{ki}\, e^{-\eta L} \tag{5.5}$$

$$\mathcal{T}^{ik} \leftarrow \mathcal{T}^{ik}\, e^{-\eta L}, \ \forall i. \tag{5.6}$$

As we can see, for state transitions, GMES uses the same decay factor $e^{-\eta L}$ that is bounded between $[0, 1]$ and gets modulated by the learning progress L and the structure growth rate η.

Growth

Learning begins with a single expert, and its prediction module's weights get initialized with appropriate initial values. For the simple moving average prediction, this is the first stimulus. When the learning capacity of the winning expert is exhausted ($\mathcal{C}^k < 1\%$), a new expert r gets inserted *before* adaptation. For this purpose, we assign r the weights and the current prediction error of k. The new expert then gets connected with the winning one through a valid transition $\mathcal{T}^{rk} \leftarrow 1$, and both share the sum of their learning capacity in equal parts. From now, the new expert is considered the winner and is allowed to adapt to the stimulus.

Alternatively, we can speed up learning if we directly initialize $\boldsymbol{w}^r$ to $\boldsymbol{x}(t)$. Let us refer to this as *one-shot learning* since an expert gets instantaneously trained to predict perfectly. Especially when using a small expert learning rate this is an immense improvement. However, for more complicated prediction techniques one-shot learning is probably impractical or impossible.

Reorganization

Considering further adaptation, we relocate less effective experts when the maximum number of available experts N exceeds. We select the expert with the highest learning capacity

$$r = \underset{i \in [1,N]}{\arg\max}\, \mathcal{C}^i \tag{5.7}$$

for relocation and its present transitions get removed. Also, the relocated expert is treated like a newly inserted expert, receiving a copy of the winner's weights, its error, and a share of its learning capacity.

Algorithm Overview

0. Start with a single expert, initialize $\boldsymbol{w}^0$ appropriately.
1. Get next stimulus $\boldsymbol{x}(t)$.
2. Let all experts calculate their predictions $\hat{\boldsymbol{x}}^i(t)$ and get their prediction errors $E^i(t)$ according to Eq. (4.30).
3. Find the expert k with the lowest prediction error. This expert k is the winner and defines the current discrete state of the system.
4. When the learning capacity of the winning expert, $\mathcal{C}^k$, is exhausted, insert a new expert r. If all N experts are already in use, select the expert with the maximum learning capacity as r. The new or relocated expert r inherits the weights and error of k. Delete all present transitions of r. Create a new transition $\mathcal{T}^{rk} \leftarrow 1$. Share their learning capacity. Declare the new expert as the winner, $k \leftarrow r$.
5. Adapt the weights of the winning expert $\boldsymbol{w}^k$ towards lower prediction error.
6. Estimate the learning progress of k according to Eq. (5.1) and decrease its learning capacity by $\Delta\mathcal{C}^k$, Eq. (5.2). Redistribute $\Delta\mathcal{C}^k$ to the other $N-1$ experts in order to preserve the system's total learning capacity.
7. Decrease all linkages of k to its adjacent experts, according to Eq. (5.5) and (5.6). Refresh the transition from previous to the current winner.

8. Continue with step 1.

Parameters

For GMES to work, we need to manually select only a few hyperparameters: the maximum number of experts N, the expert learning rate ϵ, and the structure growth rate η. In Table 5.1, we find practical value ranges and approximate presets. The preset values should be treated only as a solid starting point since the precise hyperparameter choice depends on the application.

symbol	meaning	range	preset
N	maximum number of experts	$N \in \mathbb{N}^+$	
η	structure growth rate	$\eta \in \mathbb{R}^+$	1
ϵ	expert learning rate	$\epsilon \in \mathbb{R}^+,\ \epsilon \ll 1$	0.01

Table 5.1: Hyperparameters for the growing multi-expert structure.

Complexity

I developed GMES with a focus on applications that have limited computational resources, such as autonomous legged robots. In such applications, processing power and memory capabilities are typically restricted due to limited space, weight, and battery power. The computational time of the proposed algorithm increases with $\mathcal{O}(N)$. State transitions are an optional feature as they do not directly influence the state space segmentation. With transitions, memory usage scales with $\mathcal{O}(N^2)$ at worst, since usually the transition matrix is sparse. So squared memory usage is rather an edge case. Without transitions, memory usage scales linearly with $\mathcal{O}(N)$.

A Brief Example: The Flower Manifold

The simulation described below illustrates the basic functionality of GMES. Let us set up a low dimensional dynamical system that produces sequential time-discrete data $\boldsymbol{x}(t) \in [-1, 1]$ in the form of a flower-like curve. The data is non-stationary as the flower slowly rotates around the origin and because the system is switched at time step $t = 10^5$ from the flower-like shape to a circle (refer to Figure 5.5). GMES uses moving average prediction with one-shot learning. Initial weights of the beginning expert are $\boldsymbol{w}^0 = (0, 0)^\top$. The hyperparameters used are $N = 42$, $\eta = 50$, $\epsilon = 0.05$.

Initially, the number of experts is rapidly increasing, but the insertion of new experts during learning is becoming less common. As the system drastically changes, new experts get inserted due to increased learning progress. The learning capacity remains constant within the structure but accumulates in unused experts. When all experts are finally employed, only truly redundant experts keep piling up learning capacity.

Output Representation

The outputs of GMES are beneficial in different ways. GMES processes the input vector, $\boldsymbol{x}$, of length D into the error vector $\boldsymbol{E} = (E^0, E^1, \ldots, E^{N-1})^\top$ of length N. It additionally outputs a scalar index, $k \in [1..N]$, indicating the component of the error vector with the smallest absolute value. The index k thus denotes the current state from the

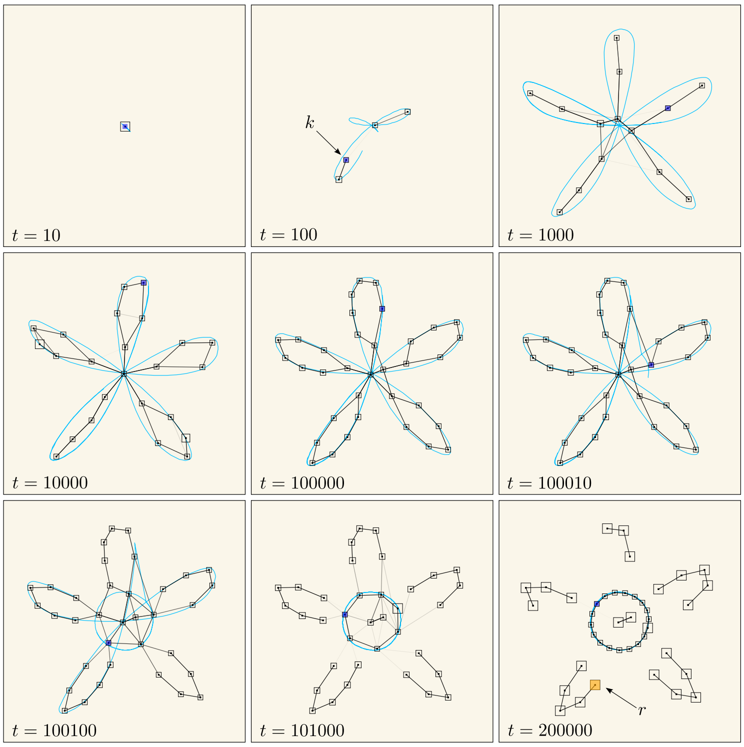


Figure 5.5: The multi-expert structure grows concurrently with progress in learning following the sensor inputs shown in light blue. Sudden changes in the underlying system cause the network to adapt rapidly at time step $t = 10^5$. The former learned structure gets preserved until more new experts are required. Box size around experts denotes remaining learning capacity. The winning expert, k, is shown in blue, whereas the expert selected for potential relocation, r, has an orange color.

set of discrete system states and can directly function as a state detector for tabular reinforcement learning algorithms such as Sarsa. When transferred into a vector, the output would be a so-called *one-hot* vector with each element zero except element k equal to one. Further, GMES outputs the current learning progress, L, appropriate as an intrinsic reward signal.

GMES for Continuous Methods

If we want to connect GMES to continuous methods such as approximating state value functions with a neural network, we better represent the states of the system as a real-vector-valued output, rather than just a scalar value or one-hot vector. To achieve this, the prediction error vector $\boldsymbol{E}$ can be translated using the softargmax function defined in Equation 4.26, which will normalize the values and convert them into a probability distribution over the possible states of the system with $\sum_{i=1}^{N} y_i = 1$. By doing so, we can obtain a real-valued activation vector $\boldsymbol{y} \in [0,1]^D$ that accurately represents the states of the system. Strictly speaking, we compute a softarg*min* output

$$\boldsymbol{y} = \text{softargmax}(-\boldsymbol{E}, \tau), \tag{5.8}$$

since smaller errors are supposed to give a larger activation of the corresponding states. A GMES with softargmax output is therefore referred to as a GMES(τ). Furthermore, we can use the self-regulatory β-softargmax to decouple the size of the prediction errors from the activation.

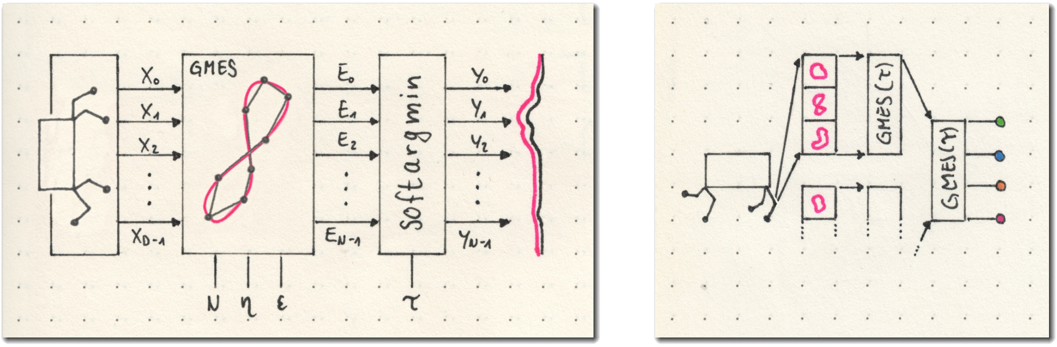


Figure 5.6: Schematics of a single GMES(τ)-layer (on the left): The prediction errors convert into activation values. A multi-layer GMES(τ)-stack (on the right): Previous layers' states are input signals to other layers and compute compound states. Raw sensor data gets compressed from the joint layer via the leg layer to the body layer, forming higher-level states of the robot.

Hierarchical GMES

When we apply GMES to a higher-dimensional system such as a four-legged robot, the question arises whether we should naively insert all sensor data into a single GMES layer or whether we can take advantage of the natural hierarchical architecture of the robot and also adapt the modeling to it. With the Softargmax output, it is now possible to easily arrange GMES in layers and thus also to perform, for example, an increasing compression of the state information over several layers.

Let us imagine that we have a structure at the joint level estimating the state of the joint in angle and angular velocity. If we combine the joints of a single robot leg into a subsequent layer, we get the further abstracted state of the individual robot leg. In the last layer, we combine the four legs and the torso-specific sensor data and obtain the abstract body state. Figure 5.6 illustrates the procedure. If each layer has a comparatively low number of experts, the state information of the entire robot gets compressed while we can still inspect it at each hierarchical level.

GMES in the Context of Reinforcement Learning and Eigenzeit

System time is defined $t \in \mathbb{N}_0$. At each time step, denoted $t \leftarrow t + 1$, there is a new sensor state $\boldsymbol{x}(t)$. Due to the discretization GMES performs, similar sensory inputs are subsumed to the same state. So ideally, the higher level state of the system, s, should change less frequently in comparison to the sensor vector $\boldsymbol{x}$. Similar to that, an action a taken must be *rolled out* for some time steps to have some notable effect on the environment.

The reinforcement learning agent shall preferably operate on a much slower time scale compared to t since frequent updates of the reinforcement learning system may suffer from too much variance and noise. While t increases steadily with fixed step size, the system reads sensory inputs $\boldsymbol{x}$ and writes motor outputs $\boldsymbol{u}$ w.r.t. system time. However, higher-level state changes occur irregularly. Therefore, I propose the term *eigenzeit* (German for proper-time) in reference to physics to describe a time measurement based on more abstract and higher-level system state changes. When eigenzeit progresses with its *eigensteps*, we refer to this as $t^e \leftarrow t^e + 1$, signaling that the agent has a successor state and must choose a new action accordingly.

Referring to the notation of reinforcement learning, we can say $s' = s(t^e+1)$. Eigenzeit passes when the system's (discrete) state changes from s to its successor s'. But what if the state persists? We use a timer to avoid getting stuck when the last action does not cause a state change and after the timeout of $\Delta t > \Delta t_{max}$ a new eigenstep is applied.

Summary

In this section, I introduced the Growing Multi-Expert Structure for unsupervised segmentation of sensory state spaces. The algorithm is tailored for online learning in autonomous legged robots, while it has potential for application in diverse domains. The algorithm ensures that computation time and memory usage remain within expected bounds by deploying experts solely when necessary and relocating unused experts during learning. Additionally, the algorithm measures its learning progress and employs this signal as an intrinsic reward to guide the development of the whole system on a higher level. Structural changes in the network happen solely accompanied by progress in learning, ensuring that the model of the sensory state space remains stable for an extended period if nothing new is learned. Once acquired, the structure retains as long as feasible, developing a robust state space representation.

GMES requires only three hyperparameters and is agnostic to the respective prediction method, where I presented two techniques of different complexity. I also showed interfaces to reinforcement learning and continuous systems using softargmax and eigentime. In the next section, I will demonstrate multiple conducted experiments that prove the algorithm with data.

5.2 Experiments and Results

Based on a selection of experiments, I would like to explain how the GMES algorithm is applied and how to use it. The experiments described in this section run on simulated robots. Another experiment follows in Chapter 9 and is conducted on several physical robots.

5.2.1 Network Growth in Non-Stationary Systems

The initial experiment illustrates the development of the growing multi-expert structure when exposed to non-stationary input data, utilizing the *flower manifold*—a four-dimensional oscillatory system projected onto two dimensions denoted $\boldsymbol{q}^\star$. The trajectory through the phase space does not overlap in higher dimensions and is a quasi-periodic orbit created by two individual oscillators. The resulting figure, resembling a flower, slowly turns like an open rosette orbit but with a much lower frequency than the primary frequency. Therefore, the grown nodes must constantly adapt and are not stationary.

The experts use the moving average predictor. We can thus plot them as nodes, i.e., their weights as a position in space. A small experience-replay buffer of 64 samples per expert gets utilized, randomly overwriting an existing data point from the buffer to accommodate the current one. The gradient descent thus trains a random mini-batch.

I define the test system as:

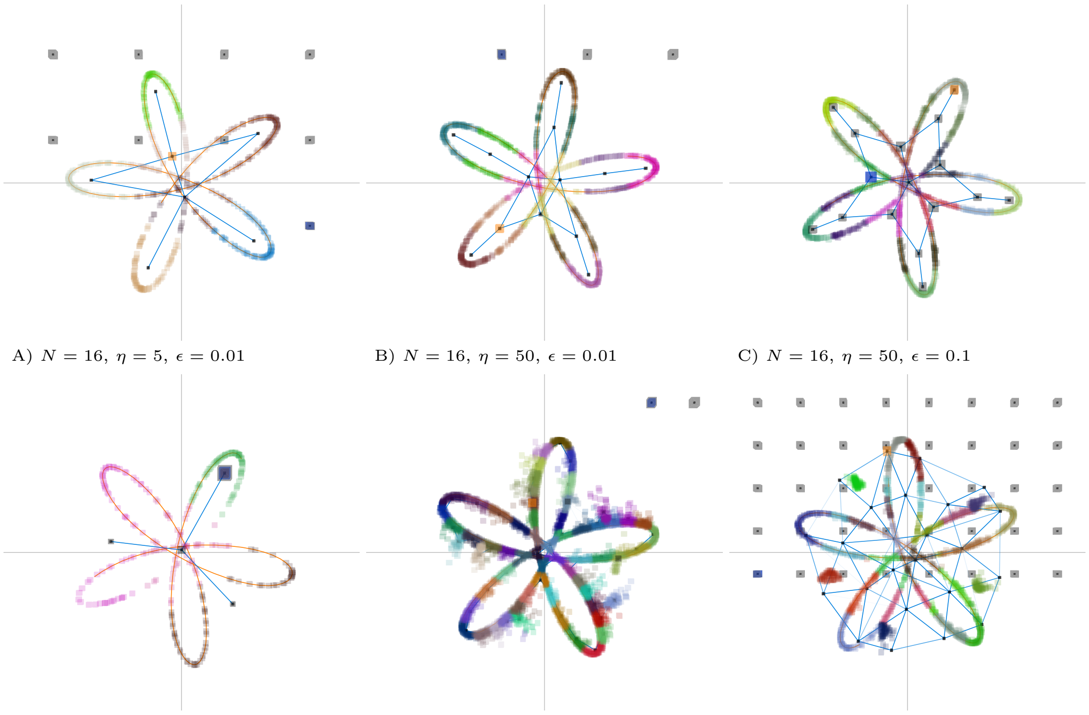
$$\boldsymbol{q}(t) = \tanh(\boldsymbol{W}\boldsymbol{q}(t-1)) \quad \boldsymbol{W} = \begin{pmatrix} s_0 & -r_1 & 0 & 0 \\ r_1 & s_0 & 0 & 0 \\ 0 & 0 & s_1 & -r_1 \\ 0 & 0 & r_1 & s_1 \end{pmatrix} \quad \begin{aligned} s_0 &= 1.005 \\ s_1 &= 1.001 \\ r_0 &= 0.1 \\ r_1 &= 0.15 \end{aligned} \quad \boldsymbol{q}^\star = \begin{pmatrix} q_0 + q_3 \\ q_1 + q_2 \end{pmatrix} \tag{5.9}$$

with $\boldsymbol{q}(0)$ initialized with small random values to start the oscillation.

Figure 5.7 shows the nodes and their adjacent edges. The samples from the replay buffer got color-coded to assign them to an expert. Since the second process of rotating the flower manifold is much slower than the periodic movement of the input data, the experts can adapt and move their position accordingly. Hence, their function got preserved. However, this does not work for all of them. In the projection, the trajectory crosses itself in some regions causing the experts' perceptive fields to overlap partly. By tuning the hyperparameters, we can address this behavior if desired.

The experiment shows the adaptability to non-stationary periodic, temporally strongly dependent samples, which could usually come from sensor data of a periodic motion like walking. It thus demonstrates the consistency of the experts that, once learned, get preserved across slow system changes that result in a non-stationary shift of the input data. Examples of a slowly altering non-stationarity of real-world applications are, e.g., wear or temperature changes in contrast to sudden system changes such as payloads on walking robots. With this experiment, I also want to demonstrate the effects of different parameterizations, exhibiting good results with minimal parameter dependency over a broad range. However, suboptimal areas exist and are also highlighted to aid in identifying problematic parameter values (see the results in Figure 5.7 and the supplementary video.[1])

[1] GMES-Flower growth: TODO upload video link!



A) $N = 16$, $\eta = 5$, $\epsilon = 0.01$

B) $N = 16$, $\eta = 50$, $\epsilon = 0.01$

C) $N = 16$, $\eta = 50$, $\epsilon = 0.1$

D) $N = 4$, $\eta = 50$, $\epsilon = 0.01$

E) $N = 64$, $\eta = 500$, $\epsilon = 0.1$

F) $N = 64$, $\eta = 500$, $\epsilon = 0.001$

Figure 5.7: GMES when applied to the flower manifold at different hyperparameters: Each outcome is presented after presenting $36 \cdot 10^5$ samples equivalent to 10 hours of growth. The scale range used is $[-1, 1]^2$. Squares connected by edges denote experts whereas colored squares denote the samples collected in the experts' replay buffers. The unused experts remain in the background. The top three results, A, B, and C, were obtained by setting hyperparameters close to optimal and demonstrating the algorithm's behavior at varying adaptation rates from slow to high. On the other hand, the bottom three results have suboptimal hyperparameters indicating potential problems. Result D had a too small number of experts, making fitting impossible. Result E, on the other hand, had an excessively high number of experts and a learning rate that was also too high, resulting in overfitting. Result F had a too low learning rate and, at the same time, too many experts, so it cannot cope with the non-stationarity and struggles to keep experts on the manifold.

5.2.2 Intrinsically Motivated Pendulum

The next step was to test the GMES in a reinforcment learning setting on a more realistic system. However testing sufficiently complex algorithms in complex environments is prone to programming errors and misinterpretation of the results. So I decided to use a rather simple system—a single actuator in form of a slowly swinging, underpowered pendulum that was introduced in Figure 2.9.

In this experiment, I employ a simulated but *physical* pendulum, i.e., a rod with a motor and joint friction, as compared to an idealized *mathematical* pendulum, a point mass on massless cord suspended without friction. I use the position, φ, and the angular velocity, $\dot{\varphi}$, as inputs, both normed in the interval $[-1, +1]$. The state space is

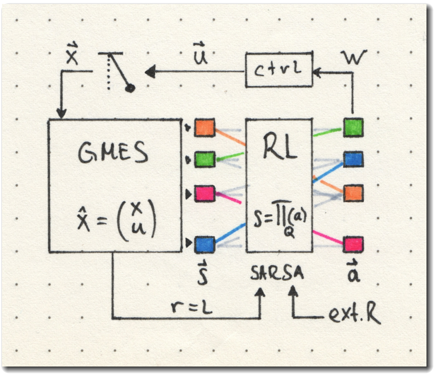


Figure 5.8: Schematics of the intrinsically motivated pendulum experiment: The state space gets discretized by a GMES module. Actions are pre-defined, similar to bang-bang control with additional breaking action. Motor controls come in the form of weight matrices executed by the controller. A Sarsa(λ) reinforcement learning creates the mapping from states, $\boldsymbol{s}$, to actions, $\boldsymbol{a}$, to maximize the reward. GMES produces intrinsic reward, $r = L$, guiding the state space exploration.

constructed by GMES in an unsupervised manner and generates the respective set of discrete states. The actuator has no end-stops so the rod can rotate freely. To avoid the discontinuity of the angular position measurement at the transition $[-1 \leftrightarrows +1]$, I expand the input dimension to the tuple $[\sin(\pi\varphi), -\cos(\pi\varphi), \dot{\varphi}]$ instead of only $[\varphi, \dot{\varphi}]$, so that the effective input dimension is higher. In reality it is still only a two-dimensional (cylindrical) manifold embedded in a 3-dimensional input space namely, $[-1, +1]^3$, but is continuous now. The maximum number of experts (i.e., states) is limited to 100. If even more new experts would be required, GMES would reposition long inactive ones in favor of making the whole system more effective.

I use a discrete tabular reinforcement learning, more precisely Sarsa(λ) as described in section 4.3. It should be noted that this is therefore also a dynamically growing action-state matrix, $Q(s, a)$. The rows of the matrix correspond to the states whilst the columns are corresponding to the actions and hence each matrix element gives information on how promising it is to execute a given action in the current state. Columns are static, but the rows grow with the number of experts. As actions I use $[0, +1, -1, -0.5\dot{\varphi}]$, corresponding to the torque applied in the pendulum's suspension point, i.e., an extended bang-bang control, where the amount 1 is the maximum torque. The action 0 does not exert any torque, while $-0.5\dot{\varphi}$ brakes proportional to the systems speed. The system is underpowered in that the maximum torque is insufficient to move the pendulum upward without several coordinated upstrokes, similar to the mountain car problem.

I defined a total of three policies each with a different reward. One for swinging up ($\uparrow$), one for resting ($\downarrow$) and one intrinsically motivated ($\star$). The first two tasks are considered classical goal-directed tasks, while the third task involves implicit goals such as exploration and learning. The reward for the third task is non-stationary since the system is rewarded for successful learning and discovering new situations. This encourages the system to explore and refine its predictions about the world. However, as the system's predictions stop improving, the rewards become progressively fewer, prompting it to further explore or revisit previous states. The table below defines the rewards, which are generated with respect to the current time-step t using $c = 100$. The goal-directed reward functions $r_{1,2}$ were compiled from narrow Gaussians placed on the desired target positions in phase space (including velocity). This implies the reward is normed between $[0, 1]$. All rewards are accumulated until the next eigenstep (discrete state change).

swing-up	resting	intrinsically motivated
$r_1 = e^{-c(1-\lvert\varphi\rvert)^2 - c\dot{\varphi}^2}$	$r_2 = e^{-c\varphi^2 - c\dot{\varphi}^2}$	$r_3 = L$

Method, Results, and Data

The experiment was carried out in four different ways. The first approach ($Q_\wr$) involved the system performing random actions without any reinforcement learning turned on. This method is similar to previous work in Kubisch et al. (2010) where the reward signal was replaced with noise of the same distribution, resulting in the system not being able to draw any information from the reward. The effects of both approaches were found to be effectively identical. In the second approach ($Q_\star$) the system was designed to learn only intrinsically motivated, where actions were rewarded based on the system's learning progress, L. It should be noted that the learning signal was found to be non-stationary since learning progress vanishes once the expert is sufficiently trained and can be quickly rising when new experts are inserted to accommodate for new samples during exploration. The third approach ($Q_{\uparrow\downarrow}$) involved goal-directed learning where extrinsic rewards were given for swinging up the pendulum and slowing it down to the rest position in alternating cycles. Finally, the fourth approach ($Q^\star_\updownarrow$) involved randomly alternating between all three policies $[\star, \uparrow, \downarrow]$. The comparison to noise is significant because it demonstrates that random actions, which are often referred to as exploration in machine learning, may not be the best exploration strategy under limited force, as the gravity's strong attraction can keep the pendulum in a predominant position unless the actions are used coordinately to escape from it.

We measure the network growth after 15 and 120 minutes of learning respectively in each of the four tasks. In the beginning hardly any differences can be seen between the approaches through random initialization but $Q_\star$ is already ahead. But after 120 minutes a very clear picture emerges: The random actions are hardly able to leave the attractor since non-coordinated actions will only lead occassionally to the rod escaping gravity with an underpowered motor joint. When it comes to employing intrinsic motivation we can see that it helps the pendulum greatly with the exploration of the phase space and it reaches the most regular phase space coverage of all four approaches using more experts than the others. It finds paths of higher velocity (longer and more coordinated swing-ups) and empowers the system to chose from more options. The purely goal directed approach finds the upright position, however with limited options to move since it receives only reward for reaching the goal directly regardless of the path. In the combined approach where intrinsic motivation and goal-directedness are coupled, the system benefits from the more *systematic* exploratory movements of intrinsic motivation. It also shows a good coverage and regular network growth and wastes less time in the bottom position, yielding more rewards for swing-up.

Summary und Learnings

What are the findings from this experiment? The use of intrinsic motivation as an exploration strategy is effective in allowing an underpowered system to overcome obstacles that can hardly be overcome through random actions alone. Additionally, the dynamic

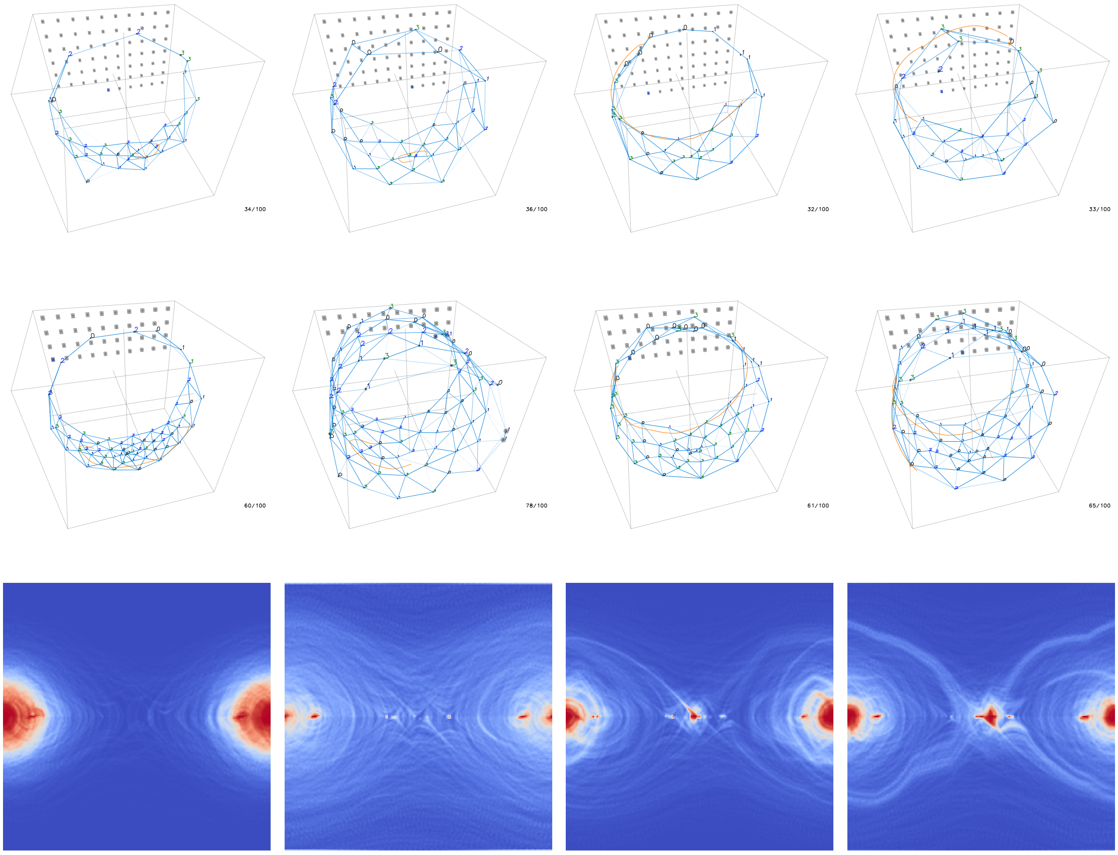


Figure 5.9: Snapshots of the system after 15 and 120 minutes of learning are shown in the top two rows. The four approaches used are: random actions, intrinsically motivated only, goal-directed only, and a combined approach. Shown in the bottom row: histograms of the pendulum's average position in phase space (2D). The axes used in the histograms are $(\varphi, \dot{\varphi})$ where the center point represents the pendulum's upright position. Results were obtained using the following parameters: $N = 100$, $\eta = 30$, $\epsilon = 0.003$ and RL: $\varepsilon = 0.05$, $\gamma = 0.99$. After 15 minutes all systems still almost look the same, but after 120 minutes clear differences emerge: With random actions only it can hardly leave the attractor, the intrinsic motivation helps enormously in the exploration of the phase space and reaches the most regular coverage and furthermore enables the goal-directed learning to find more paths to the upright position in the combined approach.

growth of state space discretization during active goal-oriented reinforcement learning was successful, even with this simple form of tabular reinforcement learning. Rewarding learning progress promotes a regular growth of the multi-expert structure through systematic exploration, making it more efficient. The first full-system test has thus been completed, and even though it is a simulated system, many components are included, especially sticking friction, sensor noise or underpowering. Not surprising but worth mentioning is the fact that even the simplest test systems like a 1D robot in the form of a pendulum can already show complex behavior in connection with learning algorithms and provides numerous starting points for further studies. Let us now step forward and apply GMES to a crawling robot and observe the outcome. In the next section, I will also address the question of where the actions may come from.

5.2.3 Growing States and Actions

In this experiment, I employ the GMES algorithm on the Tadpole robot, as introduced in section 2.3, with four joints. Unlike the previous experiment, which used the pendulum robot, we now have a more extensive sensorimotor state space to cover. To predict the state, we utilize a time-delay neural network (TDNN), as shown in Figure 5.4, with a moderate configuration of an 8-tap delay-line and a single hidden layer containing 16 neurons. Otherwise, we would need too many experts if we stick to covering the higher dimensional state space with the moving average predictor.

In addition to classifying the sensor space, the self-organized network is used here also in the motor domain and learns to categorize it. Instead of the sensor inputs, motor GMES tries to predict the action outputs. The *motor experts* use the generalized controller structure from Equation 1.15 as predictors. The experts also grow in number, making this system effectively an unsupervised learning motor space that expands as needed. Thus, the motor actions are no longer predetermined controllers but are themselves subject to change according to the robot's experiences.

Unsupervised Locomotion Learning with Homeokinetic Exploration

But where do the initial motions come from that the motor experts can learn to segment into primitives? We could apply randomized motor actions, but I did not expect it to be significantly faster in learning than evolution. So I needed a better approach. Remember, motor experts have two functions: The first is to predict motor actions exerted by the system. Since the predictors are technically generalized controllers, their second function is that we can apply them to control the robot. So they can learn to reproduce the motor actions of another source. So I implemented the homeokinetic controller from subsection 4.2.3 that drives the robot to show constant activity. It generates well-behaved motions that are a suitable source of motor exploration for the experts to learn from. The experts get collectively trained to copy the homeokinetic actions. A side note: The evolved controllers from the earlier evolution experiments would form another exploration source, but compared to the space of possible motions they are limited, and we better keep them as a test set.

How does learning happen at the unsupervised level in the motor domain? The motor experts predict the motor outputs, $\boldsymbol{u}$, and, as usual, the best-predicting expert wins and is allowed to adapt to the stimulus. After frequent learning steps, the experts' learning capacity has gone out, they might get cloned, and thus a copy of them gets added to the motor expert group.

As a comparison experiment, I tested the algorithm with a provided set of six evolved behaviors for the Tadpole robot, originating from the evolution experiments' results in section 3.5. The obtained behaviors include fast and efficient forward and backward movement, as well as clockwise and counterclockwise rotation (refer to Figure 3.5). They

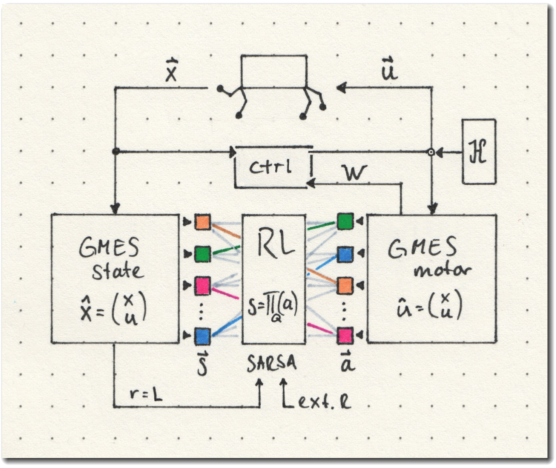


Figure 5.10: Schematics of the fully unsupervised locomotion experiment: State and action spaces get discretized by two individual GMES modules. A reinforcement learning of type Sarsa(λ) creates the mapping from states, $\boldsymbol{s}$, to actions, $\boldsymbol{a}$, to maximize the reward. Since GMES produces intrinsic reward, $r = L$, this signal guides learning the same way as in the experiment before. The homeokinetic controller, $\mathcal{H}$, creates exploratory motor actions to help the self-organizing motor space to grow. The motion primitives that the motor GMES indentifies, the weight matrices, W_k, get executed by the controller.

form highly optimized actions and its seems unlikely to me that they can be easily improved further. Therefore, I consider them as a solid baseline to which solutions found through the learning setup presented here can be compared. Additionally, these evolved actions help us to fine-tune the hyperparameters of the overall system.

The robot can practice in an open-ended way, following each policy for 60 seconds in random succession. The actions get selected according to a whole of six different policies: intrinsic state learning ($\star$), intrinsic motor exploration ($\mathcal{H}$), forwards ($\rightarrow$), backwards ($\leftarrow$), turning left ($\curvearrowleft$), turning right ($\curvearrowright$). I assume these policies will provide enough diversity in the selection process. The epsilon-greedy exploration rate is set very low at 0.1% because the system already experiences enough exploratory selection through the highly non-stationary rewards of the policies $\star$ and $\mathcal{H}$. There are a maximum of 12 different discrete states consisting of TDNN experts and eight discrete motor actions consisting of generalized controllers. The number of state experts is significantly smaller compared to the previous pendulum experiment, but the predictive capability of the experts instead is much higher. I initiate the experiment with one state and one motor expert each.

The hyperparameters for both GMES modules are identically set to $\eta = 1.0$ and $\epsilon = 0.0025$, encouraging a moderately slow growth. Parameters for Sarsa are $\alpha = 0.001$, $\gamma = 0.9$, and $\lambda = 0.9$. Initial Q-values are set optimistically to 1. Since only one expert is allowed to adapt, the average learning speed in GMES is even lower. This is essential for the RL module to keep pace. Since I employed eligibility traces in the RL, the full Q-matrix is updated according to the states' traces, whereas the experts change by ϵ/N. The ratio of both modules' learning rates is crucial for success. RL must learn faster than its states and actions drift.

The homeokinesis module comprises bidirectional model pairs instead of function inversion for C^{-1} and M^{-1}. This means, I employed standalone models for the forward and backward computation, so there is no additional effort in the implementation and tools. The controller, $\boldsymbol{y}_t = C(\boldsymbol{x}_t)$, and its inversion, $\hat{\boldsymbol{x}}_t = C^{-1}(\hat{\boldsymbol{y}}_t)$, as well as the model, $\hat{\boldsymbol{x}}_{t+1} = M(\boldsymbol{y}_t)$, and its inversion, $\hat{\boldsymbol{y}}_t = M^{-1}(\boldsymbol{x}_{t+1})$, are learned separately as single layer neural networks with tanh transfer function and minimal L1 regularization to encourage sparsity.

I conducted two different runs: First, I ran the experiment with the six evolved actions and measured the average reward as a baseline comparison curve. Second, I started the experiment with a randomized standard controller initialization and observed the development of the self-organized motor layer under homeokinetic exploration.

During an initial growth phase, I limited the system to explore the state and action space using only the intrinsically motivated policy and homeokinesis. As the number of experts increased and the state-action space underwent changes, it challenged the RL system to keep pace, leading to temporary sharp drops in performance. However, by providing sufficient pre-growth until the networks reached their predetermined growth limits, the RL system could adapt effectively to small shifts without experiencing significant performance loss.

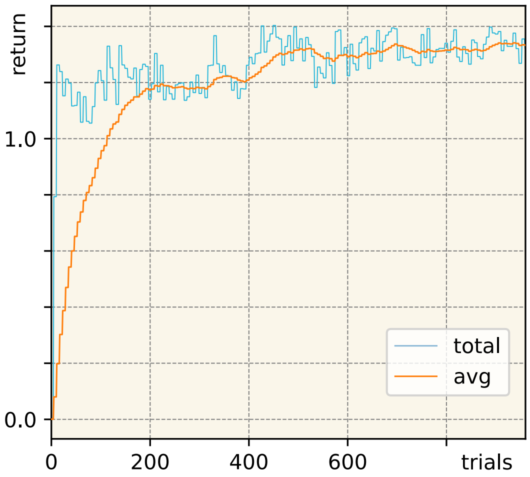


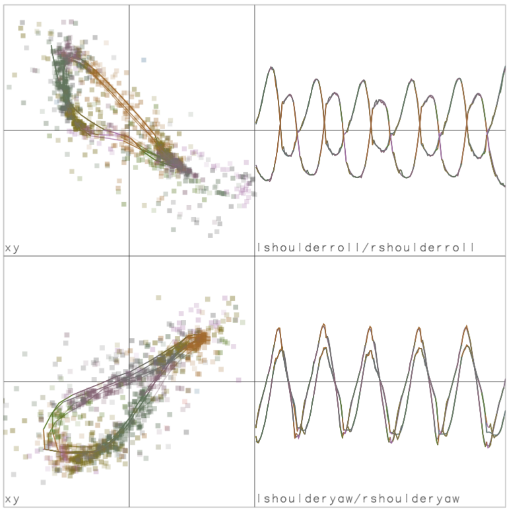
Figure 5.11: Total cumulative rewards in the Tadpole baseline experiment with evolved actions: The image depicts the results from 960 trials of 60 seconds each. The experiment entailed a 20-hour duration, including 4 hours of pre-growth following the intrinsically motivated and homeokinetic exploration ($\star$, $\mathcal{H}$) without external rewards for crawling ($\rightarrow$, $\leftarrow$, $\curvearrowleft$, $\curvearrowright$). The displayed return represents the mean rewards obtained from all six policies.

Results of the Baseline Run

The baseline experiment demonstrates that using GMES with a 4-DoF crawling robot is possible. More complex neural networks, e.g., TDNNs, can also be used as expert predictors. GMES allows for connecting continuous sensor values to discrete reinforcement learning and facilitates the interpretation of the learned behavior by inspecting the individual experts' prediction errors or Q-values.

For the baseline experiment: RL/SARSA is able to correctly associate evolved actions with self-organizing states and matches the given controllers to their corresponding policies. This results in a stable policy switching process that enables the selection of different gaits and behaviors by activating the relevant policies. In Figure 5.11 we can see the total reward of the system from the baseline experiment. From the perspective of a human observer, the resultant robot motions look indistinguishable from evolution except for an improved switching between two robot motions.

The two combined GMES modules effectively organize the growth of states and actions together. In Figure 5.12, we see samples of the multi-expert prediction and the prediction error decreasing over time. A characteristic of the error is its high variance due to the policy switching with each trial.



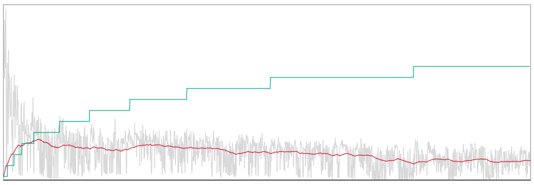


Figure 5.12: Predictions of the GMES state module (left): Each plot displays a two-second (200 samples) trajectory of the joint positions L/R shoulder roll and L/R shoulder yaw for the Tadpole robot. The colored rectangles indicate expert-specific buffer samples, while the colored sections represent the experts' predictions. The number of experts and the development of the prediction error over time is depicted below.

Results of the Full Learning Run

For the second run, the behaviors learned from scratch demonstrate a satisfactory quality. However, their performance is significantly lower than what was achieved by evolution. After 48 hours of learning it reached about 50% in the best case. It is worth noting that, in contrast to the earlier evolution experiments, the time required to find these motions with our new approach is significantly shorter. In some runs the homeokinetic controller almost instantly creates a crawling motions with a similar appearance. However for the full GMES/RL system to adopt these motions it takes about 20% of the time evolutionary methods needed.

The defined reward functions successfully reproduce qualitatively similar robot behaviors as with the corresponding fitness functions. Although symmetry regularization could accelerate learning, it was intentionally left out. The convergence of the learning process is moderate, but the performance graph shows strong oscillations. It is critical to finetune the hyperparameters, especially the learning rates, to ensure they match. Q-learning for off-policy training helps prevent a reward runaway of individual robot behavior. For the TDNN expert modules, I found that increasing the number of hidden neurons works better than using more time delay taps. The sensor space appears to contain sufficient time information. The homeokinetic module effectively diversifies actions and controller matrix growth. Also, pre-growth is crucial and intrinsically motivated, and homeokinetic exploration helps to develop the sensorimotor space and prepares it for successful goal-directed learning. Premature goal-directedness leads to early local optima. I want to emphasize that the learned actions are not single controllers, as in the evolution experiments, but are always a sequence of state-action pairs. Although the system requires frequent controller switching, the individual parts are reusable and combinable. So we might be one step closer to motor primitives.

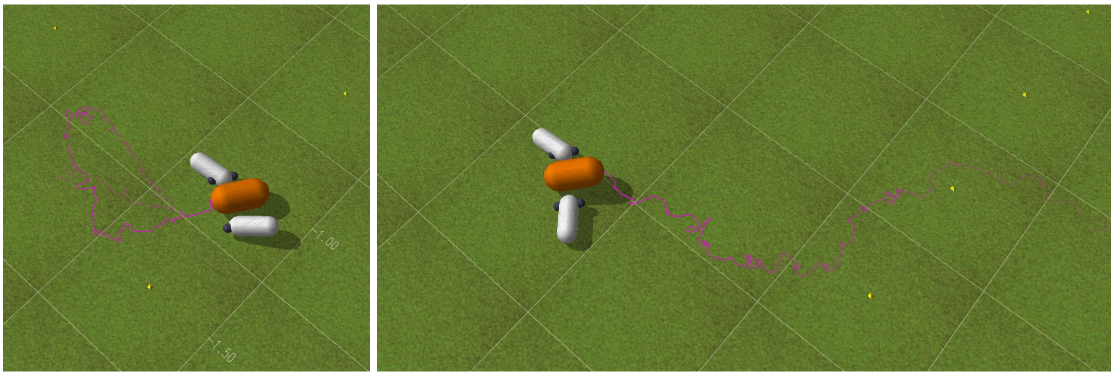


Figure 5.13: Tadpole robot crawling backward and forward learned in the fully unsupervised learning with growing states and actions: The behaviors learned in this experiment achieved approximately 50% of the performance observed in evolution experiments but required significantly less computation time (approximately 10%). While it may take the robot more time to enter the crawling attractor, the learned behaviors have the ability to switch to different behaviors mostly without getting stuck since they are learned together and share the same states and action elements. The magenta trajectory of the body clearly shows that the motion is much more irregular compared to the results from evolution shown below.

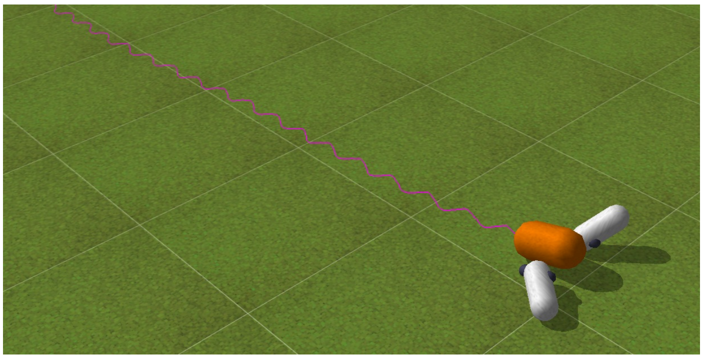


Figure 5.14: Tadpole crawling perfected in evolution experiments: In contrast to the learning experiment's results depicted above, the results from evolution show a more regular body trajectory but take more time to find the parameters and require repeated resetting of the robot. Also, the robot only learns a single behavior at a time.

Limitations and Learnings

The motor GMES is limited in adapting to what the homeokinetic controller explores. Homeokinesis does not systematically search for behaviors but strives through a succession of behaviors driven by external stimuli and the natural noise in the system. So there is no guarantee of discovering the robot's entire action space employing this method. To increase the chance of exploring more, I conduct regular resets of the homeokinetic controller and re-randomized its initial weights.

During experimentation, one particularly nasty effect emerged that cost me several attempts to fix. After an initially successful phase of continuously increasing performance, one or more robot behaviors suddenly collapsed and hardly recovered, engraving large oscillations in the performance graph. I could only mitigate this convergence failure with the proper coordination of learning rates and other hyperparameters. Also, I settled on using Q-learning for all off-policy adaptations.

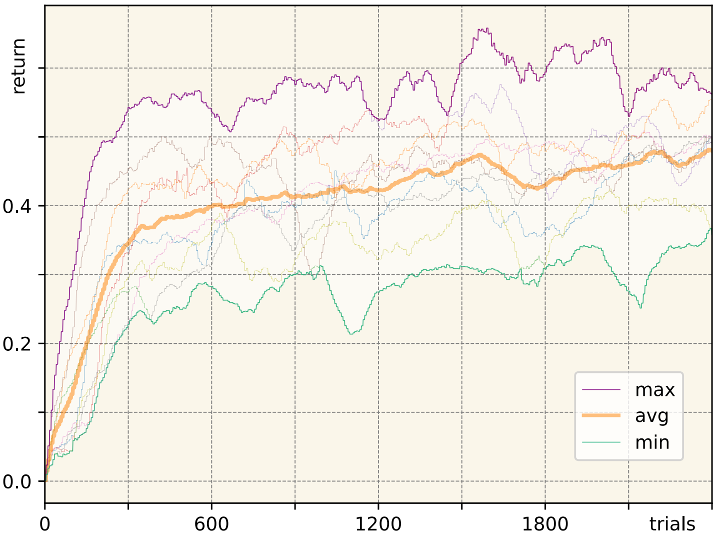


Figure 5.15: Total cumulative rewards in the Tadpole learning-to-crawl experiment: This image depicts the aggregated results from 10 distinct experimental runs, spanning 2400 trials of 60 seconds each. Each experiment entailed a 40-hour duration, accompanied by an additional 8 hours of homeokinetic and intrinsically motivated exploration without rewards. The displayed return represents the mean rewards obtained across four distinct walking directions.

Several explanatory models are possible: RL always assumes a static state-transition model. Hence, any change in this model must happen quasi-statically from the perspective of RL. Careful coordination of the ratio of GMES and RL learning rates can help to achieve this, i.e., keeping $\epsilon/N \ll \alpha$.

In this experiment, all Q-matrices get modified simultaneously, so I naturally experimented with off-policy Q-learning updates for all off-policy actions (instead of Sarsa). But this initially created a divergence of the Q-matrix when using non-stationary states and actions and optimistic initialization, resulting in also large oscillations and the collapse in performance. My explanation for this phenomenon was that using the maximum operator (see Eq. 2) is inappropriate for highly stochastic learning problems like the one I faced, notably when the state predictors are changing. It overly preserved high rewards obtained by chance, only once, or by optimistic initialization. The caused distortion in the Q-value estimates was so severe that the measured average performance of the entire system did not increase regularly.

For the sake of simplicity, I wanted to stick to using on-policy Sarsa(λ) for also adapting off-policy actions. But it did not perform well either since using on-policy adaption for off-policy training led to non-optimal action-value functions and low overall walking performance. For two behaviors with symmetrical rewards, such as forward and backward walking or left and right turning, the Q-matrices' maldevelopment caused the corresponding symmetric behavior to collapse suddenly, whereas the original one developed well. I could only prevent this by applying the max-operator in the RL update as in Q-learning but explicitly refraining from using too optimistic initialization and keeping the state space as constant as possible for RL. The update rule for Q-learning then involves taking the maximum over all action-values of the subsequent state s':

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_a Q(s',a)) \tag{5.10}$$

To further prevent an unwanted drift of the growing state and actions spaces, GMES learning gets inhibited when externally rewarded policies get trained. For intrinsic motivation and homeokinetic exploration, the GMES for both state and motor were allowed to adapt since intrinsically rewarded policies are naturally non-stationary and not expected to converge. So we can use a higher learning rate here.

Discussion of the Results

After all, I still consider the experiment a success, even if I could not achieve the same level of walking performance and a smooth convergence until now. The Tadpole robot is comparatively low in its degrees of freedom but can now autonomously adopt a spectrum of different actions in an open-ended learning setting. Despite all the limitations that are still present, the robot can acquire its behaviors from scratch in significantly less time and practices multiple motions quasi-simultaneously. Employing fully unsupervised methods for the substructures and incorporating intrinsic motivation enables this system to learn autonomously. The only necessary supervision remains in defining extrinsic reward functions by a human experimenter. A possible modification to achieve locomotion learning based only on pure intrinsic rewards, albeit complicated, has been proposed in the closing remark of section 4.4.

After conducting the evolution experiments in Chapter 3, I found myself left with the question of how to switch between individual robot behaviors that I could discover in the process. While I was able to design an experiment that could switch between two of the behaviors - standing and fast running - it was generally not possible to switch smoothly between any two behaviors, as the transitions were not part of the optimization. However, the experiment from this section clarifies how to address the issue. When the robot concurrently practices all behaviors while we explicitly avoid resetting the robot to its starting position, the learning algorithm is implicitly forced to fill the gaps between the behaviors. Instead of resetting the robot, it must recognize its current state and select the optimal action from a range of choices based on the policy, regardless of the robot's current orientation and joint position. The concurrent learning also pressurizes the generalized controllers to increase their basin of attraction, in other words, to start oscillating from a wider distribution of initial states.

Conclusion and Outlook

This chapter presented a new algorithm called Growing Multi-Expert Structure and its applicability in learning robot movements. Through several experiments of increasing complexity, I tested mixed scenarios and applications with simple robots. As a result, building blocks for a fully unsupervised learning architecture have been newly designed and tested. While the results are still modest compared to the evolutionary approach, considering the complexity of the problem, I faced enormous complex dynamic interactions in such a system. Additionally, numerous parameters need to be finely tuned. For a novel approach, I find this already significant progress. Regarding tests on high-DoF simulated robots, e.g., the Quadruped or even real machines, we need further investigations for the fully unsupervised variant with self-organized actions to reach comparable results. Fundamental questions need to be addressed first, such as how to stabilize convergence.

Summary of Part Two

In part two, we have discussed fundamental learning techniques: supervised, unsupervised, and reinforcement learning. In doing so, we explored various unsupervised approaches, such as competition in growing structures. I derived the GMES algorithm from these components and tested it through experiments. With growing structures as a fundamental building block for learning algorithms, I aimed to solve two common problems: First, the model's capacity should adapt to the system, dynamically expanding the robot's states and actions as needed. Simultaneously, the model must be robust against catastrophic forgetting while learning in non-stationary settings.

I attempted to transfer the results of my evolutionary experiments from part one into an approach based on reinforcement learning. The idea was not only to train individual behaviors using a different algorithm but to transition the learning setup into a fully unsupervised one, with an eye toward later applicability to real-world systems. My vision is to place a real robot in a field and have it practice autonomous walking, and I believe I have moved a good step toward my goal. The method I employed and explored achieved moderate crawling movements on a low-DoF simulated robot within a manageable real-time equivalent of approximately two days. It required no more than the computational power of an embedded system. Therefore, from the algorithmic side, I see no insurmountable obstacles to its future field application.

However, I left some open questions for upcoming research: How can we use the principle of homeokinesis not only as an exploration module but as individual motor experts in a growing structure, i.e., growing homeokinetic multi-experts? For this, it remains to solve how the principle of homeokinesis can transcend to the generalized controller without having the mandatory position controller in the loop and directly outputting motor voltages. Homeokinesis explores and attempts to destabilize the system to generate activity. Used as an individual motor module: How can convergence in motor learning be achieved? When are the motor primitives fully learned?

Initially, I only used the simplest form of reinforcement learning as part of my learning architecture, demonstrating its general feasibility with the fewest assumptions. For the next attempt, it is worth exploring the approximation of the action-value function as a neural network, which utilizes the capability of GMES(τ) to provide continuous softmax network inputs. On the motor side, blending individual controllers is thinkable by mixing their contributions based on the policy's softmax outputs. I expect it to smooth out behavior transients and allow for numerous combinations of controllers, effectively expanding the action space while maintaining the same number of motor primitives.

With this, I conclude the theoretical parts of this work and transition to the practical aspects, describing my attempts to bring myself closer to the goal of autonomous, self-learning of *real* robots from a hardware perspective.

Part III

Open Hardware Designs for Locomotion Robots

The third part focuses on robot hardware. So far, I conducted the learning experiments purely in simulation but tried to generalize them to later work on real robots. To build robots for locomotion self-learning, we need robotic components suitable to perform sensorimotor loops and flexible enough to create different morphologies. At the same time, they must be sufficiently simple to be used and modified by computer scientists like me without explicit engineering background.

The following chapters describe the designs of robots and electronics I built. The four-legged robot *Hannah* and the pet-like robot *flatcat* got dedicated chapters. With the publication of this thesis, I will also release their schematics and source code to the public, connected with the hope of being useful for other robotics research or developments. I want to dedicate the last part of this work to the open-hardware community and express my gratitude for sharing their knowledge and allowing me to learn from it.

6 Hardware Design Considerations

Legged robots are poised to transition from the realm of research to practical implementation. Thus, individuals interested in experimenting with walking robots now have the option to either purchase a pre-made one or construct their own. For those seeking to explore meaningful applications for walking robots, multiple platforms are already available for testing purposes. Nevertheless, the market for walking robots remains relatively small at present. Promising applications are envisioned in the fields of logistics, toys, and remote observation in the foreseeable future.

With more people involved in this technology, numerous additional and remarkable application ideas and projects will emerge. Therefore, I believe it is crucial that people further spread their knowledge about this technology. Thus, I committed parts of the chapter to discuss general design patterns for open-source robotic platforms, aiming to benefit researchers, educators, and founders. I also want to show the connection between free hardware and sustainability. The second part of the chapter then establishes the requirements based on the desired ideals by which I developed robot hardware.

6.1 Free Hardware Designs

What is it meant to be «free» for hardware designs? When using the term free, we usually refer to it in the sense of liberty or freedom, not free of cost. In contrast to software, which we can copy with negligible expenses, hardware does not (yet). So a single piece of hardware is owned by someone, and we usually mean the freedom of the hardware designs: the construction plans, the schematics, and the assembly manual. Today, the designs come as files that also can easily be copied.

So the hardware designs can generally be free, and we can share the design files as we can share software. However, we should acknowledge that free hardware does not necessarily mean that getting a copy of the designs is free of expenses. For instance, if the hardware design comes as a printed book, it naturally has cost. Therefore, we ought to differentiate between meanings of free by using the term *free/libre*.

The freedoms that come with free/libre hardware designs are twofold: first, there is the freedom for the user of the hardware design. The users are free to utilize the hardware in whatever context they like. There shall be no restrictions to specific applications or moral constraints, even though I discourage the use within military contexts or applications harmful to living beings or our ecosystem. Second, the user shall be able to modify a design when any redistribution or publication of derived work gets published under the same license. Furthermore, the user is encouraged to build atop the hardware design, for example, to invent a new business case, and create value for people.

The second freedom refers to the design itself. Once it is free, nobody can restrict its usage or can remove its liberty. After a design file is released, everybody can fabricate hardware from it[1]. However, just copying the designs and selling the products shall be harmonized with the commercial interest of the original author (if there is any) with the help of a specialized license agreement or a general non-commercial license[2].

Free hardware designs make only sense if the tools to create or modify them are widely available and affordable. These days, for instance, it would be helpful to have open-source microprocessor designs that we could change to our needs, but, unfortunately, our capabilities for producing real chips from these designs are still quite limited due to the extensive costs of creating chip factories. If microprocessor production from freely available hardware designs were as easy and affordable as PCB manufacturing today, I assume there would already be numerous freely available chip designs.

The situation is completely different for custom mechanical parts made from metal, wood, or plastic. We can mill, laser cut, or 3D print them in almost every fab lab or makespace. Also, 3D printers have entered university labs and peoples' homes. In short: personal fabrication capabilities have significantly improved since 3D printers, laser cutters, and small-sized milling machines are affordable for people's private usage. Hence, people tend to share their designs online.

Is a design free, even if not all parts are free? Yes, it is. Even though all components would ideally be free down to the last tiny detail, a few non-free ones do not affect the freedom of the overall design. General mechanical parts like screws or ball bearings are already free. Their construction designs are common knowledge, described in many textbooks, or possible patents have expired. If the essential parts of our design are free and the non-free parts are exchangeable so that we do not necessarily rely on them, they could not affect the overall freedom of our design. If, in contrast, we rely on a few non-free and non-exchangeable parts, we should consider redesigning them under a free license. This idea brought me to the design of custom servo controllers described in the next chapter.

6.1.1 Economy Based on Open-Source Hardware

In this section, I want to outline possible modern business models with free/libre hardware designs to encourage actions towards a sustainable economy based on values like sharing, cooperating, and ecological compatibility. One might argue that releasing hardware designs and schematics without patents or protection will inevitably destroy any business model. But I can hopefully point out clearly that there are various possibilities to create value with free/libre hardware designs the same way the free software movement demonstrated years ago. Some of my arguments might be obvious and not surprising, while others might be doubtfully idealistic at first glance. But from the perspective of the user (or customer), all these approaches allow for a broad distribution of related knowledge, an efficient hardware evolution, and sustainable products while

[1] `https://www.gnu.org/philosophy/free-hardware-designs.html`

[2] For readers interested in free hardware licenses, I recommend reading about GPL, LGPL, or Creative Commons Attribution Share-Alike (CC-SA).

avoiding market domination by a few companies with invisible policies. In the context of this work, I will describe my arguments using a walking robot to make it more tangible. And as I have been a technical founder myself since 2019, building open-source hardware, I find this topic is highly related to my work described here, so I naturally feel obligated to share my thoughts on this.

1) In the first business model, there is *nothing* to sell. The schematics and construction designs are freely available, and everybody is allowed and encouraged to build their copy of the robot and is enabled to modify it to their own needs. The user is furthermore free to redistribute the modified work but must publish derived work under the same license (copyleft), which guarantees that the work remains free. So, there is no expected monetary return, but instead, there is a potential knowledge return when people start building derivatives of this design and will eventually share their ideas and insights, too.

2) The second, more obvious model is about selling *assembly kits* of robots, which everybody can assemble within a few days with only the help of documentation. The developer or researcher can start with the initial idea after a comparably short setup time but is already familiar with the construction, schematics, and documentation. Also, it is possibly the cheapest way of getting a fully functional robot. This business model also includes selling replacement parts or even robot modules, e.g., motors or legs.

3) The third option involves selling fully tested and *ready-to-use* robots. Unlike the previous business case where the robot required assembly, users can begin immediately. It naturally results in higher costs for the robots as the assembly is part of the product. However, customers can quickly initiate application development, particularly without a workshop or laboratory. Additionally, replacements will sell at a higher assembly level and price, such as fully functional legs with deployed firmware and calibrated sensors. Consequently, we get parts that we can mount within minutes.

4) The fourth idea comprises selling only *services* instead of hardware. It is about leasing fully assembled robots, possibly equipped with additional sensors or software for special applications. With this option, we can use the robot without owning it. Setup and maintenance got outsourced to a contract partner.

5) The fifth proposal is similar to the fourth but comprehends the presence of a robot handler. It is about services that include the support or guidance of humans and robots mutually. So the service is merely composed of human work using the robot as a tool or as robot work using the human as support for tasks that are too complicated for robots for the time being.

6) Eventually, the last presented model is about derived work. Potential customers could require robots with specialized functionality but do not have the expertise or capacity to modify the original platform to match the requirements. Business partners could initiate the development using the robot as a platform and will make the necessary modifications or enhancements as a product.

None of the business models presented here has a compelling need to keep the construction plans or schematics secret. On the contrary, everyone, especially the customer, can benefit if the underlying designs are free. Especially if the original manufacturer no longer supports the product, others can still offer repairs or updates, leading to prolonged and more sustainable usage of products.

However, there are counterexamples where disclosing everything can turn against us. For instance, when a product can be manufactured automatically at a low cost, it may be prudent that we only disclose the essential parts of the design that are important for those willing to learn or repair. We should refrain from publishing design files necessary for setting up production machines. For example, if the product is a circuit board that everybody can order fully assembled and soldered automatically using the available CAD files, sharing detailed production information may lead to copycats flooding the market with identical products at lower prices. As a result, this will reduce the original author's profits while increasing the demand for support services or can even lead to a collapse of its business and eventually end the product evolution. In such cases, it is advisable to only release design information in irreversible formats (PDF, PNG, etc.) without overly precise production details. It still allows individuals to learn, repair and derive their products using their additional resources. All can benefit from the hardware's freedom without causing economic harm to the original author's legit business interests.

6.1.2 Sustainable Design Criteria

Sustainable design will be tremendously important for our future economy to become carbon neutral. I believe the following design criteria can help design products to be sustainable by definition from the start.

Low Cost

Why must building a robot from a free design be of *low cost*? In principle, it is beneficial that many people interested in building robots should be able to afford or build their copy of it. Robotics research projects with the need for hardware might have limited project funding. Hence, to amplify research in this domain, having low hardware costs could increase the probability of a project being approved and realized.

Furthermore, application developers of small start-up companies who want to try an idea would be encouraged to realize it when the hardware is cheap and free to modify. These days, it is not yet clear what will be the first mass-market economic applications using legged robots. Nobody can forecast how legged robots will look ten years from now and which tasks they will fulfill in our society. But it would be beneficial when different design approaches exist.

Finally, hobbyists, makers, and artists could also be encouraged to experiment with affordable four-legged robots and give essential impulses. Since they are neither bound to the scientific method nor economic considerations, they might achieve to find better solutions.

Lightweight

A *lightweight* design is the basis for a low-cost robot since less material is needed. The lightweight construction also supports easier maintenance, handling, and transport. But the most crucial part is the implications on the design itself, as a lighter robot requires lower power consumption. Still, for mobile robots, maintaining battery power is tremendously relevant. Longer-lasting battery power will increase the robot's range and make it more attractive for real-world applications. In addition, a lightweight construction might need fewer resources and helps reduce the ecological footprint, becoming especially consequential when there eventually is an effort for a large production run.

No Rare Parts

For free hardware designs, we should not rely on rare parts or the need for special tools; all materials must necessarily be *highly available*, and ready-made components should ideally be standardized (such as screws). At least, robot components should be available from different manufacturers in more or less the same quality and price, such as DC motors or microcontrollers. This requirement will enable people to build, modify, or maintain the robot. It must always be possible to replace individual parts without changing everything else. And especially when an external manufacturer eventually decides to discontinue a product line, this shall not affect the sustainability of our design.

Keep it Simple

We should prefer *simplistic* designs, which comprise fewer parts and are easy to assemble because it will lower the threshold for people to experiment with the hardware and might encourage them to modify and repair. Whether schematics or source code: Complexity increases over time in every project. It is always easier to add features than to remove them. Overloaded designs are the result and make everything more complicated. Reducing a design to the minimum required components is the most challenging task.

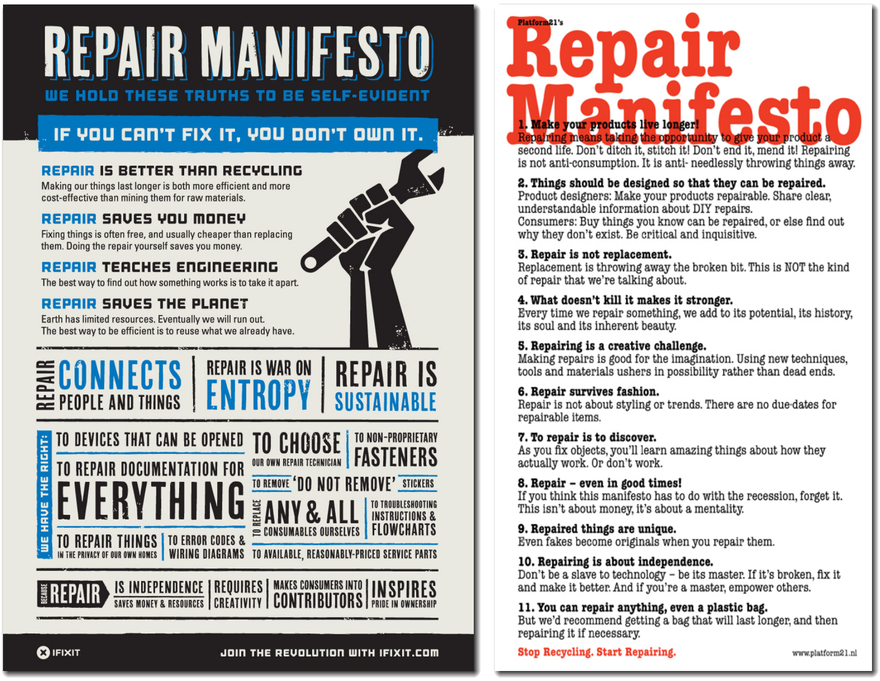
Repairability and the Right to Repair

Disclosure of construction designs and schematics does not guarantee improved repairability, but it is crucial for sustainable economics and resource protection. Easy access to spare parts or the ability to produce them can extend the lifespan of devices. Repairability should be actively promoted and demanded, as it requires effort from manufacturers. For example, batteries often have a shorter lifespan than other electronics. So batteries should be replaceable, preferably in standard forms, regardless of the manufacturer. It also allows customers to take advantage of advancements in battery technology and use more environmentally friendly batteries when available.

Repairability may concern parts, tools, diagnostics, documentation, or firmware. Ideally, a device should be easy to repair, giving end-users and independent repair providers access to original spare parts and tools at fair market conditions. Repairs should be possible by design and not hindered by software, and manufacturers should communicate the repairability of their devices. Several organizations speak out for a right to repair and have formulated their «Repair Manifesto» as depicted in Figure 6.1 with the message that repair is to be put first before thinking of recycling. They urge the people to demand their *Right to Repair*.

Reusability

One thing that highly boosts my minimalism is to step towards reusability. Initially, I made the *Hannah* four-legged robot, but with these parts, I could derive completely different robots. The electronic components and the software are free of specific assumptions about the number of limbs within reasonable boundaries. Only the mechanical components partly need to be modified. Many designs are thinkable. Though we must



REPAIR MANIFESTO

WE HOLD THESE TRUTHS TO BE SELF-EVIDENT

IF YOU CAN'T FIX IT, YOU DON'T OWN IT.

REPAIR IS BETTER THAN RECYCLING
Making our things last longer is both more efficient and more cost-effective than mining them for raw materials.

REPAIR SAVES YOU MONEY
Fixing things is often free, and usually cheaper than replacing them. Doing the repair yourself saves you money.

REPAIR TEACHES ENGINEERING
The best way to find out how something works is to take it apart.

REPAIR SAVES THE PLANET
Earth has limited resources. Eventually we will run out.
The best way to be efficient is to reuse what we already have.

REPAIR CONNECTS PEOPLE AND THINGS | REPAIR IS WAR ON ENTROPY | REPAIR IS SUSTAINABLE

WE HAVE THE RIGHT:
TO DEVICES THAT CAN BE OPENED
TO REPAIR DOCUMENTATION FOR EVERYTHING
TO REPAIR THINGS IN THE PRIVACY OF OUR OWN HOMES | TO ERROR CODES & WIRING DIAGRAMS
TO CHOOSE OUR OWN REPAIR TECHNICIAN | TO NON-PROPRIETARY FASTENERS
TO REMOVE 'DO NOT REMOVE' STICKERS
TO REPLACE ANY & ALL CONSUMABLES OURSELVES | TO TROUBLESHOOTING INSTRUCTIONS & FLOWCHARTS
TO AVAILABLE, REASONABLY-PRICED SERVICE PARTS

BECAUSE REPAIR IS INDEPENDENCE SAVES MONEY & RESOURCES | REQUIRES CREATIVITY | MAKES CONSUMERS INTO CONTRIBUTORS | INSPIRES PRIDE IN OWNERSHIP

IFIXIT

JOIN THE REVOLUTION WITH IFIXIT.COM

Platform21's

Repair Manifesto

1. Make your products live longer!
Repairing means taking the opportunity to give your product a second life. Don't ditch it, stitch it! Don't end it, mend it! Repairing is not anti-consumption. It is anti- needlessly throwing things away.

2. Things should be designed so that they can be repaired.
Product designers: Make your products repairable. Share clear, understandable information about DIY repairs.
Consumers: Buy things you know can be repaired, or else find out why they don't exist. Be critical and inquisitive.

3. Repair is not replacement.
Replacement is throwing away the broken bit. This is NOT the kind of repair that we're talking about.

4. What doesn't kill it makes it stronger.
Every time we repair something, we add to its potential, its history, its soul and its inherent beauty.

5. Repairing is a creative challenge.
Making repairs is good for the imagination. Using new techniques, tools and materials ushers in possibility rather than dead ends.

6. Repair survives fashion.
Repair is not about styling or trends. There are no due-dates for repairable items.

7. To repair is to discover.
As you fix objects, you'll learn amazing things about how they actually work. Or don't work.

8. Repair – even in good times!
If you think this manifesto has to do with the recession, forget it. This isn't about money, it's about a mentality.

9. Repaired things are unique.
Even fakes become originals when you repair them.

10. Repairing is about independence.
Don't be a slave to technology – be its master. If it's broken, fix it and make it better. And if you're a master, empower others.

11. You can repair anything, even a plastic bag.
But we'd recommend getting a bag that will last longer, and then repairing it if necessary.

Stop Recycling. Start Repairing.

www.platform21.nl

Figure 6.1: Repair Manifesto published by the organziations Ifixit and Platform21. Their message is clear: Repaired things are beautiful and sustainable. People should demand that products get designed to allow repair.

adapt parts of the structure, we can reuse many components and methods. For the pulleys, simply a reparameterization will suffice in most cases. For instance, in Figure 6.2, a bipedal robot variant, *Gretchen*, is shown as a humanoid test platform for walking experiments, which I created with our team at Aibrain from existing mechanical drawings and identical materials, methods, and electrical components within only a few months.

I started with constructing the *Tadpole* robot in hardware to continue the experiments from subsection 5.2.3 later on. But it is currently a first draft since it has no built-in power supply and is still much too big.

Another example is a Cajón-like wooden sound robot called *Monkeydrummer*, equipped with Sensorimotor boards driving solenoids instead of motors. It employs the bus system and is expandable by additional PWM-driven sound sources. It is an early prototype of a cognitive robot drummer getting inputs of external sounds via microphones and analyzing the incoming signal for rhythmic patterns. The robot shall join with drumming beats according to the estimated speed and style to accompany a musician.

And finally, the most recently completed robot is *flatcat*, a pet-like robot that reacts to touch. The robot *flatcat* has a simplified caterpillar-like body with three Sensorimotor joints applying CSL control and has an extra Energymodule presented in section 7.3.

All four robots share the same sensorimotor bus system and even some of the same mechanical components. From my work on robots so far, I could deduce recurring requirements, which I will summarize here in the next section, and that might help the reader retrace my design decisions in electronics and hardware elements.

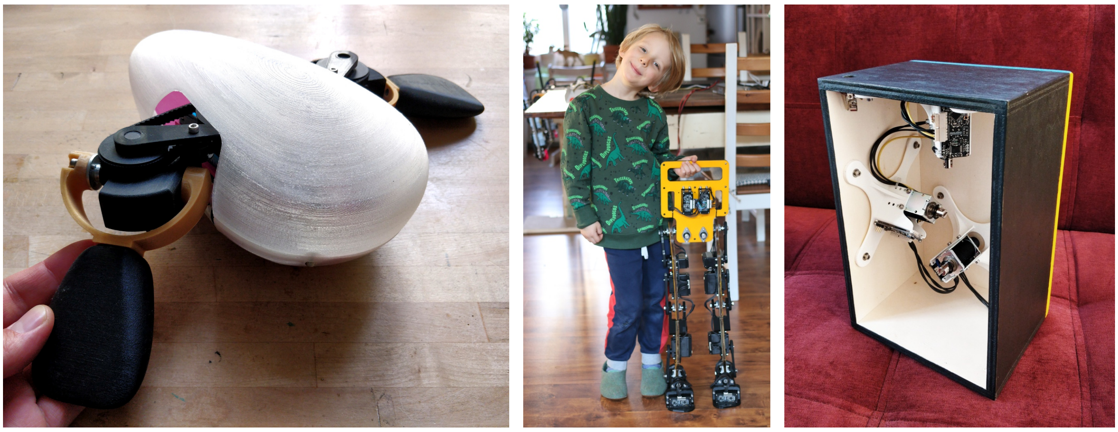


Figure 6.2: From left to right: The *Tadpole* robot as a hardware draft. The *Gretchen* bipedal robot next to my 5 year old son Elmar (for size comparison). The *Monkeydrummer*, a Cajón-sized, sound-generating robot using the distributed motor system.

6.2 Requirements for Walking Robot Drives

The actuators must have several essential features to be suitable for walking robots. They must be as light as possible and, at the same time, strong and fast enough. In addition, desirable features are low noise, low wear, low price, and simplicity.

Brushless motors have become more affordable in recent years due to increased demand in industries such as drones and electric vehicles, making them a potential option for my application. However, the cost and complexity of efficient field-oriented controllers still pose a challenge and conflict with my simplicity and affordability requirements. Available open-source brushless controllers, like ODrive and VESC, are potential candidates but must be adapted and integrated with the corresponding firmware, motor, and power requirements for my specific domain. For this reason, I chose to use DC motors in the first step. However, I am considering building a Sensorimotor BLDC variant for future iterations of *Hannah*.

As will get clear in the mechanical design chapter, the total weight of the robot is crucial since it directly influences the spectrum of possible applications for this technology. The motor position decides how much torque is needed to swing the legs back and forth. For low dynamic motions such as standing up from the ground, the motors have to lift the total mass of the robot, probably on an energetically expensive trajectory. When gears are needed, we better stick to metal gears for low friction, robustness, and good back-drivability, but the motors' cases can be of lightweight plastic when robust enough, such as ABS.

We do not need complete revolution actuators for walking robots. None of the controllers discovered in the evolution chapter need more than a 90° moving angle. Even for standing-up motions and the robot resting in a compact configuration for transportation, 180° per joint will mostly suffice. We can reduce price and complexity when employing simple potentiometers as angle encoders and do not need magnetic or optical encoders. We are not in the domain of high-precision positioning tasks, and the highly dynamic motions created by the gait controller work with moderately low encoder resolutions of 10 bit.

The speed of the actuator is crucial in our application. Consider a complete swing cycle as moving 90° back and forth. The average cycle frequency must approximately reach the 2 Hz domain to walk fast enough. So we require motors that can move the 180° in less than 500 ms under load conditions, including braking and accelerating in the return point.

Numerous DIY robot kits include inexpensive servomotors as illustrated in Figure 6.3. However, the closed-loop robot walking controllers discussed in Chapter 3 are incompatible with these servos due to the absence of feedback. Additionally, these systems primarily focus on position control only, which I found unsuitable in this context. Nevertheless, they often partly possess the desired properties, such as a powerful DC motor, adequate torque-to-speed ratio, and back-driveability. Furthermore, they are encased in lightweight yet robust engineering plastic, equipped with metal gears, and generally incorporate a position sensor, often in the form of rotary potentiometers.

Required for the gait controllers I discovered so far is a fast closed-loop control cycle, so I wanted the ability to update the target voltage setpoints a hundred times per second and get back the current sensory feedback. Ideally, all motors apply their new setpoints nearly synchronously and report their sensor data on request. Data should be transmitted digitally via symmetrical bus systems such as RS485 to ensure robust data transmission. And because we have many motors to control, all servos should share a common communication channel, at least all motors per leg.

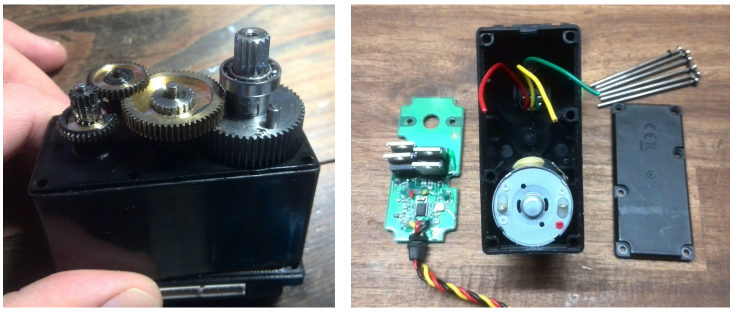


Figure 6.3: The opened gearbox of the employed low-cost servo (left). Disassembly of the servo motor (right). For the *Hannah* robot, I disassembled 12 pieces of this *Hitec HS-805MG* servo motors by desoldering the original electronics for later inserting my custom control boards.

Why Create Another Servo?

To meet these requirements, we typically delve into the realm of *smart servos*. However, using readily available smart servos like the *Robotis Dynamixel* does not align with my approach, even if they satisfy all mechanical, electronic, and firmware criteria. My objective of having free and open-source hardware and software for the robot necessitates that the robot's actuators be equally open to customization, just like the mechanical design. As the motors play a crucial role in the electromechanical design, we cannot rely solely on what the manufacturer deems useful for us. Depending solely on a single manufacturer's product would likely limit our personal freedom, for instance, by impeding firmware modifications for implementing custom control loops such as the CSL directly on the motor's microcontroller or by restricting the ability to reduce protocol overhead for faster communication in tighter control loops.

Whatever modification we need, if the robot's motors are not open-source, which is mostly the case, we would have to reversely engineer the motor's schematics and then write and deploy our custom firmware. Naturally, this comes with losing the warranty and inducing the risk of irreversibly damaging the hardware. I did not want to go that way and decided to build an open-source smart servo myself.

When considering a custom servo controller for standard low-cost motors, there are two potential approaches. Firstly, we can employ basic servos and replace their motor electronics. Alternatively, we can utilize regular gear motors along with an external position sensor to create our own servo. The essential components of such a servo controller include an H-bridge for driving the DC motor, a communication interface for connecting to the bus system, and a simple microcontroller for managing control loops and communication. Additionally, it would be advantageous to integrate various types of analog or digital sensors, such as position, current, or temperature sensors.

An open-source firmware completes the set of requirements. The firmware is ideally easy to modify and allows everybody to implement enhancements. We want servos dedicated to walking robots. Hence, we can employ the most minimal electronics and firmware helpful for the case. We can do without the complete feature set provided by commercial smart servo motors.

Robust Industrial Bus System

The real-time communication system used for the electronic components, described in the following chapter, is realized as a half-duplex RS485 bus with multiple participants. Why did I prefer this bus system? If perfectly aligns with the given design criteria: it is simple, robust, and standardized. There are many pin-compatible transceivers, and the protocol undefined, so there is freedom to design a unique one using only a few bytes.

Electrically this bus system employs symmetrical data transmission, i.e., the digital signal gets transmitted using two wires carrying the same data bits but with inverted logic levels. The non-inverting channel is called A, whereas the inverting one is called B. This technique is robust against interference since each transceiver subtracts the signals $B - A$, doubling the amplitude and removing any additional noise. When both wires are close, we can assume that the noise originating from external sources is approximately identical on both channels, so subtracting removes the significant part. Thus, the data wires must be assembled as twisted pairs[3] to ensure a constant impedance of the transmission line. I summarized the operation of RS485 in Figure 6.4.

A third wire with a *common mode* or reference ground accompanies the two signal wires. The bus has a line topology called a daisy chain, with each participant tapping this line. In such daisy chain connections, the tappings must be kept as short as possible, and we need to provide two jacks on each transceiver so that the tracks on the circuit board can be close to the receiver. Both transmission line endings need termination with a resistor, R_T, to avoid unwanted signal reflections. The resistors' value must match the

[3] Due to the twisted pair assembly this communication bus is literally a *cord*.

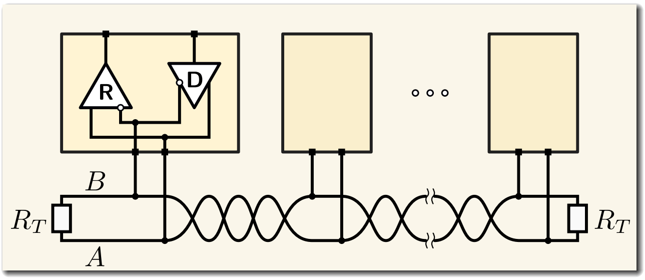
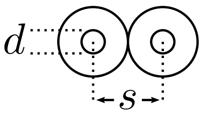


Figure 6.4: A typical RS485 communication: Symmetrical data transmission using twisted pair wires of constant impedance, labeled A and B, carrying inverted logic levels. Each transceiver subtracts $B - A$ effectively doubling the amplitude while eliminating noise even as the wires move in conjunction with the robot's motion.

characteristic impedance of the transmission line, Z_0. The impedance is independent of the cable's length but depends on the cable's specific geometry. Impedance is typically around 120 Ω for twisted pair arrangements but changes with the wires' separation, s, their conductor diameter, d, and the substrate dielectric e_r.

$$Z_0 = \frac{120}{\sqrt{e_r}} \cdot \ln\left[\frac{2s}{d}\right] \tag{6.1}$$



Hence, we need to keep the distance and shape of both wires consistent over the full transmission line. Gently twisting the cables helps to guarantee consistency. The termination resistance influences the signal quality concerning reflections. In practice, we inspect our signals with an oscilloscope and adjust the termination resistance until we neither have ringings nor slurred ramps. 120 Ω has shown to be a good starting point for the fine adjustment with a trimmer potentiometer. Higher resistances have less energy consumption in the transceivers, which is better for battery-powered robots. So using higher-than-needed values is acceptable, provided the transceiver accepts the signals. For instance, using $R_T = 120\,\Omega$ will create a current consumption of 55 mA at 3V3 signal levels with a resulting power loss of 181.5 mW per communication channel. However, receivers tolerate slight reflections, so we can select subcritical termination resistors of 220 Ω, which reduces the consumption to 30 mA (99 mW) and provides sufficient termination for our application.

Conclusion

In conclusion, I explored hardware design criteria for walking robots that I value as essential. After discussing free design hardware and repairability, I examined the limitations of low-cost servomotors and highlighted the need for open-source and customizable smart servos. I discussed the possibility of developing custom servo electronics with integrated feedback and real-time communication. With these criteria in mind, we are prepared to venture into the next chapter, focussing on robot electronics specifically tailored for legged machines. We will explore the design and implementation of electronic systems that enable efficient locomotion.

7 Electronic Modules for Legged Robots

In this chapter, I will discuss the electronic and software concepts utilized in my prototype robots. The developments focussed on designing robots capable of autonomous movement. Consequently, I selected sensors, computation units, and actuators based solely on fulfilling this purpose. The design of these components adheres to the criteria outlined in the previous chapter, emphasizing reusability, repairability, and simplicity. The electronic building blocks get employed in creating various robot morphologies with limbs, each consisting of joints driven by a single actuator. The proprioceptive sensors described here perform the measurements required by the locomotion algorithms. Not discussed here are sensors related to localization or obstacle avoidance.

The electronic architecture of our robots consists of three major components: *Sensorimotors*, *Limbcontrollers*, and *Energymodules*. Sensorimotors combine actuation and sensors within a single unit, one module for each of the robot's joints. For instance, a single leg of *Hannah* currently has three joints. Each employs sensors for position, temperature, and current, accompanied by DC-servo-based actuators encapsulated into smart-servo units, the Sensorimotors. Multiple such actuation units drive a leg in cooperation and connect to their host, a Limbcontroller in the case of a multi-limb robot or any bus-connected computation device otherwise. Limbcontrollers are computational devices that function as hierarchical elements. Each is dedicated to an individual leg and handles communication and power distribution. Energymodules encompass in-system battery charging and power management functions. Currently, I utilize the Energymodule only in the *flatcat* robot, but I plan to adapt it to use it in *Hannah* as well.

This chapter is structured simply with dedicated sections for each building block, providing a comprehensive overview of their functionality and integration within the robot prototypes.

7.1 Sensorimotor

The *Sensorimotor* is an open-source networked motor controller in a bundle of electronics and firmware, primarily developed within the scope of this thesis and optimized for the application of robots with limbs. It comprises a fully integrated H-bridge[1] for driving DC motors, a failsafe RS485 transceiver for communication, and a microcontroller for the computations. It has onboard sensors for temperature, current, and supply voltage. Furthermore, it has safety features like reverse polarity protection and transient voltage suppression for electrical robustness. An externally placed potentiometer connects to

[1] An H-bridge comprises four transistors (in an H-shape) and inputs used to control the direction of current flowing through a load such as a DC motor.

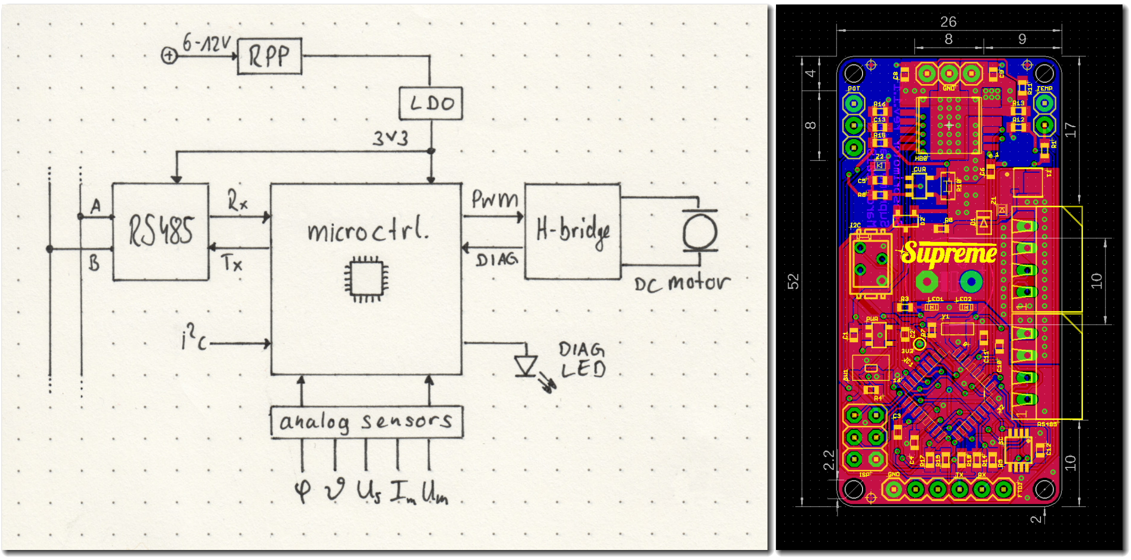


Figure 7.1: Block diagram of Sensorimotor (left), the printed circuit board layout (right).

the module as an angle sensor. The circuit board of the first revision provides an I^{2}C connection for additional external sensors, such as an acceleration sensor. The most recent version comes with a touch sensor connected to the motors' metal housing.

The 8-bit ATmega328P(B) microcontroller operates at 16 MHz with a ceramic resonator. The microcontroller sends and receives communication packages, calculates motor control loops, sets PWM values, and reads analog and external sensor inputs. The user can program the initial firmware using a standard 6-pin in-system-programmer (ISP) adapter and can update the firmware via the bus with a custom bootloader.

The fully integrated H-bridge receives inputs for enable, PWM, and direction. It has higher-level features such as internal overtemperature protection and open-load-detection (if no motor is connected). The H-bridge on the Sensorimotor revision 1.3 used in *flatcat* allows for better coasting of the DC motors and is hence even better suited for using CSLs due to improved back-driveability. Furthermore, the firmware can perform variable frequency PWM ranging from 20 Hz to 100 kHz, as demonstrated in Chapter 9. This feature helps with friction reduction as discussed in subsection 1.2.2 when we drive motors with very low frequencies. Alternatively, the user can create simple sounds with the motors. However, variable frequency PWM may significantly impact motor and H-bridge performance, especially at the edges of the frequency range.

The input voltage range of the Sensorimotor is roughly 5–12.6 V. A low-dropout linear regulator provides the necessary 3V3 for the microcontroller and digital components. The input voltage is passed unregulated to the power electronics. If we want to restrict the motor to use less than the input voltage, we can reduce the duty cycle of the motor output to lower the average voltage accordingly. The layout of the Sensorimotor, as depicted in Figure 7.1, fits into servo housings of different manufacturers, replacing their original electronics. The reader can find the schematics of both revisions in Appendix A.

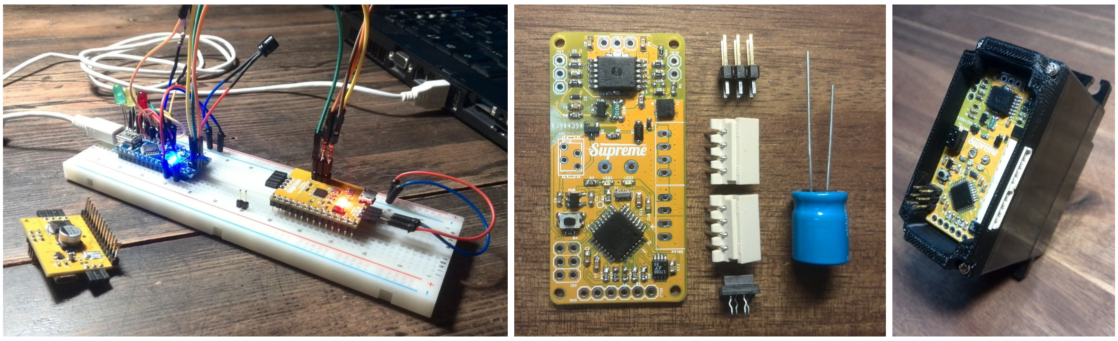


Figure 7.2: Pictures of the Sensorimotor (from left to right): Pre-version while developing firmware using an Arduino as a programmer. The board's revision 1.1 with connectors and bulk capacitor left to be soldered. The Sensorimotor when assembled into a servo.

Overvoltage Input

Many motors will accept higher than the rated voltage for a short time. So we can apply a short-term overvoltage when we provide more than the motor manufacturer's rated voltage. We could think of situations when the actuators of our robot need a little more torque for the robot standing up, for instance. However, since this operation will inevitably increase thermal losses, we must watch the motors' temperature development when we use overvoltage operation frequently in heavy load conditions. Electrical power grows quadratically with the current, $P = I^2R$. So when the DC motor is in stall condition, i.e., has low mechanical power output, most of the input power will get lost by joule heating of the motor's coil. Also, the H-bridge will heat up due to switching and conduction losses, determined by the MOSFET's drain-to-source resistance for the conducting state, $R_{DS(on)}$. For the servo of my choice, 6 V is the nominal voltage. The Sensorimotors built into *Hannah* can shortly provide up to 12 V to the motors in *overvoltage operation*.

Constant Voltage Output

The Sensorimotors measure their input supply voltage. Since the power stage directly connects to the unregulated input voltage, we can adjust the output PWM's duty cycle dynamically concerning the supply voltage to keep the average output voltage independent of the decreasing input supply voltage. This is useful when the battery level decreases or when we use voltage sources of different levels. For instance, let us assume our motor has a rated nominal voltage of 6 V, and we have a decreasing battery supply voltage from 12 V to 9 V. This implies that for full motor power, we must have a duty cycle of 50% when we have charged batteries and 66% duty cycle with low batteries. Hence we can apply motions that behave the same independently on different battery levels. However, this is just an approximation due to the PWM type of the driving voltage output.

Motorcord

I refer to all motors connected to the same serial bus as the Motorcord. It is physically based on the RS485 standard and connects the Sensorimotors in a single limb to a host.

Only the host initiates communication; the Sensorimotors only respond to requests, never to broadcast commands. Each unit on the bus receives all transmitted bytes, but they only respond to their specific ID and never initiate communication actively. Communication on the Motorcord usually begins with a request from the corresponding host to the first motor.

The host sends requests and receives responses from all connected motors, one after the other. Such host requests include the motor's ID, a command ID, and additional payload data, such as the setpoint for the subsequent cycles' motor PWM. The recipient board responds with an acknowledgment of the respective command ID and sends its status and measurements. Each data packet is terminated with a simple and fast checksum[2] to indicate possible bit errors to the receiver. The Motorcord transmits data with a rate of 1 MBaud.

For communication with the host machine, I wrote a C++ library called *libsensorimotor* that is Linux-compatible and comprises an optional Python wrapper. Further, an embedded C++ implementation completes the software stack. The embedded code currently runs on the firmware of the Limbcontrollers described in section 7.2.

Bootloader

In addition to the firmware, I developed a bootloader for the Sensorimotor boards to be programmed while built into the robot. Often the ISP connector is hard to reach with motors installed, and flashing the firmware on multiple boards via ISP can be time-consuming and prone to errors.

The bootloader is a separate piece of firmware installed in a different flash area of the microcontroller. It gets activated by a special command sent from the host via the RS485 bus. The firmware shuts down and jumps to the bootloader section. Upon obtaining the update command, each board receives a series of data packets containing the new firmware and overwrites the old firmware, resetting the board to start the new code. The Python-based bootload script handles the firmware update process on the host side.

With the bootload script, the host can update all motors at once during paused operation without us having to remove individual motors or use a programming adapter. What a tremendous advantage, especially during a remote SSH session. For example, it allows us to remotely maintain the *flatcat* robot, including firmware updates without physical access to the device.

Evolution of the Board Layout and Self-Manufacturing

An elementary feature of free hardware designs is the possibility of self-production. The Sensorimotors of rev. 1.1 and 1.3 have SMD components that we can still assemble by hand. However, soldering with a regular iron I experienced was time-consuming due to the compact layout required, so I recommend reflow soldering. Devices like tweezers, paste, stencils, and ovens are available for home use for little money, with the help of

[2] The algorithm adds up all of the message's bytes, discarding any overflow bits. The checksum is the two's complement of this sum and gets appended as the last byte of the package. For message verification, the receiver follows the same process of adding all the bytes, including the checksum, and again discarding overflow. If the resulting sum is not zero, an error must have occurred.

which we can assemble even dozens of boards in a few days with some practice. These days, we usually[3] let the board production to the factory, and the automatic assembly has become so cheap that even hobbyists can enjoy fully automatically produced boards.

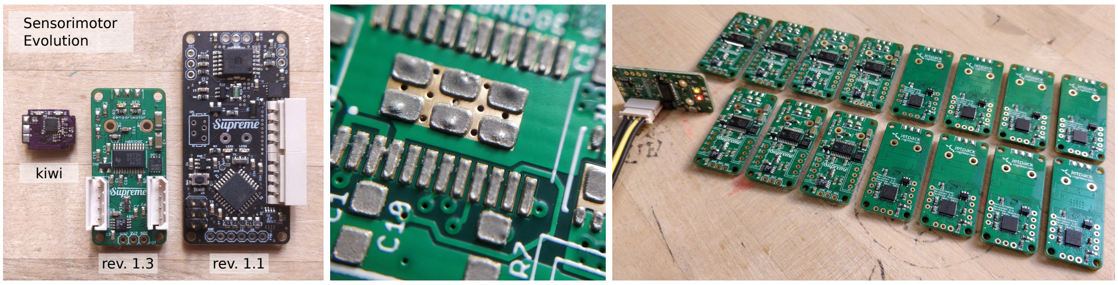


Figure 7.3: Left: The Sensorimotor Evolution. Rev. 1.1 is the basis for *Hannah* and *Gretchen*, whereas rev. 1.3 got installed in *flatcat*. The missing rev. 1.2 used the new form factor but still had the same components as 1.1. I need to discard it due to the chip shortage in 2020. The most recent Kiwi version is still in development, and I plan to integrate it into micro servos for space-constrained applications. Center: Solder paste (an emulsion of tiny solder balls and flux) applied with a stencil. Right: A freshly soldered batch of rev. 1.3 boards.

7.2 Limbcontroller

Limbcontrollers represent a hierarchical element in a robot between the host computer and the Sensorimotors. For example, if the robot has multiple legs, each with several motors, it makes sense to divide them hierarchically to maintain the update rate and reaction speed. For instance, *Hannah* has four legs with three servos and one Limbcontroller each.

Limbcontrollers communicate with each other on a high-speed RS485 bus, called the *Spinalcord*, and speak to each leg's motors on separate Motorcords. The Limbcontroller is a gateway for sensory information captured from the legs and motor control setpoints sent to the legs. Communication between the host computer and the spinal cord is separated to prevent timing constraint issues that may happen when we establish real-time communication with a multi-threaded embedded OS. For this purpose, a Limbcontroller has multiple RS485 interfaces.

The microcontroller selected for the Limbcontroller is a 32-bit STM32F411 Arm Cortex M4 which contains a separate floating point unit and multiple serial interfaces. Three full-duplex serial USART interfaces are converted to half-duplex using RS485 transceivers: The first interface is used for the Spinalcord connection and transmits data

[3] During the Covid-19 pandemic, we experienced a worldwide semiconductor shortage exceeding two years, so many integrated circuits were no longer in stock, and many others would soon get sold out (approximately half of my material list). For producing the 60 Sensorimotors and 20 Energymodules for the *flatcat* robots, I had to urgently buy components I never used before and do a manual assembly, picking and placing several thousand millimeter-sized parts by hand.

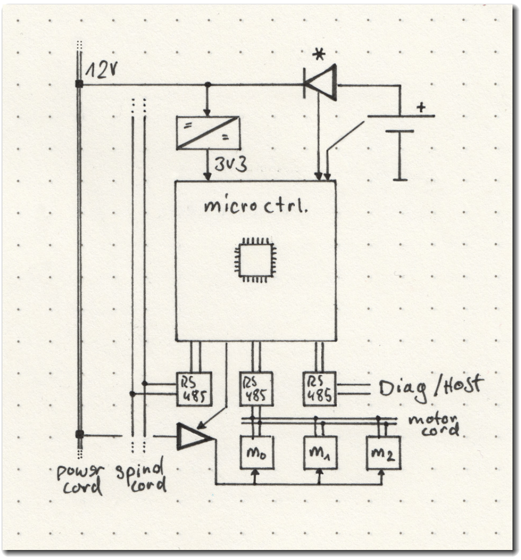


Figure 7.4: The functions of the Limbcontroller: An ideal diode circuit connects the local leg battery to the power cord. The microcontroller operates three serial interfaces to communicate with other limbs, with the motors, and with the host. The microcontroller can connect/disconnect the motors from the powercord, read the battery voltage, and can detect which voltage source is currently selected.

with 3 MBaud. The second one is the Motorcord which connects to the Sensorimotors of the respective leg. The third one is optional for diagnostics or connecting the host.

The Limbcontroller adopts a simple but effective power management circuitry from the electronics of the Myon robot showcased in subsection 1.3.1, using a so-called power path controller. Motors and Limbcontrollers get their power either from a local battery or an external source, such as batteries connected to other Limbcontrollers. At least one Limbcontroller battery is required to be connected. However, I recommend installing several batteries for equal load distribution and reduced battery stress. The Limbcontrollers synchronously discharge their batteries by consistently tapping the highest connected voltage. I realized this using the power path controller LTC4412 in ideal diode mode. The Limbcontroller measures battery voltage, detects undervoltage, and reports the currently used voltage source. The circuit board is designed for higher current flow and has divided copper areas for powering Sensorimotors and communication, as we can see from the board layout depicted in Figure 7.5.

Spinalcord

The real-time communication between multiple Limbcontrollers is called Spinalcord. The concept was first introduced as the communication basis of the Humanoid Robot Myon published in Hild et al. (2011). All Limbcontrollers share a set of data arranged in a simple black-board architecture, i.e., each board provides sensor measurements or internal state values to all other controllers as the basis for subsequent motor values. Hence, the robot's data gets distributed to all other units of the Spinalcord. Each Limbcontroller must provide the motor setpoints for its associated limb and sends the new target values to its Sensorimotors.

The Limbcontrollers report their data independently. In contrast to the Motorcord, we need no initial request from a host machine. Instead, the boards report their data in the order of their unique ID. Initially, after reset, when no communication has started, the

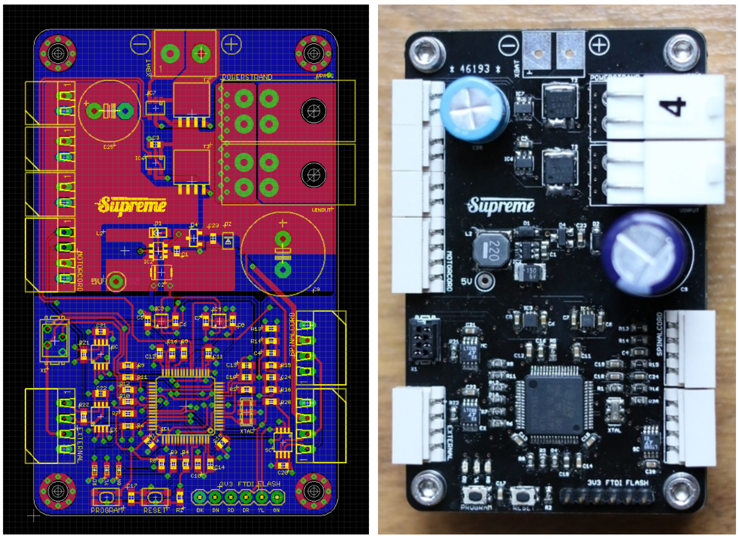


Figure 7.5: The Limbcontroller board layout: the upper part of the PCB comprises the power sections with large copper areas to handle the high current needs of connected Sensorimotors and to daisy-chain the power cables. The lower part includes the logic section with all communication interfaces and the microcontroller.

board with the lowest ID initiates the data transfer so the others can enqueue their data according to their respective time slot. If the lowest ID does not start communicating, the board with the next to lowest ID waits for a defined time and then initiates the communication alone. This way, the Limbcontrollers can be disconnected and reconnected during operation.

For host communication, the Spinalcord transmits data between limbs and the host back and forth with the help of a separate Limbcontroller board denoted as the *trunk*. This board is technically identical but it dedicates to managing serial communication with a potential non-real-time capable computation unit.

In the Spinalcord, each data point has a fixed size of 16 bits. Data gets arranged in columns dedicated to the individual board. In the philosophy of Spinalcord communication, there are no explicit commands, just status messages of individual units that get distributed to all the others. However, some status messages of the trunk implicitly encode the desired controls or setpoints.

Tree Hierarchies

In its current form, it would be possible to use the hardware in the context of smaller tree-like architectures by using Limbcontrollers as hosts for sub-level Spinalcords. However, this approach does not scale arbitrarily to communicate all sensorimotor data up and down from root to leaf. In such a setup, the local Limbcontrollers should operate the control loops independently, and their respective higher-level structures should only set control parameters so that the sensorimotor loops remain local and thus reactive. The delays introduced with each new level in the hierarchy would otherwise lead to decreased update rates hindering reactive control for walking robots.

7.3 Energymodule

This section describes the Energymodule, a separate electronics, that is currently only used in *flatcat*. The focus of this section is on the technical description only. The behavior part derived from what is possible with this module I explain in Chapter 9. The Energymodule has the following tasks in a robot: battery management, status data report, power on/off control, and managing bus voltage.

The Energymodule's microcontroller section is almost identical to the Sensorimotor's and uses the ATmega328PB with an RS485 bus interface. It allows for sending vital status data to the host, initiating robot shutdown, and monitoring bus voltage and button status. Using the same core components also enables the adoption of the Sensorimotor's bootloader, simplifying firmware updates in the installed system. The Energymodule includes integrated circuits for battery charging and protection, a supercapacitor with a current limiter, a boost converter, and a MOSFET-based soft switch for enabling and disabling the bus voltage to power the motorcord.

Battery Management

An Energymodule connects to several parallel battery cells. For instance, in *flatcat*, three cells achieve eight hours of battery runtime. Each lithium-ion cell has a built-in protection circuit and a voltage range of 2.5–4.2 volts. I prefer using the 18650 standard form, so cells are easily replaceable, and non-technical users can perform replacements if needed. Using standard-form batteries, we might enjoy longer availability of replacement parts, promoting sustainability.

A dedicated circuit protects all connected cells against overcharge, over-discharge, overcurrent, and overvoltage. Additionally, a chip fuse protects the entire system. The cells charge in parallel with a maximum total current of 1 A. The slower charging rate increases battery life and is a more sustainable approach[4].

Two LEDs indicate the charging process, and the Energymodule's microcontroller is informed about a connected charger through input pins, e.g., to inhibit a robot's motions while charging. An output line can temporarily inhibit the charging process required to measure the battery voltage. The system charges via a standard USB Type-C socket. In particular, due to its low current demand, it is therefore compatible with most charger devices.

Sensors and Controls

The Energymodule's sensor data includes battery voltage, limiter current, supercapacitor voltage, time to close the main switch, button status, and status flags. The control data sent to the Energymodule by the host includes the desired target bus voltage and the command to initiate a self-power-off. The power/pause button is the only immediate user interface: short press for pause/resume, long press for powering off.

[4] This approach also aligns with the idea that I designed *flatcat* not to be a machine that is always available but would instead take a significant amount of time to recharge, similar to a sleeping pet.

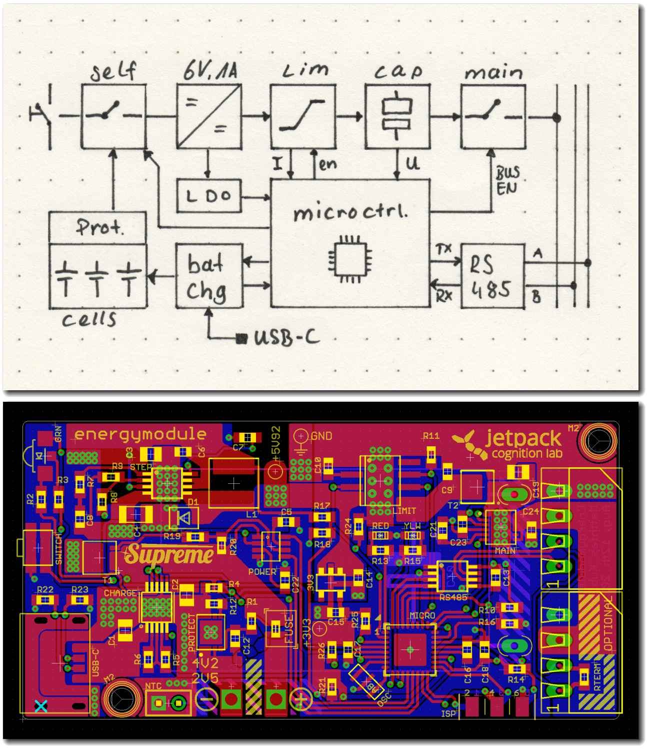


Figure 7.6: The Energymodule explained: The microcontroller interfaces with the host via RS485. The module provides battery life, bus voltage, and current measurements and receives setpoints for target bus voltage. The board includes a self-power on/off switch that gets either operated by pressing a tactile button or by a self-power-off request of the host. Parallel Li-ion cells with 3V7 nominal voltage and a dedicated protection circuit ensure a safe and reliable power supply. A battery charger circuit recharges the cells with 1A via USB-C. A step-up converter (6V, 1A) charges the supercapacitor through a microcontroller-operated current limiter. The limiter also provides a current measurement. The capacitor provides backup power for flattening peak demand. The bus voltage, managed by a main switch, can be dynamically regulated, offering flexibility in bus voltage levels.

Shutdown

When the host is an embedded Linux System-on-Chip (e.g., a Raspberry Pi Zero) with an SD card as storage and writes log data, performing a clean OS shutdown is mandatory to give the memory controller time to flush the data. So when we desire to power off our robot, we can press the tactile button for a few seconds to initiate the shutdown. After signaling a soon shutdown to the host, the Energymodule waits for 10 seconds and then disables the bus voltage to power off the Sensorimotors and the host. Afterward, it initiates a self-power-off.

Variable Bus Voltage

At the beginning of the hardware part, we spoke about sustainability and simplicity, so I placed particular emphasis on economic electronic components in the design of the energymodule. Accompanied by the need to make the robot's motions safe for the user, it drove me to the decision to buffer the bus voltage with a relatively big 5 F supercapacitor. The buffering is particularly suitable for handling the required current peaks during motor actions at low cell voltages and allowed me to employ a comparatively small-sized, low-current step-up converter. Otherwise, the circuit and its surrounding components would cover a lot of board space solely to provide the necessary peak power of approx. 6A for rare occasions. Charging such large capacitors with less than 1 A takes a few seconds. It creates additional protection by limiting the maximum torque exerted by the motors to short periods, thus making injuries for users very unlikely. In Chapter 9, I will share some thoughts on how this feature can Additionally influence the robot's behavior to be more intuitive to humans.

The Energymodule measures the voltage of the supercapacitor, providing it as a sensor value. Through the limiter, we can regulate the amount of charge going into the buffer to reach the desired bus voltage and thus also influence the robot's behavior. The limiter (also called an e-fuse) is connected upstream of the supercapacitor, is microcontroller-operated for regulation, and automatically prevents an overload of the electronics by inrush currents[5]. The limiter circuit also measures its current flow, provided by the Energymodule as a sensor value. The limiter has a fixed maximum current set to 900 mA, leaving enough headroom for the logic components. The total current must not exceed the recommended 1 A of the step-up converter to reduce the components' thermal stress.

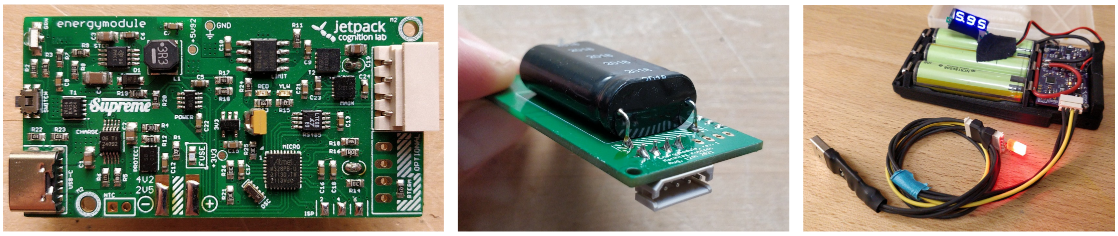


Figure 7.7: The Energymodule viewed from the top and bottom side showcasing the 5 F supercapacitor. Also depicted is the test rig with installed batteries and an LED signaling the active bus voltage.

At the output stage of the Energymodule, there is a MOSFET-based load switch, which can turn the bus voltage on and off. When the buffer has enough voltage to operate the host and the motors, the Energymodule opens the main tap. The firm power reset is crucial since microcontrollers might not boot reliably with a too-slowly rising supply voltage.

Another MOSFET with a gate-source threshold voltage[6] of $V_{GS(th)} = -2V5$ is connected in front of the buffer to prevent a too-deep discharge caused by the Energymodule still supplying itself from the supercapacitor after being disconnected from batteries. But below the microcontroller's brown-out (or reset) voltage, it makes no sense to let the remaining charge go. Therefore the MOSFET is connected in the drain-source direction, acting like a better diode, and keeps about 2V on the capacitor for a comparatively long time. Having a residual charge in the buffer shortens the time for the robot to power up again and might save some energy. Furthermore, since charging up from 0V creates the highest current peak and thermal stress, keeping an offset helps prolong the component's lifetime.

[5] When powering on, an empty capacitor of this size behaves like a short circuit for several seconds. So current limiting is mandatory.

[6] For a metal-oxide semiconductor field-effect transistor, the $V_{GS(th)}$ is the voltage at which the transistor channel begins to conduct.

Conclusion and Outlook

The electronics discussed in this chapter are the basis for the robots *Hannah*, *Gretchen*, and *flatcat*. All three use Sensorimotors for actuation, although in different hardware revisions. The Limbcontrollers naturally empower the hierarchical structure of the four-legged robot, while the Energymodule adds crucial functions to *flatcat*. A unified application of all three components in a single overall system is still pending. We can achieve this by increasing the Energymodule's maximum applicable bus voltage, allowing us to meet the higher torque demands of the large-scale servos. For a potential future version of *Hannah*, I would employ multiple distributed Energymodules locally connected to each limb.

Meanwhile, I started developing a tiny version of the Sensorimotor boards called «kiwi», which we can see in Figure 7.3. I want to use it in micro servos. Despite its small size, I hope I can keep most of the Sensorimotor features so we can use it and later build small-scale robots or maybe a robotic hand from it.

The Sensorimotors equip a robot with distributed actuators and sensors, the Limbcontrollers provide the necessary hierarchy and power distribution, and the Energymodules care for battery management. Thus, these components already cover large parts of a potentially complete robot architecture available for various projects. I kept the definition of the host in this architecture deliberately open to maintain the flexibility of choice. Whether a desktop PC, Linux SoC, or another microcontroller forms the host depends on the respective application. For the last two chapters, I want to demonstrate the application of the proposed components under real-world conditions using two real robots.

8 A Four-legged Robot for Research and Education

> «Why did the quadruped robot need a therapist?
> Because it had trouble finding its footing in life!» (by ChatGPT)

This chapter introduces the concepts of the *Hannah* prototype and shows how the mechanical ideas practically derive from our design guidelines. Since I am not a mechanical engineer, I selected the design aspects concerning the manufacturing techniques available for hobbyists and from the viewpoint of a maker. So reproducing this robot is achievable by almost everyone willing to do this.

When I started the project, I had no idea how to use CAD-based construction methods. So I prototyped the robot's legs with household methods. With the help of individuals like Thomas Lobig and Andreas Gerken, who thankfully contributed parts and ideas to the design of *Hannah* and *Gretchen*, I learned to construct mechanical parts while doing these projects. So the mechanical design of the robots shown in this chapter grew from the awesome teamwork with these two friends.

The chapter is structured as follows: The first section explains the mechanical design, especially for the legs. Section 2 shows how the electronic components from the previous chapter integrate into the robot. Section 3 demonstrates how we could reuse ideas and methods of the quadruped Hannah to derive the bipedal robot Gretchen from it. The last section shows my not yet finished attempts to apply parts of the results from the evolution chapter to these robots.

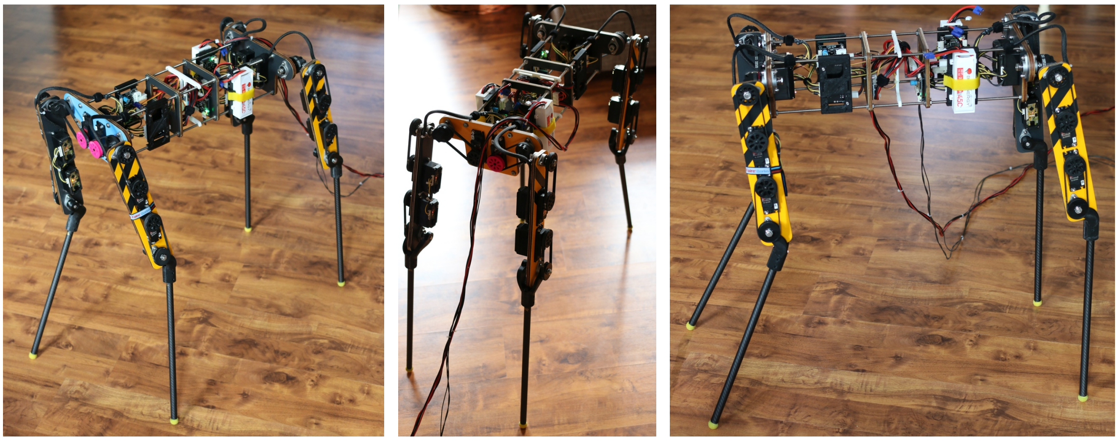


Figure 8.1: The *Hannah* robot from different view points.

8.1 Mechanical Structure

In the first design phase, I prototyped a non-motorized leg structure, seen in Figure 8.2, with available materials to determine the potential range of motion and, above all, to physically experience the weight before continuing the design implementation by electronic means. This way of rapid prototyping has turned out to be valuable in various of my projects. I based it on the assumption that with today's engineering tools, you can only sense your design partly; the feeling of size, weight, or rotational inertia, for instance, gets lost too easily in too much early detail.

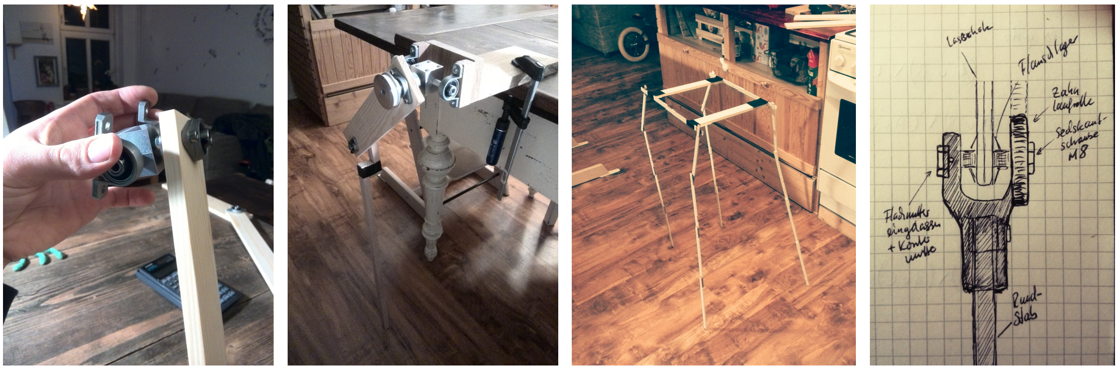


Figure 8.2: Early joint and leg construction drafts and size estimation using a folding ruler.

If we want to make it possible for everyone to make the robot themselves, the cost of the system's hardware must remain low. I planned the robot's structural pieces to be manufacturable with simple laser cutters and 3D printers. The remaining parts, such as bearings, nuts, and bolts, can be purchased at any tool store.

Material for Body Parts

The material of my choice for the robot's structure parts is wood. It is cheap, available, and modification friendly. We can cut easily with high precision on a laser cutter. Furthermore, wood brings some inherent flexibility, a valuable property for compliant walking robots. Wood is lightweight, non-toxic, and renewable. From my viewpoint, it is the perfect material for rapid prototyping and robot experimentation. But the most selling point is that many people already have an intuitive understanding of how to process this material and have a clear expectation of its physical properties. Wood is available in numerous types, with a dense spectrum of properties regarding strength, weight, density, fiber orientation, elasticity, and the like.

The type of wood I selected as the base material for the legs and torso structures is a 5 mm thick plywood multiplex. Similar wood types are possible according to availability, price, or strength. Most laser cutters or CNC milling machines have no problems cutting wood of this thickness. I then glued two layers together to achieve a 10 mm compound piece. Afterward, I colored the pieces with acrylic paint and finished with a clear coat.

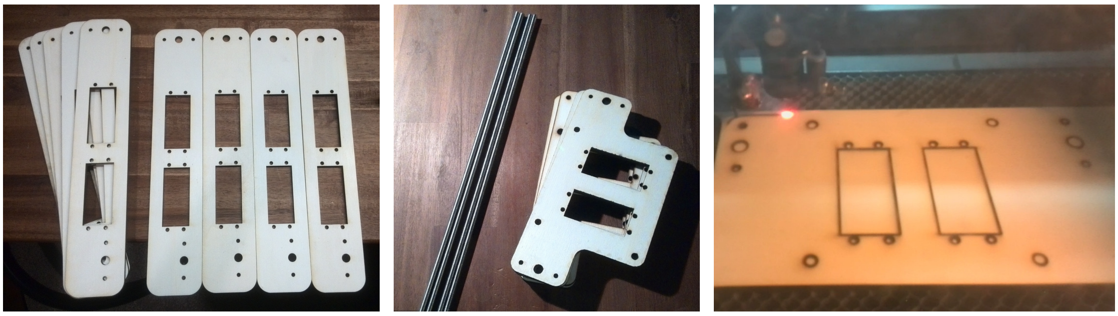


Figure 8.3: The laser-cut wooden pieces for the legs and torso plates.

Leg Layout

When creating legs for walking robots, we should remember: 1) Legs will accelerate a lot, requiring careful distribution of rotational mass. 2) Legs are the robot's interface to the ground and will, therefore, be subject to substantial impacts.

Legs must be light and robust at the same time to move dynamically. All leg-irrelevant masses must be positioned compactly in the torso, e.g., the batteries and the motors for the shoulder and hip roll joints. But components that we must inevitably mount on the legs should be placed with minimized inertia since they will get accelerated with every step taken. Easily but crucial, we can reduce the mass of the lower leg. Since a quadruped does not necessarily need foot joints, no heavy components remain below the knee. We position the motor for the knee in the upper leg above the knee's pitch axis. All other masses in the upper part get raised close to the rotation center of the hip pitch joint, reducing the inertia and the torque needed to bring the leg into motion.

Impacts propagate through the leg and will affect every component mechanically coupled. So we should provide just enough damping capabilities to reduce the effect of shocks but keep the systems dynamic properties. So I wanted to avoid placing motors in the joint centers because forces of different directions will come together in these locations. The motors' gear trains should ideally only experience torques in the preferred shaft axis, with all other forces kept minimal.

Symmetries

The legs of a quadruped usually differ in front and rear, left and right. The front legs comprise the joints for the shoulder and elbow, whereas the rear legs comprise the hip and knee. For simplicity and a low part count, all four legs of *Hannah* intentionally share the same parts, reducing the total costs and facilitating manufacturing and maintenance. My results from the simulation experiments indicated that with identical legs, walking at a moderate speed is possible. Joint axes perpendicular to the robot's reference frame axes work similarly well for walking robots, so I used this convention, too. It is simple to implement and remember. So-called pitch joints move the legs forward and backward, while the roll joints move the legs laterally. Shoulder and hip have both roll and pitch joints, while elbows and knees only have pitch configurations. Ideally, the two-dimensional joints would have axes going through the same point.

8.1.1 Timing Belts, Pulleys, and Bearings

To reduce mechanical stress in the Sensorimotors' gear trains, I wanted to have the motors placed eccentrically to the axis. I found that torque transmitted via belts favors a simplistic design. Belts have lower friction than cables and can transport torque over distances easier than gears. So I employ a separate axis with metal ball bearings which can take more load and protect the motor bearings from heavier force peaks.

Using a timing (or toothed) belt, we decouple the motor from the joint axis and can further optimize the positioning of the actuator. Optionally, we could change the gear ratio by using differently-sized pulleys. A certain serial elasticity and damping are inherent in belts which already provide some protection. Besides, belt gear transmission is robust, silent, and needs little maintenance. It transmits mechanical power without backlash and is highly efficient (up to 98%).

The introduction of belt gear transmission has many advantages, such as serial elasticity and damping for motor protection, decoupling of the motor and joint axis for transmitting torques only, and a more flexible motor positioning. This type of drive using belts perfectly aligns with our other design criteria regarding the friendliness to modification and favors our independence from too specific components.

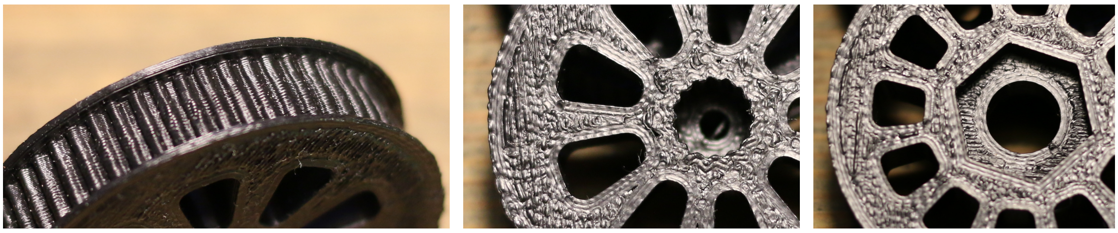


Figure 8.4: Detailed photos of pulley tooth quality, spline adapter, and hex nut pocket.

3D-Printed Pulleys

The wheels used for timing belt transmission, the pulleys, are usually fabricated of aluminum or cast iron. They are widely available, equipped with a simple borehole and fixed on the shaft with a blind set screw. However, metal pulleys are too heavy and needlessly expensive for our robot. Furthermore, I could not purchase them with a native spline connector used by our servo (compare with Figure 6.3). So the price, weight, and compatibility were reasons enough to create our custom pulleys from 3D-printed PLA plastic.

When creating custom pulleys, we can merge the pulleys with other parts as the axis connector. So we have different types of pulleys in our robot: For instance, we have pulleys with motor spline adaptors or axis connectors. Both axes of the hip/shoulder are perpendicular and connect with a 3D-printed part, which also comprises the pulley for the roll joint. We thereby ensure that both axes go through the same point to form an easy-to-model universal joint with minimal offsets.

Compared to other 3D printed parts of this robot, the teeth of the pulleys have higher precision requirements, but it was feasible with usual home printers. The precision of the teeth is crucial since it directly influences the maximal transportation force and the quality of synchronous transportation. The tooth ratio and form must match the belt as close as possible. For each joint, I favored identical distances between pulleys to have the same type and diameter of pulley and belt, reducing part count, cost, and complexity.

Belt Tensioner with Serial Elasticity

A belt tensioner increases the pulley's wrapping angle, avoiding belt slippage. In our setup with servo motors using absolute position encoding, slippage would be disastrous since it would require the frequent removal of the belt for recalibration.

Considering belts, we generally distinguish between the working side (part of the belt under load) and the belt's loose side. Usually, the tensioner gets placed near the driving wheel on the loose side. In our robot, though, the motor alternately drives in both directions. Likewise, the joint can drive the motor when running freely and experiencing external forces. I needed to find a way to create a tensioner that fulfilled these requirements. The resulting device depicted in Figure 3 is a belt-mounted tensioner with different elasticity options for experimentation. Depending on the desired stiffness used, we can introduce serial elasticity in the belt drive if required.

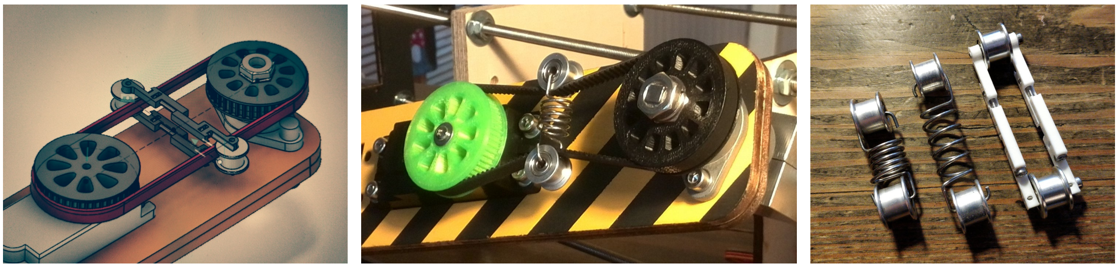


Figure 8.5: CAD drawing and implementation of the belt transmission with different tensioner options: Two tensioners with differently stiff springs and a non-elastic but length-adjustable one.

Standard Parts

Standardized components such as flange bearing KFL08 are employed, which stabilize the 8 mm shafts. I strictly use metal ball bearings for reduced joint friction. However, 3D-printed parts that hold a respective ball bearing in position could easily replace metal-cast flange bearings, allowing for a reduction in weight and unit price.

The chosen shaft is an M8 metric threaded bolt, which is a standardized option that is readily accessible and cost-effective to obtain. Form fit is the preferred option for the shaft to ensure secure torque transmission. We achieve this by utilizing a hexagonal screw head at one end and a 3 mm splint inserted into a pilot hole drilled through the other end of the shaft.

8.1.2 Body Structure

The lower leg consists of a carbon fiber tube mounted to the 3D-printed knee (or elbow) and ends with a custom silicone-cast foot piece. The knee, made of PLA plastic, integrates the pulley, the axle suspension, and the connection plug for the lower leg segment in a single piece. The feet of the robot are kept as simple as possible. Future versions will include force sensors to recognize foot-to-ground contact. The flexible silicone feet permit the integration of a 3D magnetic sensor, allowing the robot to measure the contact pressure applied in the direction of the leg next to the shear forces.

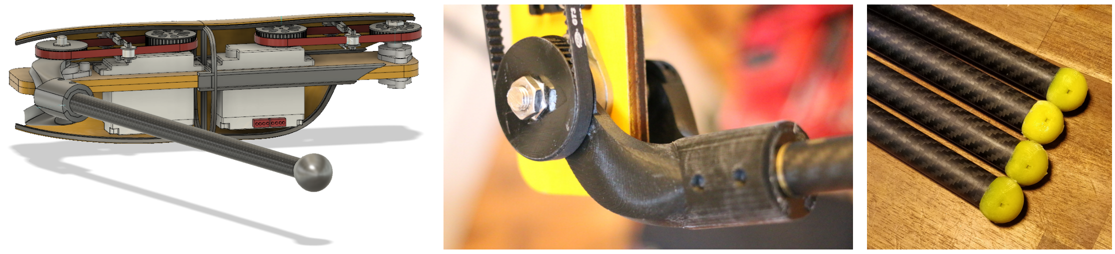


Figure 8.6: CAD model inside view of a leg with, belts, pulleys, shells and Sensorimotors (left). A detailed view on the knee (center) and silicone-cast feet (right).

Torso

The torso is simplistic. It connects the legs, integrates the hip and shoulder motors, and includes the required control electronics and batteries. The torso's layout comprises plywood manufactured with a laser cutter identical to the leg parts as shown in Figure 8.7. Threaded rods connect the wooden shapes to give the construction stability. Assembly comes without additional special tools. The material choice is a compromise between weight and stiffness.

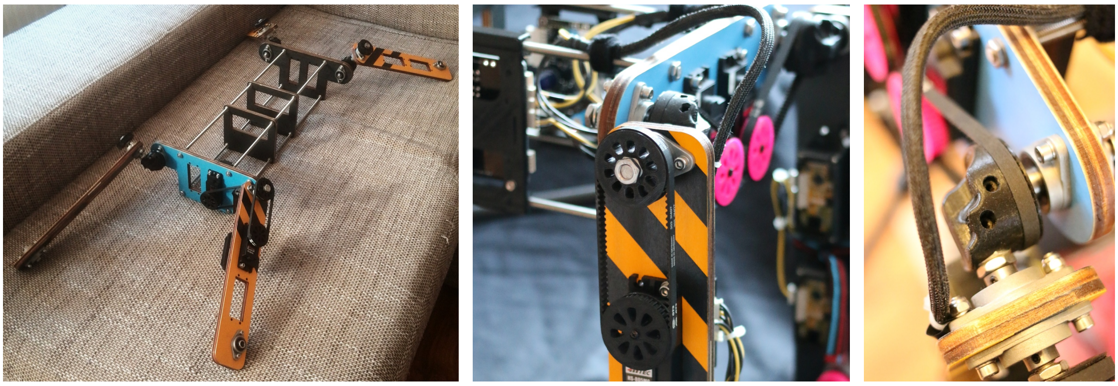


Figure 8.7: Torso during assembly and shoulder joint from different views.

The threaded rods make it easy to adjust the torso's length and to flexibly shape its moment of inertia by freely positioning the masses, such as battery packs. The flexibility for experimentation is essential here since it allows for directly influencing the torso's acceleration concerning each rotation axis. A well-adjusted moment of inertia saves energy and contributes to efficient walking. We can manipulate the passive body dynamics to align with the torso's eigenfrequency, an exclusively passive effect provided by the physics at no additional cost in control. When we have found the optimal inertia, upcoming torso versions can be designed without metal rods but could favor using 3D-printed parts or wood instead.

Shells

The shells of the legs are essential for experimentation since they protect the electronics against mechanical impacts. Also, for storage, transport, and robot handling in general, protective shells are valuable. Therefore, I created a thin 3D-printable shell that wraps around the existing components, such as motors, cables, and timing belts.

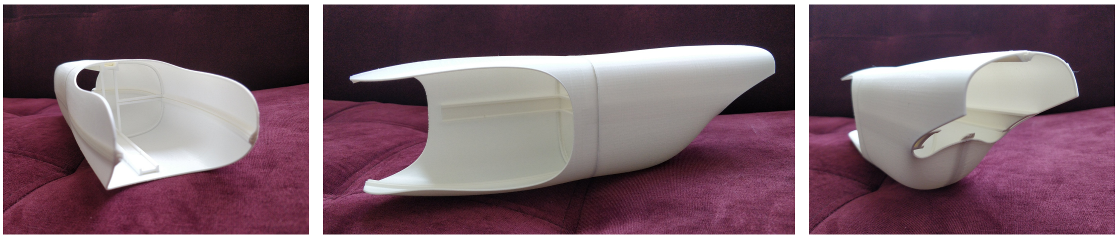


Figure 8.8: Robot shells made from 3D-printed *Tough*-PLA for protection of the leg electronics and belt drive. Assembling shell parts comes without tools because they possess a mechanical snap lock at their edges, connecting the shell halves solidly.

The production of the shells is feasible with usual 3D printers, has low cost, and is fast. It took about 20 hours of printing time using standard settings. They consume 120 g of material for less than 10 EUR per leg since each shell is only 1.2 mm thick. Afterward, we can sand and paint the PLA material as we like. I split the shell halves into smaller segments and later glued them together, as shown in Figure 8.8, to accommodate smaller printers.

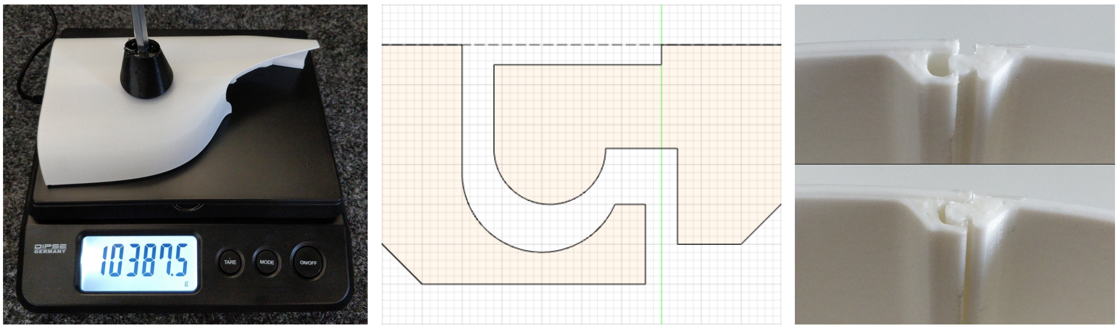


Figure 8.9: Stress testing shells with a 10 kg static load applied to an area of 20 mm x 20 mm (left). The snap closure profile (middle) and details of its print result (right).

I gave the shells a snap closure as seen in Figure 8.9 for a tool-free servicing and refrained from using screws that would otherwise bring extra weight. Double-sided tangential pushing closes the shell, and a slightly firmer radial push on the snap edge opens it up again, each action accompanied by a distinct click sound. The shell parts attach to the wooden part of the leg on the structure's edges with their integrated rails. The Figure 8.10 below shows the fully-assembled robot with colored shells.

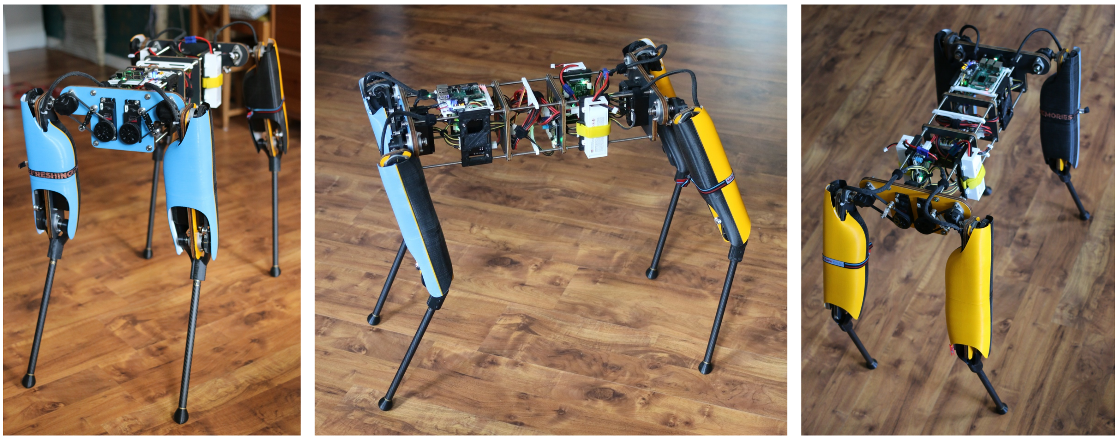


Figure 8.10: Different views on the fully-assembled robot, including shells. In the moment of these pictures, for the first time, the robot is autonomously powered by its batteries and actively maintains its balance by control loops calculated on the robot.

8.2 Electronics and Control

The *Hannah* robot comprises electronic components as extensively described in Chapter 7. It sums to 12 Sensorimotors and 5 Limbcontrollers, up to 5 Lithium Ion batteries, a star-node power distribution, and a Raspberry Pi 4 mounted in the front/top of the torso, running Linux as the host machine. The center of the torso additionally holds a 3-Axis accelerometer that connects to the I^2C port of the Sensorimotor with ID 0. In Figure 8.12, we can see the complete electronic architecture comprising power and communication used for my experimentation.

Batteries and Power Management

The power supply comprises four limb batteries, each powering a single leg. The batteries have distributed locations to fulfill the local power requirements and keep the losses low using short power lines. Locality in the power supply is often used in electronics and avoids unintended interference between components. But depending on the kind of motion, the power demand between legs will certainly not be equal. The passive balancing mechanism of the Limbcontrollers equalizes the state of charge of all batteries. More precisely, we ensure an equalized discharging of the batteries by always using the highest voltage source.

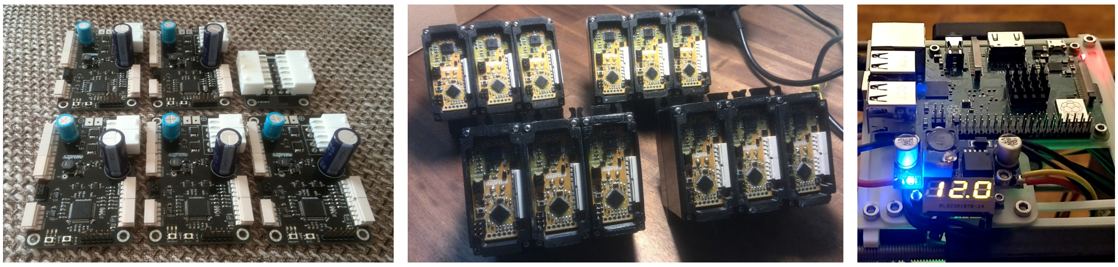


Figure 8.11: Electronic components installed into the robot: 5 Limbcontrollers, 1 power distribution node, 12 Sensorimotors, and a Raspberry Pi 4 using a separate 5V voltage regulator.

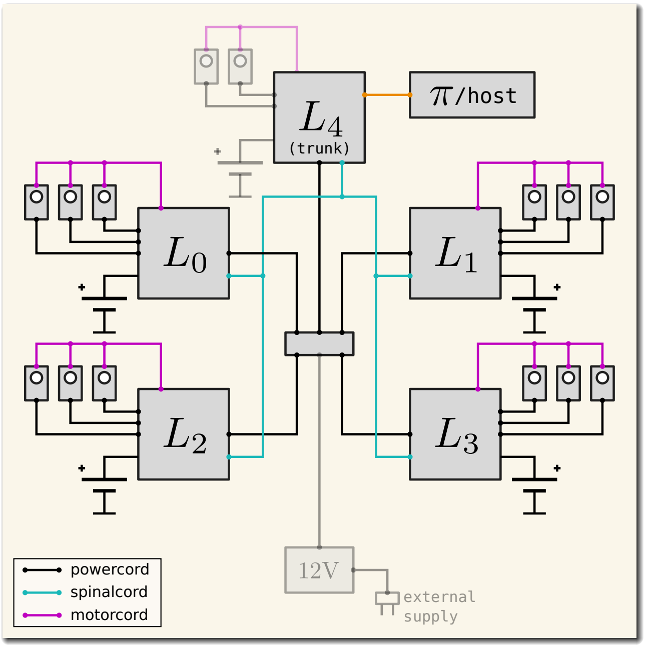


Figure 8.12: Overview diagram of the *Hannah* robot's electronic architecture: Each Limbcontroller powers from a single battery pack. Power distributes from the highest voltage battery, accommodating passive equalization. Sensorimotors receive their power via Limbcontrollers. Power connections use a star topology to minimize losses, while data transmits via separated RS485 Motorcord busses in line topology. An optional external supply voltage can connect to the distribution node. The spinalcord is the basis for hostless Limbcontroller communication. The Raspberry Pi connects to the trunk via separate RS485 to maintain an interference-free real-time Spinalcord communication.

In contrast to a single larger assembly, multiple smaller batteries also improve weight distribution and allow us to shape the torso's inertia tensor. Each cell pack gets fixed to the torso's rods with a 3D-printed pocket back to back with the Limbcontroller boards, ensuring direct connection to their respective battery and allowing convenient battery pack replacement.

The robot's batteries come in packs of three lithium polymer cells in series (3S1P) ranging from around 9–12.6 Volts. In contrast to the *flatcat* robot, *Hannah* currently has no in-system charging, so batteries need to be manually removed from the pockets and recharged with a separate charger device. But since the four-legged robot is a prototype for experimentation on walking algorithms, changing batteries was not an issue yet. For convenience we have the option to connect a 12 Volt external power supply.

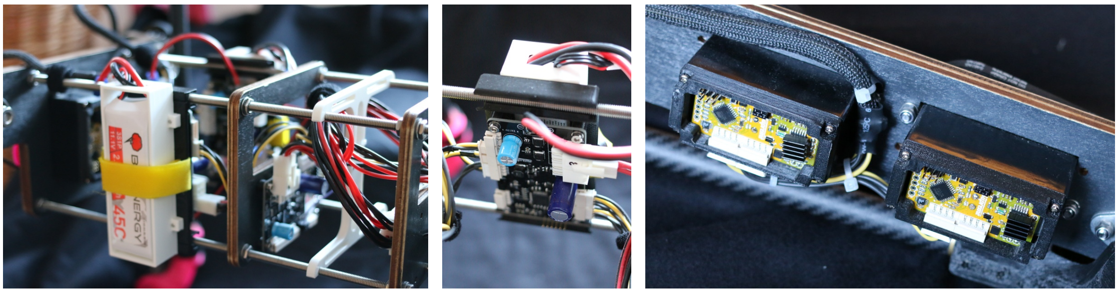


Figure 8.13: Batteries and Limbcontroller units assembled back to back in the torso. (left, center) Sensorimotor units mounted in the upper leg (right).

Control

The Raspberry Pi mounted in the torso features wireless communication, and the application module running onboard the embedded OS regularly reports the sensorimotor data via UDP protocol to the remote client terminal on the user's device. So we can observe sensory data during operation and testing controllers. For now, the client can select the motors control modes position or CSL for each motorized joint individually or set a generalized controller affecting all robot motors in a combined motion. The control loops are calculated locally on the robot for real-time purposes since a wireless connection would introduce unpredictable delays interfering with the concept of closed-loop control.

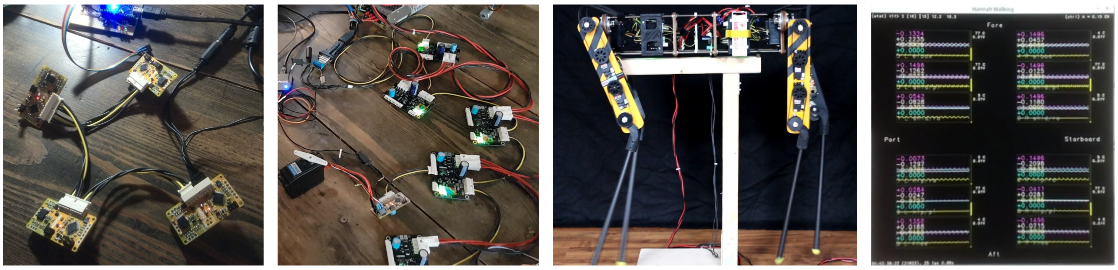


Figure 8.14: From left to right: Wired Motorcord and Spinalcord during firmware development and testing. The robot fixated on a test arrangement during controller development. The client terminal shows live sensory data.

8.3 Bridging the Reality Gap

I was curious whether the simulated locomotion behaviors from the evolution chapter would also work on *Hannah*, as I had been able to remove many specific assumptions, so in principle, there should be nothing standing in the way except the reality gap. Therefore, I designed a simple simulation model of the robot to train and then transfer at least a basic walking behavior. However, creating an exact model is very time-consuming, as I learned from earlier research projects, so I had low expectations for the transfer quality.

The attempt to create an exact simulation of a sufficiently complex robot is bound to fail. Nonlinear dynamic systems behave predictably only in small regions of the real world. However, in their boundary regions, predictability can sharply decrease and eventually fail at bifurcation points, resulting in irreversible changes in state. Accordingly, simulations of robots can only depict a sufficiently simplified form of reality. The effort we put into creating the model reflects the quality of the simulation, so better simulated details typically involve more complex modeling.

Reality Gap
The mismatch between real-world systems and their corresponding simulated models is called the *reality gap*, a deviation that we can only reduce further by enormous cost but that probably can never be eliminated. It is not because the process would be too complex to model or is unknown. It is the effort we must make that is not economical. However, two approaches to minimize this gap are to increase the simulation detail as long as the effort is limited and to improve the system's linearity in its design if possible, thus building the bridge from both sides.

But before continuing, we should ask: What can we expect from a bridged reality gap? If we reduce the distance between the model and the system as far as possible, we can attempt to transfer results from the simulation to the system. However, this generally applies to a particular operating point considered. The results are not necessarily universally valid. However, transferring even a single result might already be worth it.

For walking robots, pre-training in simulation can mean enormous savings in training time on the machine if the path to specific results would otherwise mean many preliminary trials of walking, falling, and getting up again. These days, robot hardware is not yet self-healing and is likely to consume much human maintenance.

This chapter organizes as follows: The first section describes the simulation model of the Hannah robot used for experimentation. I repeat selected experiments for the simulated quadruped, such as walking forward, backward, or turning. I demonstrate again that we need no fundamental changes to switch to another morphology. In the last section of this chapter, I report on the results of the transfer attempt.

8.3.1 A Simulation Model for the Hannah Robot

The first thing I did for the transfer was to model Hannah to use it in the environment. The method I used is invariant to the specific model by using only the sensor inputs $\boldsymbol{x}$ and motor outputs $\boldsymbol{u}$. So, essentially, the entire software stack is reusable. What remains to do is to bring a sufficiently detailed model of the robot into the simulation and manually tune the hyperparameters.

It is not necessarily clear what constitutes a sufficiently detailed model. So I did everything that came to mind and what was feasible for me with household methods. I modeled the robot parts using simple measuring tools such as a kitchen scale and caliper gauge. I weighed the parts and approximated them as boxes, assuming their mass distribution would be approximately correct. Whether a component has the proper shape, color, or surface is secondary. Essential is that the relative spatial position, dimension, and mass (therefore the density, too) fit in their main axes so that the moments of inertia remain consistent for the expected translation and rotation.

Of course, a high degree of detail in physical simulation comes at the expense of computing time. The more detailed the physical model is, the less benefit of the faster-than-realtime calculation is left over. We can see the simulated counterpart of *Hannah* in Figure 8.15. Please note that I made no effort in a pretty visualization[1].

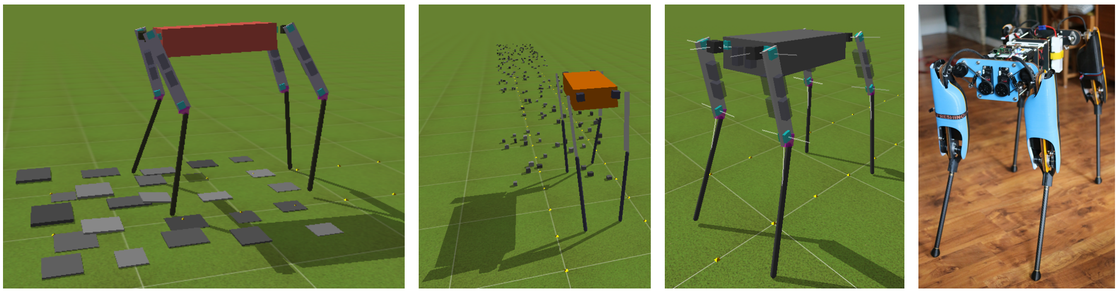


Figure 8.15: The *Hannah* robot's simulated counterpart.

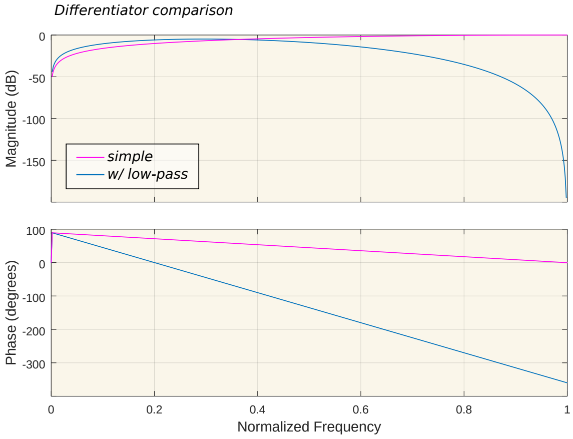
Matching Sensors

An essential aspect is the similarity between the simulated and actual sensor measurements. It turned out that, initially, the simulated sensors were too precise to be comparable to the more noisy inputs of the Sensorimotor. Therefore, for the Hannah simulation model, the position sensors were quantized to only 10 bits resembling the Sensorimotors' ADCs, and they correspondingly adopted their signal-to-noise ratio.

The same counted for the velocity signal as the second highly relevant input variable. Particularly noticeable here was a misjudgment on my part concerning the velocity sensor signals. In the Sensorimotor, I implemented the velocity signal as a time-discrete derivative of the position signal.

But, if such highly quantized and slightly noisy signals are derived discretely, a stark increase in noise accompanied by a little time delay inevitably occurs. To increase the signal quality, I implemented an elaborate low-pass differentiation filter as proposed by Holoborodko (2009, 2008) that met the computational limitations of the 8-bit microcon-

[1] Usually, there is a separation between an approximated model for the physics engine to be efficient, consisting of simple shapes, and a model for visualization purposes that might be visually appealing but gets not checked for collisions or dynamics in the physics engine solver.



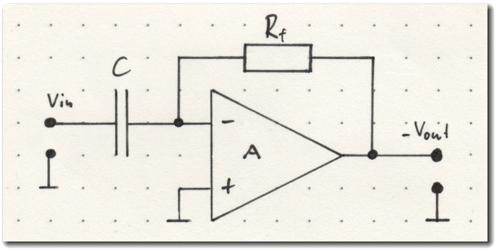


Figure 8.16: Frequency and phase response of an efficient low-pass differentiator using only integer coefficients compared to the simple differentiator (left). An Op-amp differentiator that computes the velocity of an analog signal as $V_{out} = -R_f C \frac{dV_{in}}{dt}$ (top).

troller platform. The filter has the form

$$y_n = \frac{1}{16}\left(x_n + 3x_{n-1} + 2x_{n-2} - 2x_{n-3} - 3x_{n-4} - x_{n-5}\right) \tag{8.1}$$

and efficiently runs on the 8-bit, 16 MHz microcontroller due to its integer-style coefficients and the divisor number using powers of 2. Thus no floating point operations are needed, and instead of using general division, we can use the shift operator that takes fewer cycles. It is an FIR-type filter with the general form

$$y_n = \sum_{i=0}^{N} b_i \cdot x_{n-i}, \tag{8.2}$$

where the x_{n-i} are commonly known as taps of a delay line. Since the coefficients are symmetric, the phase of this filter is linear, and hence the group delay is $\frac{N-1}{2} = 3$ time steps. So the noise could be mastered by low-pass-differentiation, but unfortunately, it comes at the expense of an even more delayed velocity signal, i.e., 30 ms.

To my astonishment, however, it has turned out that the gait controllers are, without any restriction, robust enough to handle the *noisy but reactive* signals, and on the contrary: the smoother but delayed velocity signals caused the gait controllers to overshoot and get highly unstable. Finally, I sticked to the most simple discrete derivative

$$\dot{\varphi}(t) \approx \frac{\Delta\varphi}{\Delta t} = \frac{\varphi(t) - \varphi(t - \Delta t)}{\Delta t} \tag{8.3}$$

and, for now, accepted the noisy velocity signals. However, for future Sensorimotor revisions, I consider an analog computation of the velocity signal based on an operational amplifier that forms a derivative circuit. Without quantization, the velocity signals will improve with lower noise but without significant time delay. Additionally, from the analog implementation, I expect the CSLs will become more sensitive to even the tiniest external forces.

Motor Model Parameters

The motor and friction models from Chapter 2 now need new parameters[2] to match with the Sensorimotors as adequately as possible. If available, few parameters I could obtain from datasheets, and others I measured, if feasible. If neither was possible, I put in estimates as meaningful starting values. These parameter sets formed the initial parameterization, and I tried to find them automatically to fit the data.

For this purpose, I set up a single test joint and a corresponding simulation model, as we see in Figure 8.17. Using the same evolutionary method as for robot walking experiments, I could identify a set of parameters as the basis for the Hannah model. The test joint had to move in a long sequence of different periodic motion commands to record enough data. In the evolution experiment, the motor parameters changed by mutation and crossover to best fit the real motors recorded data.

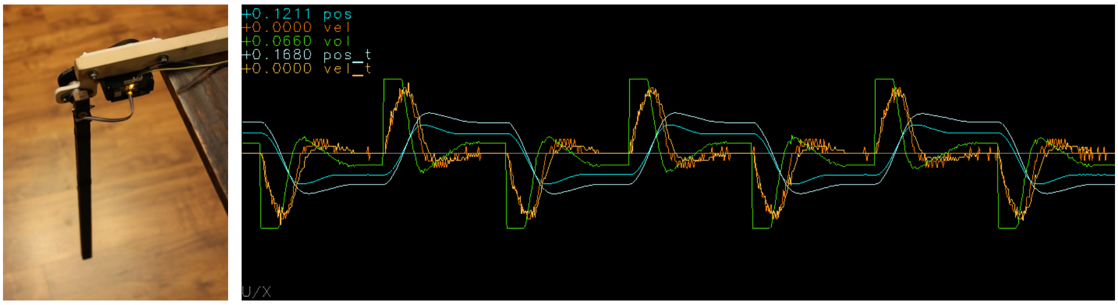


Figure 8.17: The picture shows the test leg used for the automated motor model parameter estimation. The recorded data comprises position, velocity, and motor voltage next to their model-based position and velocity estimates. The motor driving voltage (in green) originates from a PID controller. As we can see, the velocities match quite accurately, whereas the position has a remaining offset.

8.3.2 Domain Randomization

To further close the reality gap, I applied a technique called *domain randomization*, as demonstrated by Tobin et al. (2017). The idea here is to adapt the control to differently parametrized instances of the same system by shuffling its parameters and presenting many randomized versions during the adaptation. So learning a model means generalizing to a distribution of systems accompanied by the assumption that the reality is among them only one instance of many, i.e., close to the mean of the distribution of randomized variations. It is up to us to specify the variance in the system parameters, for instance, of a specific robot segment's size. If we want the gait controller to generalize to the length of the robot's legs, we must present many examples of robots with different leg lengths during training. When the leg length of the real robot is close enough to the center of the distribution, generalization can happen, and the performance may be sufficient.

[2] I conducted the evolution experiments years before I built the first Sensorimotor. So the parameters used in the earlier models were based on a different motor type.

Simloid was adapted for domain randomization to create and reload new robot models at runtime. So the evaluation module can decide when a new robot instance is loaded and, additionally, transfer desired parameters or the standard deviation to which the parameters get randomized. In the generation-based evolution, the model is changed for each new generation, whereas in the generation-free approach of section 3.2, a new robot gets created after passing through any number of individuals which is a multiple of the population size. So we partly keep the comparability of the fitness values when the phenotype changes. We randomized the following properties: The lengths of all legs and the body segments, the masses, the maximum motor torque, the motor and joint friction model parameters, the amount of ground friction, and the default position of the robot's joints. In Figure 8.18, we see some examples of randomized simulation models that significantly differ in size and proportion.

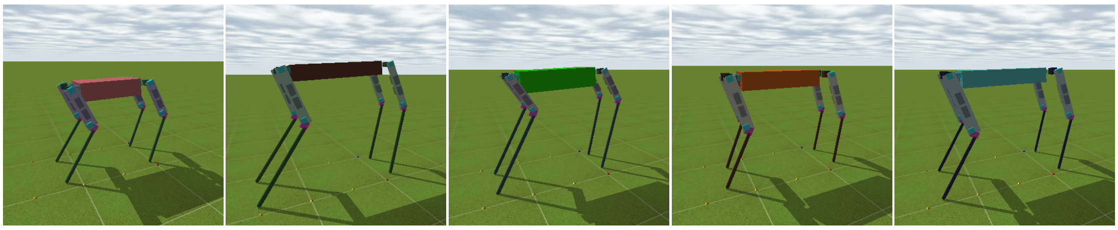


Figure 8.18: Different instances of the randomized robot model.

Results

Results from domain randomized evolution experiments suggest that we can increase the standard deviation of our model parameter's distribution to approximately 15–20% to change the robot's morphology before the performance begins to break down. I find this quite remarkable, considering the gait still works despite a robot having 10% lesser power but 10% more weight. A steady increase in randomization eventually comes at the expense of the locomotion's stability, and thus the fitness function becomes more and more stochastic. Falls are becoming more common, reducing the fitness of those individuals. If the fitness of multiple individuals breaks down, the entire population is soon at risk, and the optimization process loses its function. The single-layered controller's model capacity is insufficient to accommodate that much variation.

The most obvious influence on the walking performance has more torque or the equivalent parameters like a higher supply voltage U_s, higher motor torque constant k_m, or less internal resistance of the coils, R_i (compare the motor model in section section 2.1). Hence, varying battery voltages lead to different walking performances. The choice of the motor determines the other parameters. Less prominent but more interesting are the length ratios of the legs. Generally, the upper legs must be shorter than the lower legs, whereby a ratio of about 3:2 (lower:upper) seems optimal. Lighter lower legs also have advantages. The trunk's weight weakly influences the performance, but unexpected to me, the current weight, combined with all other parameters, is less optimal. A little heavier or lighter would be better, the data suggests. I cannot explain this effect so far.

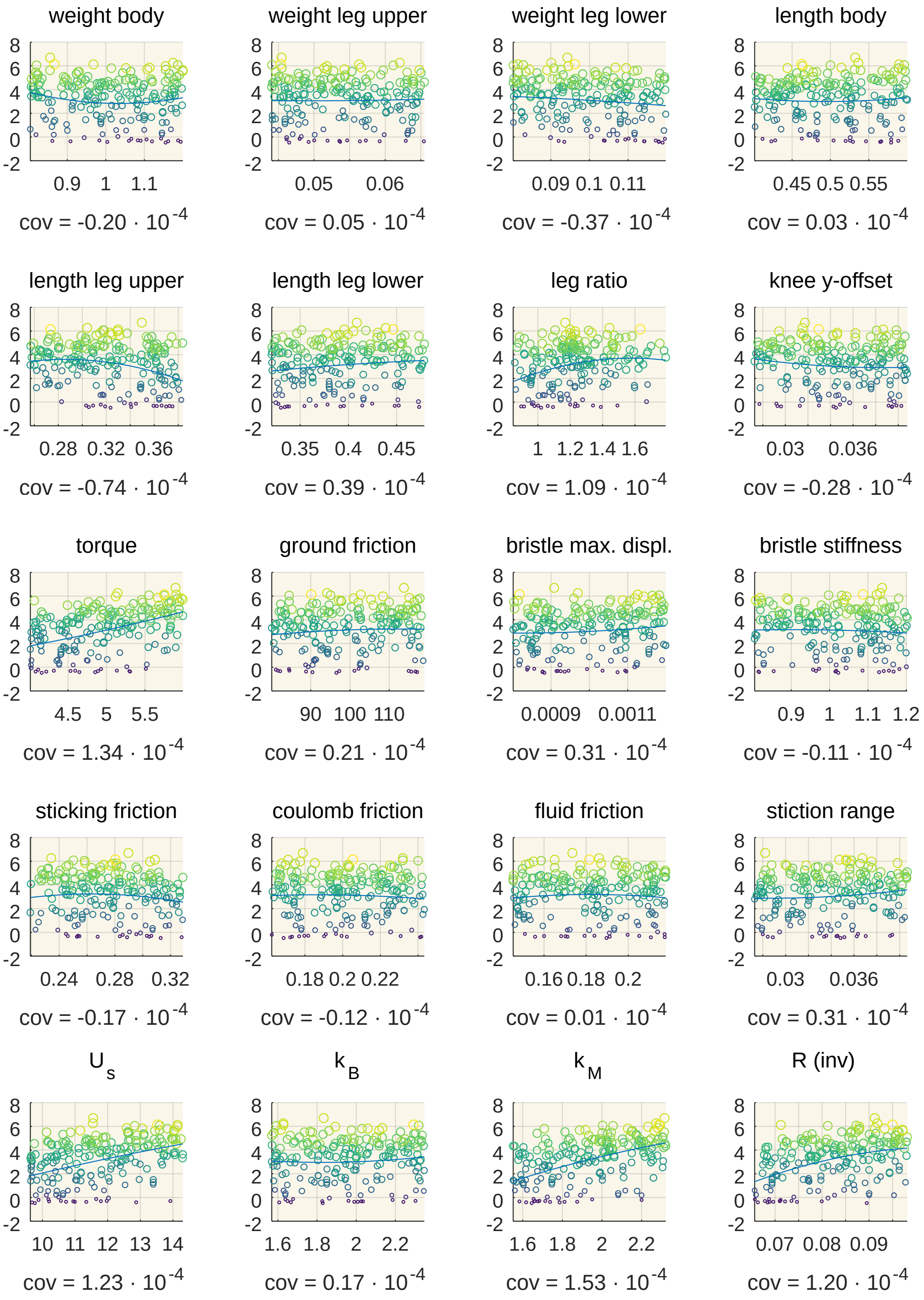


Figure 8.19: Impact of model parameters on the simulated Hannah's walking performance: Shown is the average fitness (walked distance in [m]) of 200 experiments with randomized model parameters. The covariance indicates the influence of the respective parameter. All units are SI.

8.3.3 Testing Simulation Results on Hardware

Testing the walking controllers of a robot of this size is very complex, especially if you do the testing alone. Therefore, to protect the robot, I had to install a retaining rope to catch Hannah just above the ground in case of falling. A pair of pulleys ensure that it can move along that rope.

The first behavior I tested was whether the robot would stand stably with one of the evolved stabilizing stop controllers. I can answer this with yes. The stabilizing controller tries to take the energy out of the system as effectively as possible. Visually, it hardly differs from a PD-controlled stand or a stand using CSL hold mode. All three solutions behave similarly for disturbance-free situations. But when you apply nudges in the forward direction, however, the stabilizing controller makes little steps forward. In contrast, the PD or CSL controller would try to solve the situation with more torque and eventually fall.

The stable standing doesn't handle steps to the side or back. This behavior does not surprise me since the controller was only optimized to slow down the forward walking motion, which again shows the marginal applicability of individual controllers and, in summary, argues for another more comprehensive approach.

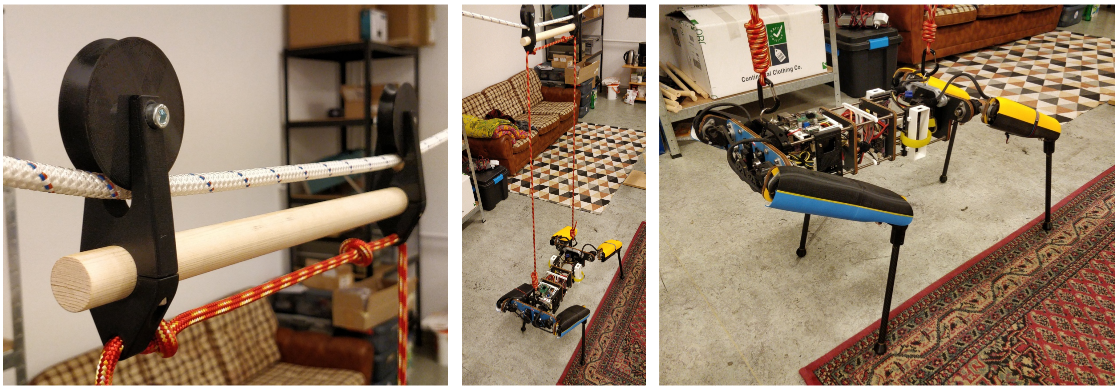


Figure 8.20: The robot *Hannah* during experimention.

The second tested behavior is forward walking evolved with the efficiency fitness function using domain randomization. Initially, I tried it in the test mount without the robot having ground contact. Afterward, I tested with the robot secured at the rope with ground contact. When activating the controller, a running motion initiates immediately, indicating that, already without a closed kinematic chain over the floor, the attractor starts stably, and the multi-joint oscillation runs. The motion patterns look visually correct but run significantly faster than expected.

The fact that the oscillator starts without ground contact is not a desired property. It shows crystal clear that the behaviors are not fully grounded. Desirable would be a behavior that only becomes active when the robot has solid contact with the ground. However, a redesign of the experiment with foot contact force sensors in the simulation and on the robot is still pending. I expect the additional inputs to keep the walking attractor inhibited, i.e., a fixed point when there is no contact with the ground. But what adjustments to the fitness functions are necessary to achieve it remains unclear.

When the robot has its feet on the ground, it takes small steps forward, but neither very straight nor stable for long. After approximately 5 - 10 seconds, the oscillation breaks down because the front feet push outwards while the rear legs push too much inward, and the robot eventually falls. The behavior has quite obviously still a comparatively too small stability range. Slight pushes with the hand can keep the robot on track. Stopping the robot's body manually also stops the walking oscillation, a feature we already know from the Myon running experiments. The type of floor, its friction, and hardness have an enormous influence on the walking motion. With a simple carpet, the robot showed to have sufficient grip. Without adequate friction, the robot slips almost instantly. I have not yet been able to carry out systematic tests with different ground materials. Same here. The robot lacks detailed sensory information about the ground.

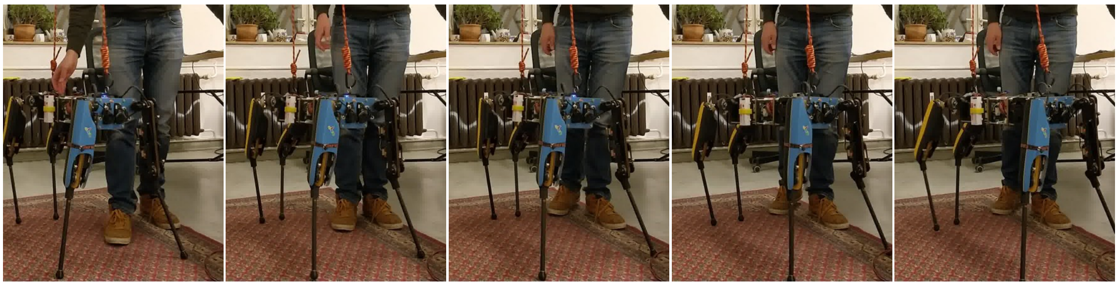


Figure 8.21: *Hannah* while testing evolved gait controllers in autonomous hands-free operation.

Summary

Despite all my critique, I consider this experiment a success. Even if the robot does not run out of the lab right away, the experiment generally demonstrates that the controller architecture can execute meaningful walking movements with a 12-DoF real robot. It also shows that even with a simple simulation model, it is possible to evolve a behavior in the simulation which can already hold the self-made robot in walking motion for a few seconds and advance it a few centimeters. Used as an initialization for pure hardware experiments, this is a good start because we can already measure the distance walked so that it can get gradually improve. It also shows that, for these purposes, we can use hardware with comparatively low costs and simple manufacturing methods.

Additional thoughts: The gearboxes are noisy, and the motors seem to be tendentially undersized, which speaks for brushless direct drives in the future. I suspect a stand-up motion will hardly be achievable in its current form. After all, the transfer in its current raw end-to-end version doesn't seem like such a good idea. I will keep the force-based approach as it has proven superior to position control. But I want to follow a CSL-based approach of abstracting the low-level controller from the behavior, catching all motor-specific characteristics, and having the motors behave like idealized drives. Thus we can continue to use behaviors if motor properties change.

8.4 Gretchen

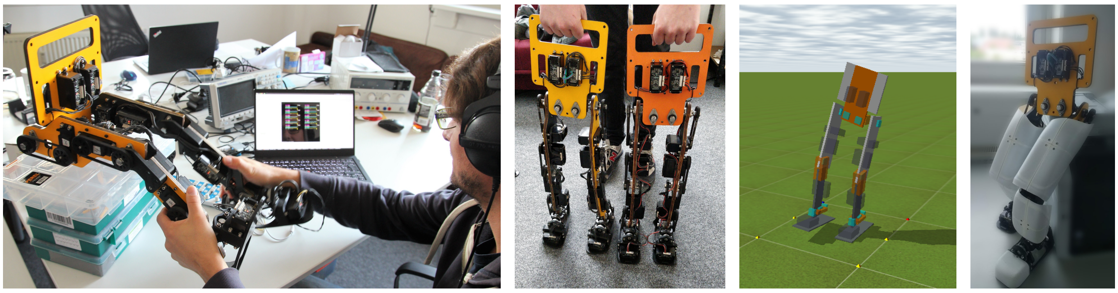


Figure 8.22: *Gretchen* during testing, after finished assembly, in simulation, and with shells.

Gretchen is an open-source humanoid robot development platform designed for education and providing affordable access to bipedal robots. We built the prototype of the 10-DoF humanoid robot from the same concepts, parts, and manufacturing methods introduced in this chapter for the *Hannah* robot. The purpose of this machine was to conduct preliminary experiments to test mechanics and algorithms. In fact, a fully humanoid platform with arms, hands, and a head, similar to the robot Myon, was planned to be built at Aibrain. Unfortunately, this did not happen. So it remained with the first prototype. Nevertheless, it became apparent how we can quickly design new robots with different morphologies from the components and ideas developed for *Hannah*.

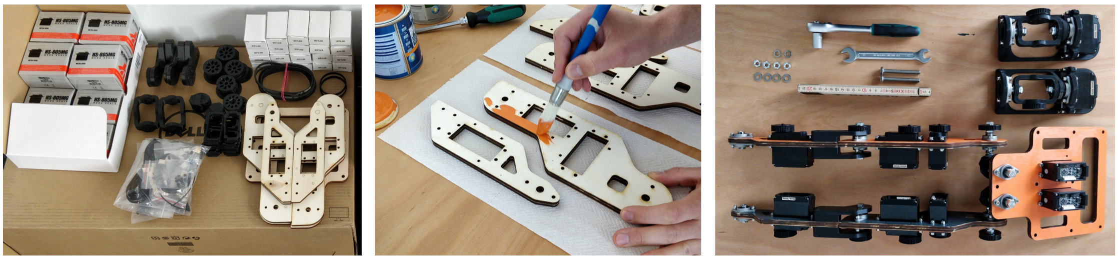


Figure 8.23: The *Gretchen* robot as a kit, during coloring of plywood parts, and during assembly.

I built myself two robots and made three more kits, one of which was assembled here by students of Humboldt-Universität zu Berlin as documented by Prisacaru (2020). Last year, another robot, including all the necessary parts, was also made there, this time from scratch based on the assembly manual and documentation[3]. The Figure 8.23 shows the assembly kit of the robot.

Gretchen's electrical architecture is still kept simple. It consists of a string of 10 Sensorimotors (five joints for each leg: including hip roll/pitch, knee pitch, and ankle roll/pitch). The robot is not yet autonomous: behavior calculation and power supply are

[3] Assembly manual: `https://github.com/ku3i/gretchen`

still external. Any computational unit capable of establishing 1 MBaud serial communication, such as Raspberry Pi or Arduino with appropriate interfaces, can become the central control board for Gretchen. Libraries for Python and C/C++ are available for motor interfacing, and users are encouraged to develop their libraries for other programming languages. Control of the legs is possible through direct voltage/PWM control and position control via the provided library. The default motors' updated rate is 100Hz.

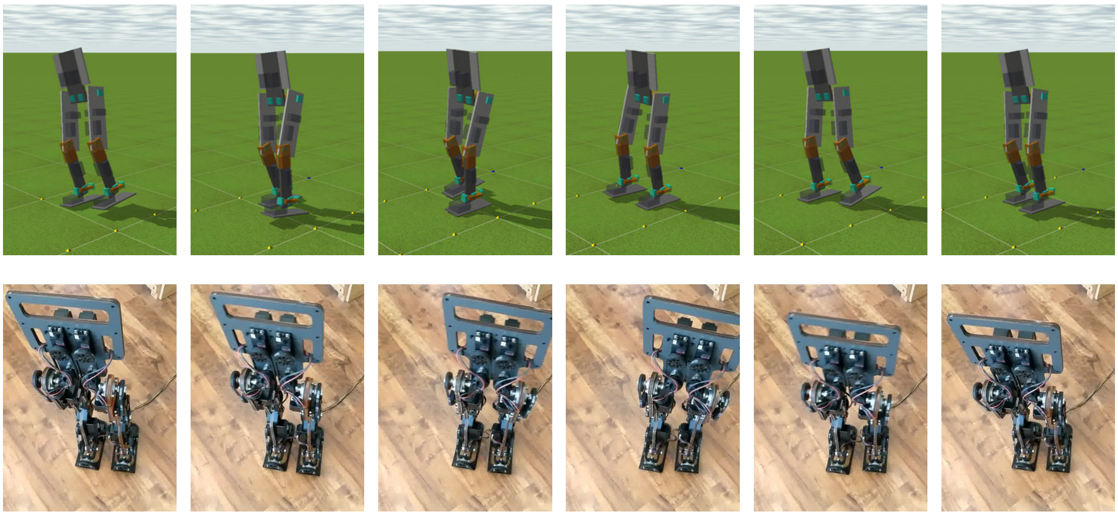


Figure 8.24: The *Gretchen* robot forward walking in simulation and backwards in reality.

I also developed a simple simulation model for the Simloid and was able to generate simple forward walking based on learnings from evolution experiments. Furthermore, I implemented a backward walk with pretty small steps on the hardware for testing purposes, which used the manual approach from subsection 1.4.2.

Despite lacking an upper body and a pair of hip yaw joints, the robot is useful for experimentation and education. Perhaps precisely because it feels incomplete, it encourages students to contribute their ideas. Due to its openness, everybody is welcome picking a niche and start developing: Firmware or simulation, electronics or mechanics? There is rich potential to discover in students when the tasks are diversified.

Conclusion and Discussion

In this chapter, I presented the construction of a quadruped robot using the provided design principles and electronic components. In this way, I demonstrated the use of low-cost materials and accessible manufacturing techniques for modern studies on walking machines. Moreover, I illustrated that anyone willing to can achieve this, too. Additionally, I proved that the investigated controllers effectively operate on real hardware, specifically on robots with multiple limbs, and that, in essence, pre-trained behaviors from simulations can transfer to the physical hardware, albeit with performance trade-offs.

Lastly, I highlighted how the developed techniques apply to the prototype development of other robots.

The transfer from simulation to hardware was only partly successful. Walking broke down after a short despite several countermeasures. I reduced almost every robot-specific assumption and tried to close the reality gap by matching models, using domain randomization, and introducing obstacles or noise. When the gait controller started, the reflex-like patterns were present and performed correctly. Walking was just not stable enough to take more than a few steps in succession. However, if I minimally assisted the robot while walking, it was indeed possible for the robot to walk longer. That maybe means I got pretty close, and learning with the robot system could generally continue, albeit with a different approach. If the proposed foot contact sensors create the desired effect, I do not know. But this is currently my best bet. Along with an idealized motor abstraction or an increase in the controller's model capacity, I am sure that transferring behaviors will eventually succeed.

But my main conclusion is that I first need to find ways to merge the learning results from part 2 into a hardware experiment and gain more experience in online learning on hardware that will endure several hours or even days. After spending months creating hardware and making it robust for experimentation, the four-legged robot is still too brittle for a multi-hour self-exploration learning experiment. So my conclusion was to step back a little from too-complex ambitious systems in terms of hardware and learning algorithms and bring a unified experiment to life to test a long-term learning experiment on simpler hardware but under real-world conditions.

9 A Pet-Like Robot that Responds to Touch

Social robots are a thankful alternative to pets for people who cannot, do not want to, or are not allowed to keep animals on their own. However, I currently see that only a limited number of robots can partially fulfill the functions of a pet. In a recently published paper, Kubisch and Berthold (2022), we introduced *flatcat*, a minimalist, pet-like social robot that does not imitate any existing animal and reacts to human interaction in a novel way.

With *flatcat*, I strive to create an immersive, tactile experience for human users through Cognitive Sensorimotor Loops discussed in section 1.3 and aim to maintain their interest in it through variations in robot behavior driven by intrinsic motivation as discussed in section 4.4. In this last chapter of my thesis, I will fully describe the robot's mechanical, electrical, and software design and present long-term recorded proprioceptive robot data. Many of the techniques, principles, and approaches discussed in this thesis are coming together for the first time in the design of the robot to prove their suitability in everyday use.

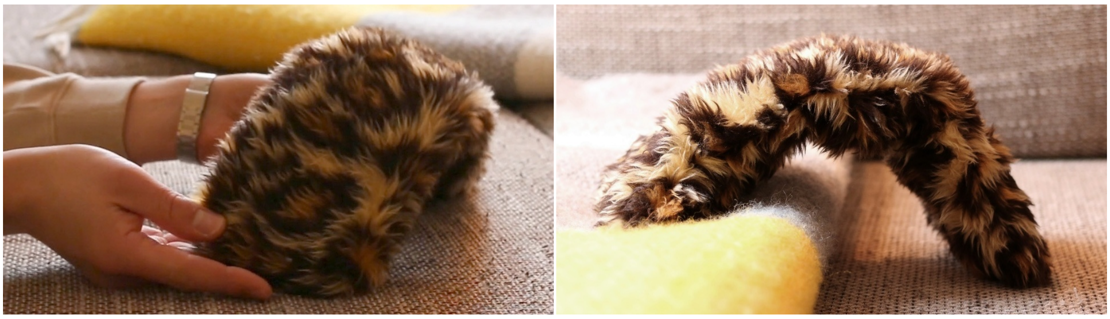


Figure 9.1: A *flatcat* in a typical posture from two different viewpoints (hands for scale).

9.1 Why Pet-Like Robots?

Due to medical progress, the average age of our population is increasing. However, this initially positive result already creates other problems for which we must find appropriate solutions soon. Loneliness of the elderly and related health problems are the subject of constant public debate. Therefore our society searches for new ways to cope with the increasingly difficult situation. Pets today are primarily kept by people for personal pleasure, and Maujean et al. (2015) as well as Kamioka et al. (2014) showed that pets positively affect health. However, pet ownership gets increasingly criticized from a modern animal ethics point of view (Sezgin (2014)) and might no longer be adequate for many people striving for a vegan way of living.

In contrast, elderly or disabled people living in nursing homes may not be physically or mentally able to care for a pet. In other cases, they might not be allowed to keep them for hygienic reasons. More pragmatically, a pet such as a dog or a cat can also be an organizational challenge and is a non-negligible cost factor. The initial investment for the animal and accessories, the ongoing costs such as taxes and food, and the extraordinary but often high costs for doctor's visits and even the eventual funeral add up to several thousand euros. These are obvious reasons why people in the future could consider using suitable robots instead that are expected to create similar benefits for a variety of people who do not want, cannot, or should not have access to a pet.

Animal-Assisted Therapy

The therapeutic benefits of pets are evident across multiple conditions as observed by Bharatharaj et al. (2015): Animal-assisted therapy produces positive psychological effects such as relaxation and motivation, physiological effects such as improving vital functions, and social benefits such as improved communication. Interestingly, they found that there appears to be a preference in some autistic people for rather communicating with animals than with humans.

But animal-assisted therapy has enormous drawbacks, too: On top of the obvious hygiene issues for the residents and the additional work for the staff of care homes, there is also the question of how much stress we should put on an animal. When we use cats and dogs as therapeutical tools, they need regular medical check-ups and washing. Exposing them to many different people means enormous stress for them. If we can provide the same therapeutical use for the patient with non-living animal-like robots, why not use them instead and end animal exploitation in this sector?

Financially, the benefit can be the same or even higher using robots. Nursing home staff can effortlessly take a robot off the shelf when needed, disinfect them afterward and put them back. The expense of purchase and maintenance is comparatively small compared to the cost of a dog handler, and patients could even enjoy their individual pet (robot) way more often than what might be financially possible with therapy dogs, for instance. I made my comparison on the assumption that pet-like robots will soon enter a similar price domain like entertainment consumer devices such as cameras or tablet computers.

Limitations of Pet-Like Robots

So pet-like social robots, sometimes called AI pets, are evaluated to what extent they can fulfill the requirements to replace animals in therapy (Banks et al. (2008); Wada and Shibata (2007); Kanamori et al. (2003)). Unfortunately, such machines are either still expensive, which limits their user group, or their function is not sufficiently fulfilling the needs of people. While developing *flatcat*, I assumed that the central reason for both drawbacks is imperfect mimicry of existing animals: i.e., humans design robots to imitate certain animals, such as a cat or a dog. But here is why this design approach must fail, in my opinion. Rebuilding the appearance of animals is extremely difficult and makes development expensive. On the other hand, it can create discomfort among users if the animal imitation is only partially manufactured and stays below human expectations. Many people who are familiar with the behavior of a cat or dog get easily irritated and reject imitated behavior as not natural (*uncanny valley*). With *flatcat*, I focused on creating a unique robotic *species* rather than mimicking animals. This approach keeps the expectations minimal on how it should behave. Rather it has the opportunity to surprise with how it behaves actually. I observed people are irritated when they see a *flatcat* for the first time but for a different reason. I suspect this is because they never saw such a robot or species before. I believe this is similar to when we are watching a nature documentary and discovering a species of animal that we didn't know before. But as soon as people take a *flatcat* in their hands, their worries are mostly forgotten.

The touch capability is often overlooked by robot designers, despite research showing that touch positively affects health, as demonstrated by Hoffmann and Krämer (2021); Eckstein et al. (2020); Grunwald (2017). Sadly, many robot companies prioritize a sleek and industrial design over a soft and haptic exterior and neglect the power of haptic robot interaction. By touch, we mean physical interaction through force, not just touching sensors, as we commonly know it from phone displays.

But from my perspective, the biggest limitation of social robots today is their lack of long-term adaptability and surprise. Pets, particularly mammals and birds, learn to adapt to humans over their lifetime by living with us. They develop habits and inspire curiosity through sometimes unexpected behavior. We cannot find this level of adaptability yet in pet-like robots.

Examples of Successful Pet-Like Robots

From my perspective, the most notable examples of robot pets are Paro, a 60 cm long seal-like robot, and Aibo, a cat-sized robot dog. Paro is primarily designed for therapeutical use in elderly care homes and for individuals with dementia (see Wada and Shibata (2007)). It is thus not specifically a mobile robot. Aibo, on the other hand, is geared towards a younger audience. It can walk and incorporates many features found in toys. Aibo's design is focused on entertainment. Its creators regularly provide new versions and additional amusing material. Both robots have been on the Japanese market for more than 20 years. Promising newcomer robots that align with my philosophy include Qoobo, a minimalistic robot pillow with a wagging tail, and Moflin, a cuddle robot, both of which got designed to fit in a living room setting. Qoobo is designed for home use. It is intended to soothe with its pillow shape and mimics a heartbeat. It reacts to ambient sounds and wags its tail accordingly. Moflin exploits the child schema and pretends to have emotional capabilities. To my knowledge, all robots react to touch, but none of them respond to forces.

Your Next Robot is a Pet

When designing *flatcat*, I got my inspiration from biological principles without explicitly replicating specific animal characteristics. The two main properties I borrowed from nature are the force-based motion approach and artificial curiosity. The CSLs generate highly reactive movements that often appear more natural to humans than pre-defined motions, which I suspect is mainly because of their instant response to force-based interaction. Intrinsic motivation, on top of the lower-level control, ensures constantly changing but non-random motion patterns, so the robot's behavior is partially regarded as «curious» since the observer recognizes its drive for exploration. I developed *flatcat* at Jetpack Cognition Lab as an affordable consumer product. We manufactured an initial prototype-like small batch of 20 units after completing a crowdfunding campaign with the support of our dedicated community. I chose an unconventional flat shape that avoids mimicking real animals, minimizes the manufacturing complexity, and further lowers the peoples' expectations to maintain an element of surprise, thus sparking their curiosity.

My objective was that humans forget that *flatcat* is a machine. Therefore, I gave it a soft fur and the most minimal user interface by design. It cannot be remotely controlled or voice-commanded. Instead, people interact with it purely through touch and force. As I observed in numerous sessions, the flat and unique form, combined with its autonomy, often led to initial skepticism among people that usually transformed into curiosity once a physical interaction happened. With the help of our friends, we filmed[1] initial interactions between *flatcat* and individuals of various ages. However, due to the restrictions imposed by the Covid-19 pandemic at the time of filming, we could not obtain footage of interactions between elderly individuals or dementia patients in care homes. But through the many more (unrecorded) interactions we have had since then, we have strong anecdotical evidence that people of all ages showed great interest in that robot, often accompanied by expressions of surprise and happiness.

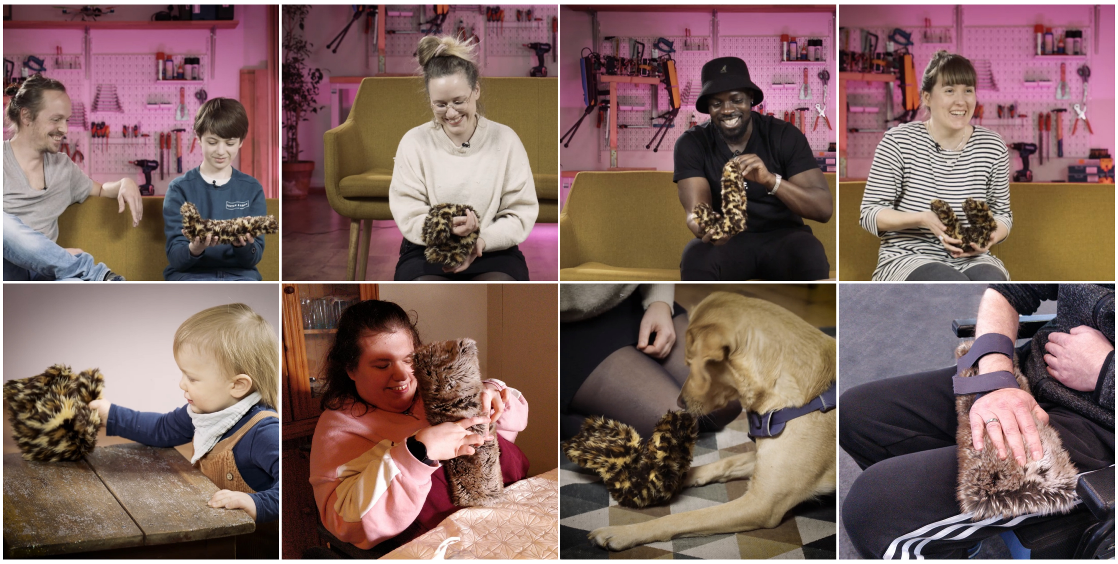


Figure 9.2: The *flatcat* robot interacting with people.

[1] video 1) *flatcat* introduction: `https://youtu.be/bWqo2P_viWA`
video 2) playing around with *flatcat*: `https://youtu.be/jIPktgG2l2E`
video 3) peoples reactions (german, engl. subs): `https://youtu.be/m6Fx-jfwh60`

9.2 Unique flatcat Features

flatcat has a greatly simplified structure. Three motorized joints connect four segments, including a head and a tail. All joint axes point in the same direction. It takes the form of an oversized flattened caterpillar, hence the name: The word *cat* is short for caterpillar. I picked this name intentionally to be a bit provocative because people first think of an actual cat, favored by the fluff. Functionless button eyes integrate into the animal-colored fake fur as requested by some customers because people tend to be unsure where the front and back are, and the eyes helped them orientate themselves. Figure 9.3 shows the robot in different poses without the fur.

As a daily companion that can withstand everyday household conditions, *flatcat* must be robust and easy to maintain. *flatcat* is lightweight and has an intuitive grip, making it easy for children to hold. Additionally, its flat shape allows for easy transportation. Together this makes the *flatcat* a practical and user-friendly companion robot. The intended robot's environment is the sofa or a person's arms, hands, or lap. It is hence prone to regular contamination from food or drinks. So the fur is washable and can easily be put on and off without technical expertise, addressing hygienic issues simultaneously. The fur's design comes as a sock without zippers or Velcro closures.

The Sensorimotors developed for *Hannah* were too big for *flatcat*. So the board layout had to be redesigned (Revision 1.3) to accommodate smaller servos. The DSServo DS3225 with a stall torque of 25 kg/cm at 6 V was a suitable option for *flatcat*. It integrates a coreless DC motor and metal gears that are already easily moved by the passive weight of the robot's limbs. Coreless DC motors have a hollow, self-supporting rotor that brings a significantly reduced mass and inertia, making them way more back-drivable than common DC Servos, which makes them an ideal choice for a CSL-based control.

The electronic setup is straightforward and consists of three motors, a Headmodule, and an Energymodule as fully described in Chapter 7. In Figure 9.4, we can see the specific flatcat architecture. From software to hardware, *flatcat* is fully made open-source. The interested reader will find the *flatcat* application software, the schematics, firmware, bootloader, and middleware of the Sensorimotor revision 1.3 in the Jetpack Cognition Lab's git repository[2]. Despite its simple structure, the *flatcat* robot possesses unique properties that we will discuss for the rest of this section.

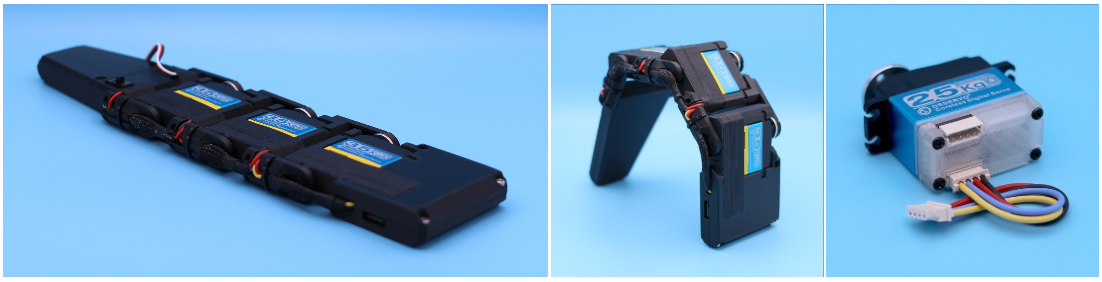


Figure 9.3: The *flatcat* robot *undressed* and a single Sensorimotor servo drive.

[2] `https://gitlab.com/jetpack-cognition-lab`

9.2.1 Purring-Like Sounds

DC motors are structurally similar to speakers, and I experimented with playing sounds during motor movements. I became interested in creating pet-like sounds, similar to a cat's purr. The pulse width modulation (PWM) that the Sensorimotors use to control their attached DC motors with varying voltages commonly have inaudible high frequencies above 20 kHz to eliminate unpleasant *motor singing.* However, to create a purring sound similar to that of a cat without additional speakers or dedicated vibration motors, the PWM frequency should be as low as 25 Hz. However, continuously controlling motors at such a low frequency will make movements jerky, stresses the gears, and feel unnatural.

Therefore, the firmware had to be able to change the motor's PWM frequency dynamically. In addition to the inaudible default setting of 22 kHz, I can now adjust the PWM to frequencies as low as 20 Hz, which creates a purring noise that users can simultaneously feel, giving the *flatcat* a unique sound design.

To avoid a dull and unrealistic purring, I needed to find a pattern that was not overly predictable but also not too random. I assigned a single tone from a low blues scale («b-flat») to movements of each of the three joints in both directions. The six tones are $B\flat_0$ (base tone), $D\flat_1$ (minor third), $E\flat_1$ (perfect fourth), F_1 (perfect fifth), $A\flat_1$ (minor seventh), $B\flat_1$ (octave). I skipped the E_1 (the diminished fifth or blue note) from that scale because it was too close and barely distinguishable. The tones range roughly from 29–58 Hz and only play when the motor voltage is between 1 and 2%, where the motors do not even start moving.

Employing a musical scale enhances the sound for human comfort. Although the fundamental frequencies of the scale may not be audible, the overtones are. This approach ensures that the purring is varied yet harmonious to some extent. In video number 2, you can listen to the sound of a *flatcat*.

As a result of the sound design efforts, we can now also use the aforementioned nonlinear extension on PID position control of subsection 1.2.2 to overcome sticking friction by switching the motor to a low-frequency vibrating mode using dynamic PWM when the target position is close. Jerking prevents sticking before reaching the target or getting stuck in a cycle of constant overshooting. But even though *flatcat* does not employ position control, dynamic PWM is relevant here. Playing the purring just before the movement starts reduces sticking friction to some extent and smoothes the onset of the motion.

9.2.2 Artificial Endurance

Most electric machines require a regulated supply voltage. However, if the robot resembles a pet, voltage regulation may not be necessary. In nature, animal behaviors rely on changing energy levels, and animals show alternating periods of activity and rest. To create more realistic and animal-like robots, we can either mimic this behavior through software or replicate the underlying principles that produce it through simple body dynamics as suggested by Braitenberg (1984). Therefore, I propose that we regard the

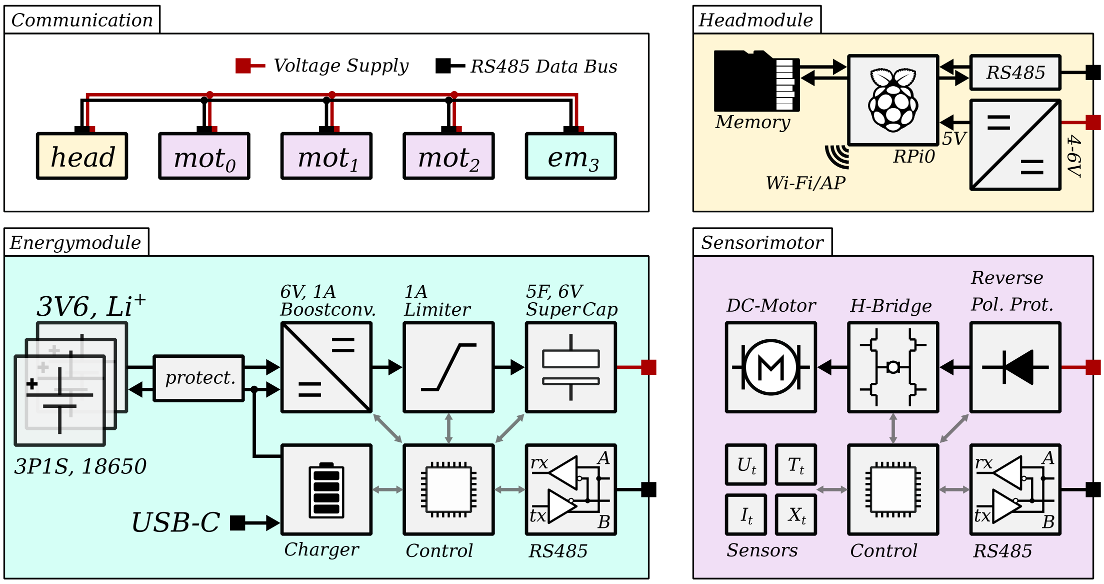


Figure 9.4: The electronic setup of *flatcat* comprises a Headmodule, three Sensorimotors, and an Energymodule. All connect to the power supply with varying voltage levels and a daisy-chain-type RS485 data bus. The Headmodule integrates a low-cost embedded Linux System-on-Chip with wireless communication. The Sensorimotor boards of rev. 1.3 provide sensor feedback and control modes like CSL. The Energymodule manages battery charging and includes a supercapacitor-based voltage buffer.

robot's supply voltage as similar to an animal's vitality, and we should further investigate this aspect of robot design. As a byproduct, humans can observe the power reserves of the robot easily through its movements. It allows for an intuitive understanding of the robot's vitality through instinct and empathy, incidentally eliminating the need for a separate battery life indicator.

The Energymodule, as presented in section 7.3, is a custom-designed electronic board that manages battery power and regulates the bus voltage through a microcontroller. The *flatcat* robot is equipped with three parallel battery cells, providing a capacity of approximately 10 Ah to achieve a runtime of around 8 hours. Due to the limited voltage range of a single lithium-ion cell, a boost converter increases the voltage and charges a supercapacitor. This supercapacitor can supply the motors with peak power for several seconds. It recharges at a limited rate for several seconds until it restores the maximum voltage. Since a lower voltage is observable through slower and less energetic movements, such short-term recharging cycles result in natural-looking periods changing between activity and rest. It also protects the user by limiting the amount of mechanical power the robot can use.

In Figure 9.5, we see an example of the robot's activity showing battery and bus voltage, power consumption, and temperature development. The right column highlights a particular 2 minutes section of the time series where we can watch the activity cycles.

The bus voltage constantly recharges during robot rest. When the supercapacitor gets charged, the battery voltage usually drops under load. When the bus voltage drops below 5 V, the robot inhibits motion during the recharge, so the motors can also cool down to some degree. After a while of activity, the motors' temperature has exceeded a preset limit, forcing the robot to stop its activity until a lower temperature threshold is reached again (hysteresis). Since the cooling of the motors takes time, these pauses are significantly longer, depending on the ambient temperature. So we can observe several self-regulation mechanisms active in *flatcat* that correspond with cycles of activity and rest also found in animals. Thus, the variable bus voltage is useful as a simple way of simulating the physical endurance of the robot, making it a bit more lifelike.

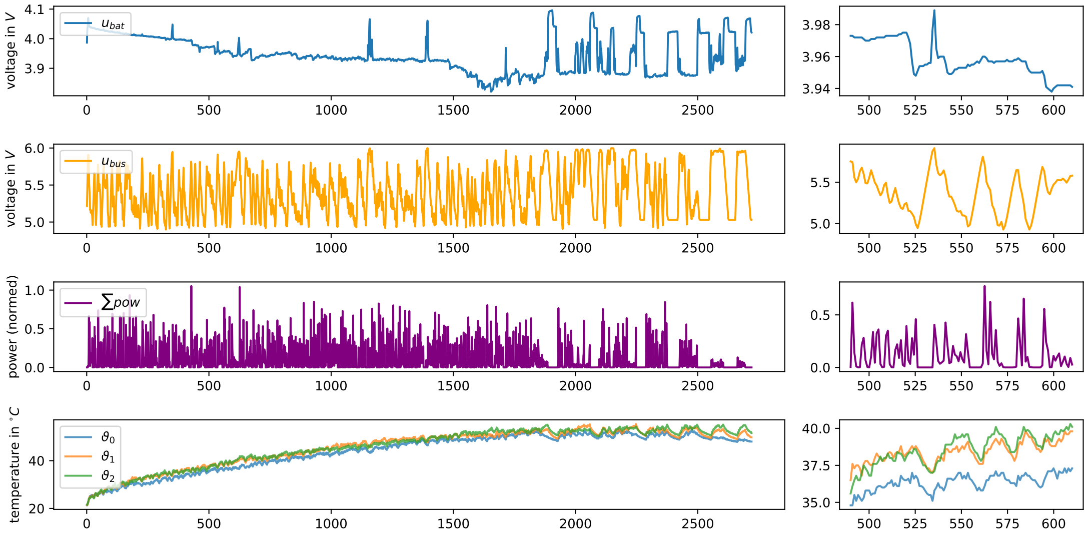


Figure 9.5: *flatcat* activity: The first row displays the battery voltage showing two dynamics, the slow discharge over time and the rapid drops under load when the supercapacitor gets charged. The second row depicts the buffered bus voltage ranging between 5 and 6V, showing the activity cycles of the robot. The movements are more energetic with higher voltages. Below 4.5 V, the sensorimotors stop working, so the robot inhibits its activities when the voltage drops below 5V, which can be best seen in the 3rd row showing the sum of electrical power consumed by the sensorimotors. The last row shows the temperature development of the motors with increasing temperature under load.

9.2.3 Intrinsically Motivated Self-Exploration

From my perspective, the ideal pet-like robot acts independently, develops individual preferences, and expresses its personality to arouse the user's interest. I am confident we can achieve this by balancing its behavior between the two extremes of having strong physical predictability on the one hand and by showing explorative variations that are not purely random but follow a certain systematic. I assume such an algorithm will create a unique and constantly evolving experience.

Again, *flatcat* does not aim to mimic any existing animal but to convince with behavior that we normally perceive as animal-like. So that humans get convinced and forget that *flatcat* is a machine but without being too confident about what specific species it is. Allowing it to make its own decisions will probably result in behavior matching its physics when we use force-based controls. From my experience, humans usually perceive such motions as «realistic» or «natural», promoting our animal illusion.

We extensively discussed lower-level motor controls in section 1.3. The challenge is now to find out how to blend between the CSL's behavioral modes of contraction and release, i.e., to determine when each joint should follow or counter the force. To address this, I implemented a self-exploring learning system for the robot to let it find out by itself. It should exhibit curiosity towards movements not already explored and continually improve its ability to predict the outcomes of its actions. To accomplish this, I employ growing neural networks and an unsupervised approach to create a model of the state space and identify opportunities for learning, setting appropriate intrinsic rewards for reinforcement learning.

My goal for *flatcat* is simple—to maintain the user's interest. I hypothesize that for this to happen, the robot must exhibit a certain level of interest. By continuously generating diverse behavior through artificial curiosity, it can uncover new states of itself in its environment. I believe that users want robots that are neither dull nor unpredictable and that robots offering occasional changes in behavior can keep them engaged.

Hence, the only reward driving the robot's future behavior is its learning progress, $L = -\dot{E}$, which we defined as intrinsic motivation in section 4.4. The only requirement for the robot is to establish a self-improving world model as introduced in Chapter 5 and to evaluate its improvements by comparing its predictions with actual sensor data. That does not necessarily mean that other reward sources would not be useful for pet robots, e.g., rewards for self-protection, but for *flatcat* intrinsic rewards are currently the only ones implemented.

My *flatcat* learning model has the following components: A growing multi-expert structure for model building, determining the robot's state by employing the winner-takes-all principle. The expert nodes are Time-Embedded Neural Networks that generate predictions of upcoming sensory values and adjust them through stochastic gradient descent. This dynamically growing set of states connects to a $\textsc{Sarsa}(\lambda)$ reinforcement learning system only fed with L as rewards. Actions come from a predefined collection of CSL behavioral modes allowing only contraction and release for each of the three robot's joints. The model is hence very similar to the one of the learning Pendulum or the Tadpole robot learning to locomote, as I illustrated in Figure 5.8 and Figure 5.10, with the difference this time that I deployed it on multiple real robots in a complex environment.

Grounding

Since every movement that *flatcat* performs is also massively influenced by external forces, and since it can only decide whether to move with or against a certain force direction, I consider the robot grounded. We should think about the term *grounding* in a way that the robot doesn't try to correct for any disturbance to follow a trajectory we plan. Rather it embraces the forces from its environment that are inevitably present and accepts them as they are, then moves accordingly in the force field and seeks its way through it. So it is always in touch with the force. However, the limitations of this approach we already discussed should not diminish the potential of a completely new control paradigm. Because even if it is not yet entirely clear how highly dynamic movements or regular gait patterns emerge from CSLs, the function provided already has, so far, been completely satisfactory for the *flatcat* robot and is often a reason for users to be happily surprised. Please refer to Figure 9.6 which exemplarily illustrates the behavior of *flatcat* when all motors are in contraction mode.

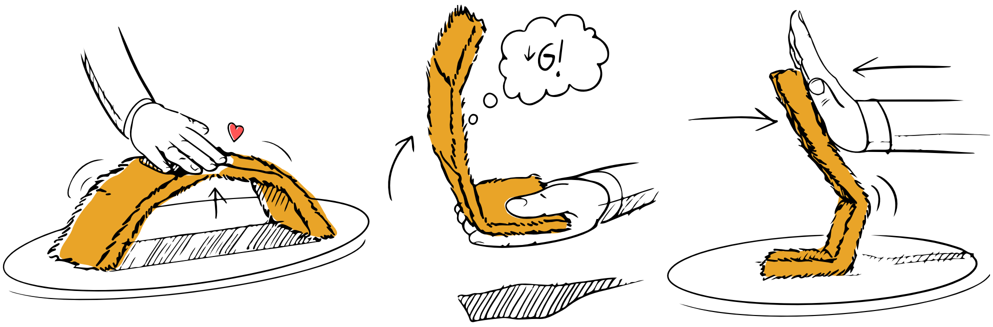


Figure 9.6: *flatcat* in contraction mode defying gravity or other applied forces.

9.3 Long-Term Field Experiments

In the last section of this chapter, I want to look at the usage data of different *flatcat* robots. The robot writes log files for each session writing data points every second. The recorded data does not allow any conclusions to be drawn about the user's identity since a robot only records proprioceptive sensor data, joint positions, currents, voltages, temperature, etc., with no audio and no video. I show the usage times by intention session-wise. It is hence impossible to conclude the exact time or day of the usage. Since a *flatcat* does not have a real-time clock, the system time stops when switched off. If the users do not use the option to set up wifi (e.g., when checking for updates which many users forgo to do), the system cannot retrieve the actual time.

My first question about the recorded data is, can we derive how the users' interest develops over time? Hence, is the behavior of the robot engaging and motivating? Does it arouse curiosity? In Figure 9.7, we evaluate the usage time per session and look at how the duration of use develops over the number of sessions. It is noticeable that there is an irregular distribution of the durations. The majority of times are less than an hour. The median value ranges from 28 to 54 minutes for different users. In addition, there are some extremely long sessions of more than four hours. The robots *Susi* and *Bingo* have similar use patterns, and their users seem to sometimes prefer very long sessions. Robots *Betty* and *Olli* have more frequent but shorter sessions. Overall, the data situation here is difficult to assess how the users' interest develops over time. The session length does not support a statement about increasing or decreasing user interest. Rather it seems to remain roughly constant.

The next question is, how do the robots develop and (statistically) behave differently depending on their environment and users? In Figure 9.8, we can see the proprioceptive robot data of the four robots. Each subplot shows the last 10^5 seconds of behavior, depicting the head, trunk, and tail motors' position vs. the driven motor voltage. Each dot in the diagram is weighted by the absolute value of the motor current and colored with the actual motor's temperature. Areas of denser color represent positions in action space that are visited more often than lighter-colored areas, with a white background representing no visits.

The action space is not developed equally among the robots. The patterns of *Betty* and *Olli* look similar, whereas *Susi* and *Bingo* show distinguishable preferences. The sharp vertical double line in the middle of each diagram corresponds to the current peaks during purring sound emission created by the motors at a 1–2% voltage level. The temperature development varies over the three segments: tail motors tend to heat the most because they are closer to the end of the fur (more isolation) and due to the Energymodule emitting heat.

The data indicates the robots prefer curled position instead of lying flat on the ground. The longer usage times of *Betty* and *Olli* show a more regular coverage of the state space. Since all diagrams show the same number of samples, there is an indication that there might be more variation in behavior.

In Figure 9.9 we can see histograms of the joint angle space $\varphi_0, \dots, \varphi_2$. We see that *Olli* discovered the most area, even partially to the mechanical edges. Please note that the CSLs in contraction uses safe bounds as described in section 1.3, therefore, this exploration of the joint stops is based on user actions (probably while the robot relaxes) and not on the robot's self-exploration. Interestingly, the action and joint spaces of robot *Bingo* peak out significantly. Presumably, this user liked it more to watch the robot during self-exploration and was less interacting with it.

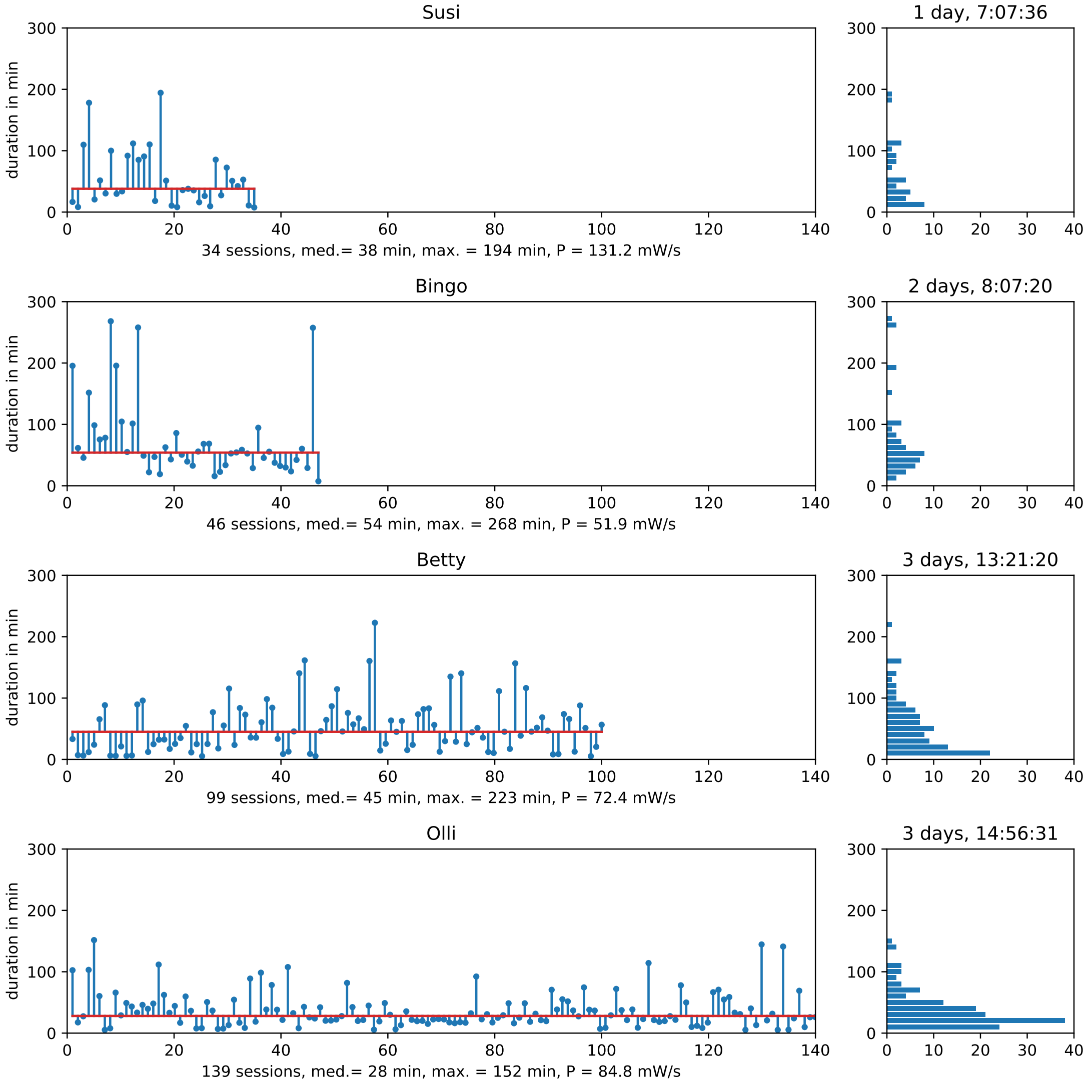


Figure 9.7: *flatcat* usage data statistics: Duration in minutes per session for four different robots. The red line denotes the median. The right column shows the respective histograms of the data and the total runtimes. The usage duration varies greatly for the different robots. Some users prefer a few but very long sessions, while others like frequent shorter sessions. Overall, the data situation regarding the development of user interest is still ambiguous here. At least, we can observe no decrease in the session length, lightly indicating preserved user interest.

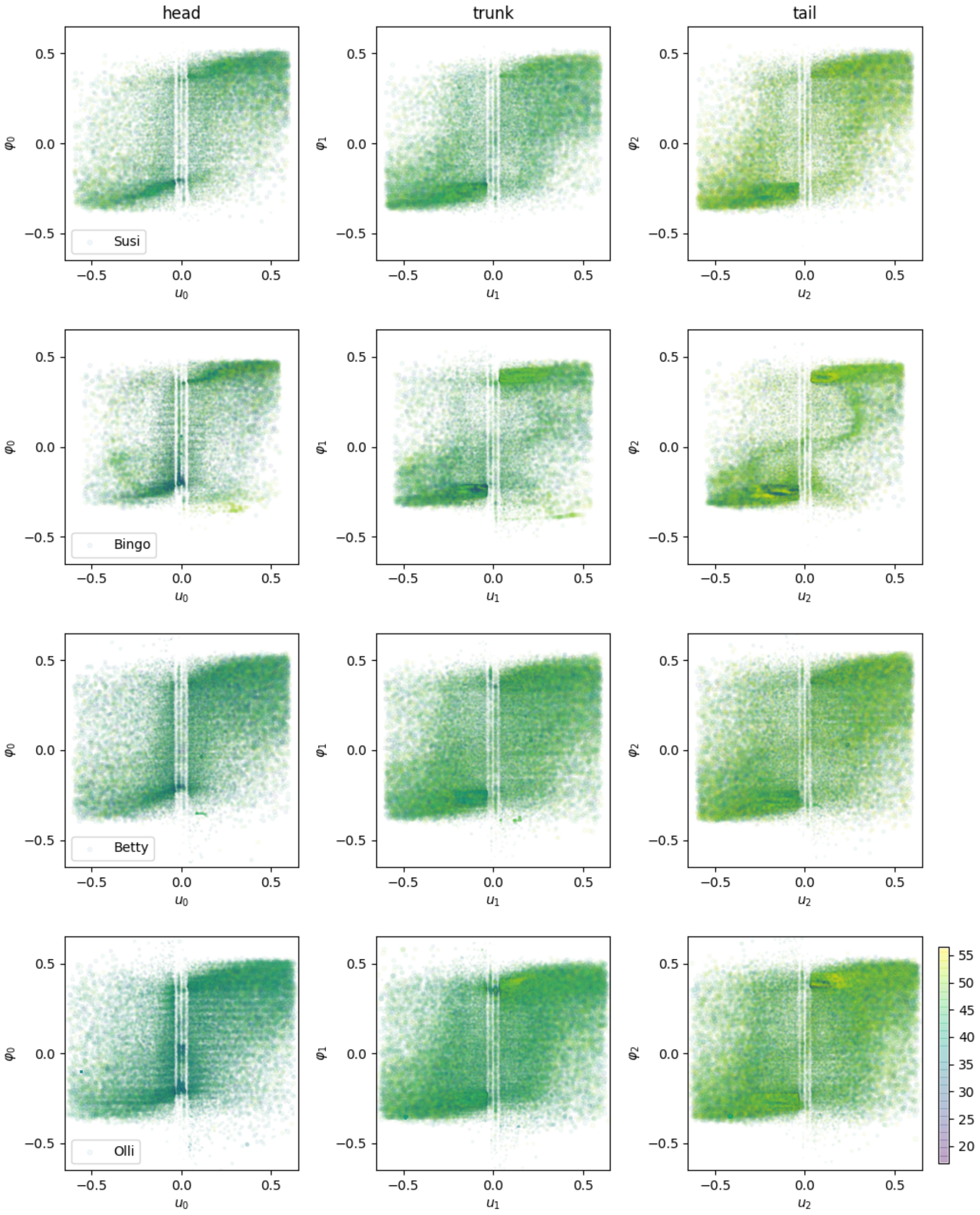


Figure 9.8: *flatcat* differences in behavior between environments (action space): Areas of denser color represent tuples (u_i, φ_i) with higher motor current, indicating higher motor torques, whereas blanker areas denote less torque. The color denotes the sensorimotors' temperature in $°C$. The data indicates that robots in differing environments use other actions, presumably induced by their users reacting differently to the force-based sensorimotor loops. For the second robot with the lowest average power consumption, I assume the user liked it more to observe the robot while playing intrinsically motivated instead of constantly opposing its movements.

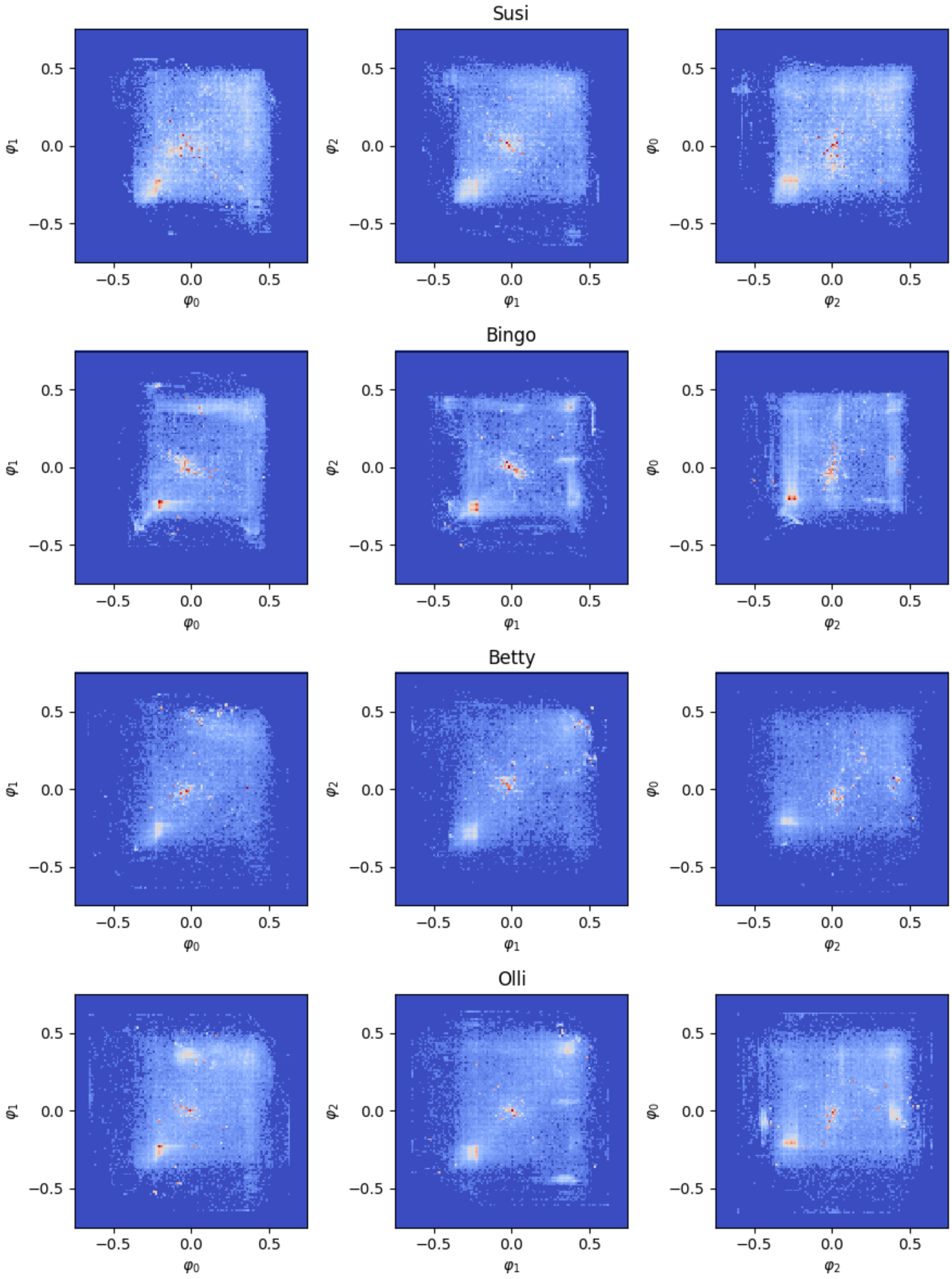


Figure 9.9: *flatcat* differences in state space coverage: Blue areas were never visited, white areas had a few visits, and red areas denote places of frequent visits or longer stays. Every row represents the robot's joint angle space ($\varphi_0, \ldots, \varphi_2$) for an individual *flatcat*. In addition to the user-specific preferences in the action space, there are differences in the degree of self-exploration in the state space, too.

Conclusions

I introduced a novel and affordable pet-like robot featuring unique combinations not previously seen in a single robot or end customer product. It uses newly developed force-based control loops run on open-source motor control boards, allowing the robot to react sensitively to forces such as gravity or user interaction. Allowing for variable supply voltage creates a foundation for natural cycles of activity and relaxation. The robot's growing neural network learns its states through unsupervised methods, guided by a policy that maximizes learning progress and creates curious behavior driven by intrinsic motivation. To a certain extent, all threads come together in building this robot, which means that most of the topics discussed in this work ultimately found their application in real life.

I gave the *flatcat* robot an unconventional design, not based on any specific animal, but instead using force-based movements for a more natural and convincing behavior. So far, the robot has received positive reactions from many people, who are impressed with its liveliness and autonomy. It could be due to the technical methods or its unique form that does not set expectations based on known animals. However, it strengthened my belief that the *flatcat* robot has the potential for use in therapy, rehabilitation, and care.

Figure 9.10: Dr. Oswald Berthold and I proudly shipping our first batch of *flatcat* robots.

Summary of Part Three

This last part was about designing electronics and hardware in the context of an open-source culture, which I then directly used in creating robot systems specified for different applications. The behavior controllers and learning procedures I studied in parts one and two got tested on hardware and even partially made it into productive use on robots now owned by end users while still continuing their learning.

I developed custom electronics, granting me complete freedom in designing the controller and enabling me to explore modern and experimental forms of robot control. Testing learning procedures on real machines proved significantly more challenging compared to simulation. Notably, my time and patience stand out as a potential bottleneck. Creating a robust and reproducible experimental setup poses substantial challenges since the possibility of unexpected breakdowns can lead to maintenance-related delays. Some of these findings were quite demanding to digest.

At this point, I got partly disillusioned to admit that the autonomous robot experiment, the self-built quadruped on the meadow, stays reserved for the future and would mean some more of my effort. Nevertheless, I finally created a setup with reasonable restrictions where robots with an open-ended learning algorithm continue running and adapting to their current sensor data. Even if the changes in their behavior are visually less spectacular than I had hoped, it is still a fantastic achievement for me that 20 playful, force-sensitive robots, being rewarded for maximizing their individual learning progress, are in regular use in so many places. This goal thus sums up most of the strands opened in this work in a single point and makes me want to climb onto the next level of expanding this.

Outlook

What will be the next steps? The work on the quadruped will probably get less priority. I realized that it would likely need a bigger team. Neither are my algorithms mature enough for rushing to the next level of complexity, nor is the quadruped's hardware sufficiently robust to give free reign to an unsupervised learning process. It is not a good fit yet and still needs some work. First, the four-legged robot will have to reassert itself in the simulation and show progress in learning different movement skills simultaneously, just like the tadpole robot from subsection 5.2.3 did.

At the same time, I will develop *flatcat* further, increase her robustness, and, if necessary, make her a little smaller, cuter, and wilder. To do this, I plan to conduct mock-up studies with various people, where I vary in lengths, shapes, and furs to record their opinions. I further need to find out when it is the right opportunity for *flatcat* to be wilder and to perform highly dynamic movements in her current habitat.

The CSL controller, intrinsic motivation, and the variable bus voltage fully determine *flatcat*'s behavior. Currently, the user still has only indirect influence. I want to change this in the future. Although the CSLs are already sufficient to align with gravity and make interesting movements just based on angular velocity at the joints alone, I hope that an added inertial measuring unit (IMU) will help behavior to emerge that plays even more intensively with the robot's body dynamics and with forces induced by the user. In the Sensorimotors, the electronics for capacitive touch sensors on the motor metal surfaces are already present and can send signals. However, this feature is not in productive use so far because hardly any signal passes through the fur. After eliminating this flaw, it would allow coordinated strokes with the hand along the body and might be beneficial as the first external reward signal given to the system. The user could now provide explicit feedback to the robot, effectively rewarding the robot for individual user-desired behavior.

Incorporating acoustic subverbal sentiment analysis as a derived sensor value would introduce a fresh level of quality, enhancing the *flatcat*'s ability to receive additional feedback regarding the user's current mood.

For almost every sensor, I currently have set homeostatic rules explicitly programmed to ensure the user's and robot's safety. The robot did not learn these rules by itself. They include bus voltage control, CSL stall detection, and motor temperature shutdown. Adding more of them for each new sensor would further impact the robot's behavior. Excessive noise, excessive acceleration, or excessive stroking would similarly trigger protective reactions.

I also think about an abstract voice module that generates simple subverbal sounds via the motors by evolving the nature of the sounds from internal state activations. For example, the continuous vector of GMES(τ), related to the prediction error, would feed this module so the sound sequences verbalize internal state sequences. High prediction errors could amplify certain utterances, resulting in immediate audible feedback for the users.

On the algorithmic side, I would like to continue the search for higher-order goals. With intrinsic motivation as a starting point, I am looking for further signals in the learning system that we would interpret as learning progress. The creation of new experts, for example, could become a reward. Alternatively, the system measures at which expert the prediction error is maximum and sets the internal goal to visit this state. If it reaches this goal, the system gets a reward. Especially exciting, I find the idea of integrating the empowerment principle of Polani et al. (2001) to make it possible for *flatcat* to approach states in which its overall options get maximal.

To close this outlook, I want to express another idea I am already working on. Using the title *flatcat entangled*, two of my pet-like robots connect peer-to-peer via narrowband IoT. Together, they form a combined system featuring a common control but possessing two bodies in different locations. For the learning methods described in this work, the number of sensor inputs and motor outputs play only a minor role: doubling them is not a problem. But the behavior of the robots is then entangled. The force exerted on one body yields an internal state change and manifests into a reaction on both robots and vice versa. Two people, e.g., a therapist and a patient, can exchange forces in a joint therapy session accompanied by video telephony. Two force-based robots that enable hand motor recovery over distant locations via the internet. And because we only need to transmit the slowly changing internal state while the force-control is still done locally by the CSL, there is no noticeable latency.

These were still some open ideas to the conclusion reserved for future works. I hope this work has left you in a contemplative state.

A Schematics

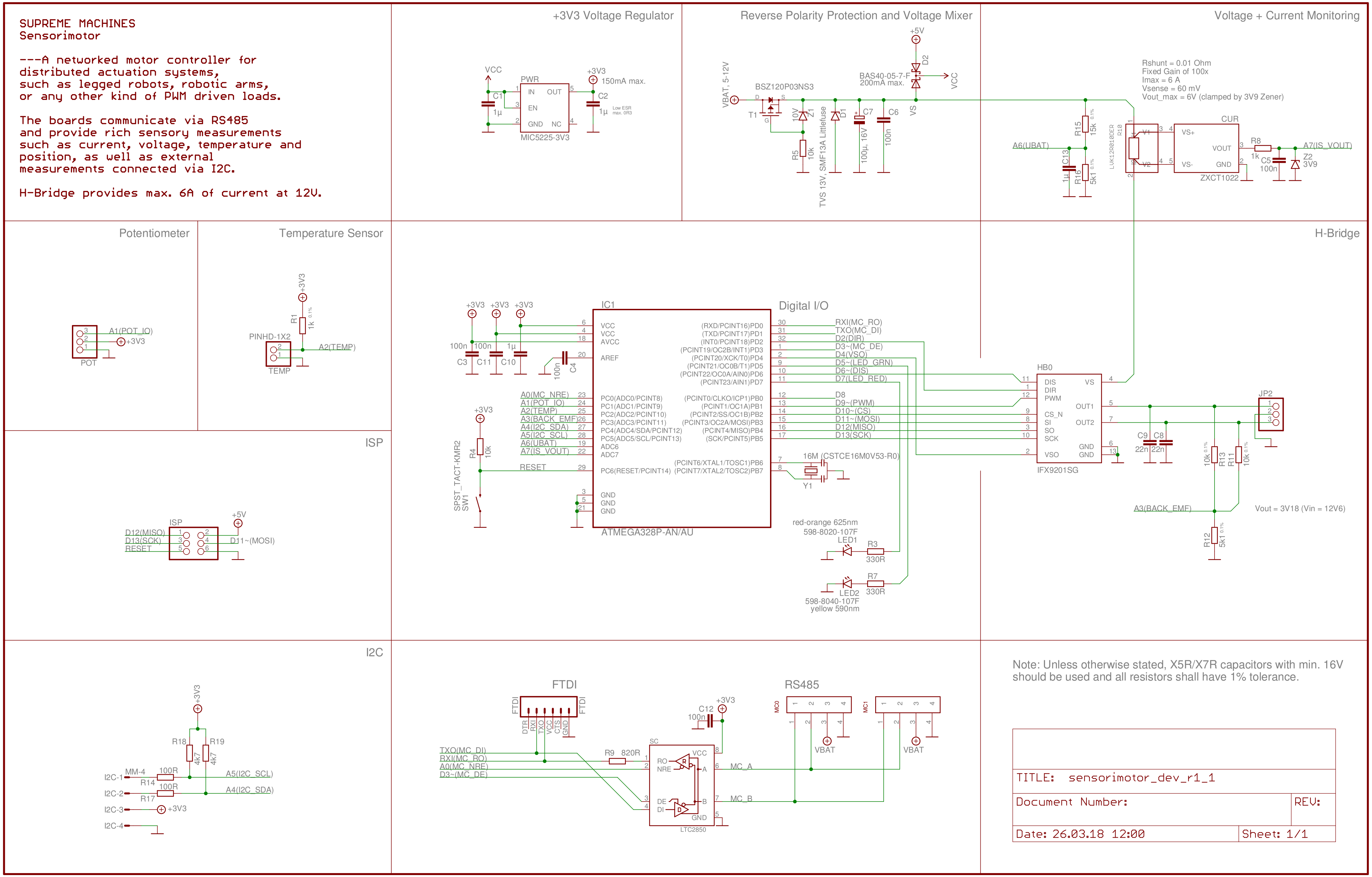
The following schematics are the core of my electronic development described in this thesis. I choose to license these documents under Creative Commons CC BY-SA 4.0. You can find the corresponding firmware in the git repository of Jetpack Cognition Lab: `https://gitlab.com/jetpack-cognition-lab/`

SUPREME MACHINES
Sensorimotor

---A networked motor controller for distributed actuation systems, such as legged robots, robotic arms, or any other kind of PWM driven loads.

The boards communicate via RS485 and provide rich sensory measurements such as current, voltage, temperature and position, as well as external measurements connected via I2C.

H-Bridge provides max. 6A of current at 12V.



Note: Unless otherwise stated, X5R/X7R capacitors with min. 16V should be used and all resistors shall have 1% tolerance.

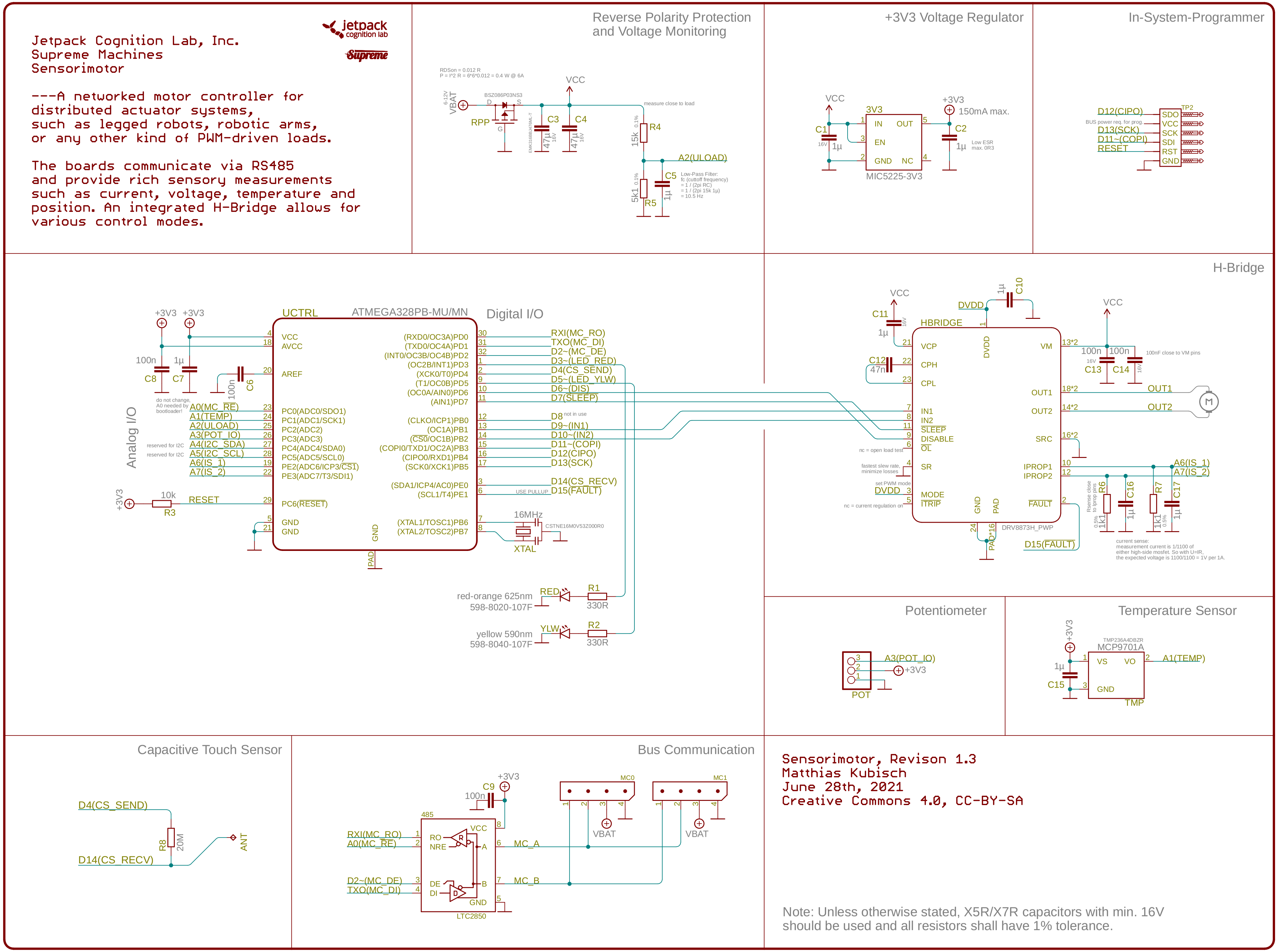
TITLE: sensorimotor_dev_r1_1

Document Number:

REV:

Date: 26.03.18 12:00

Sheet: 1/1



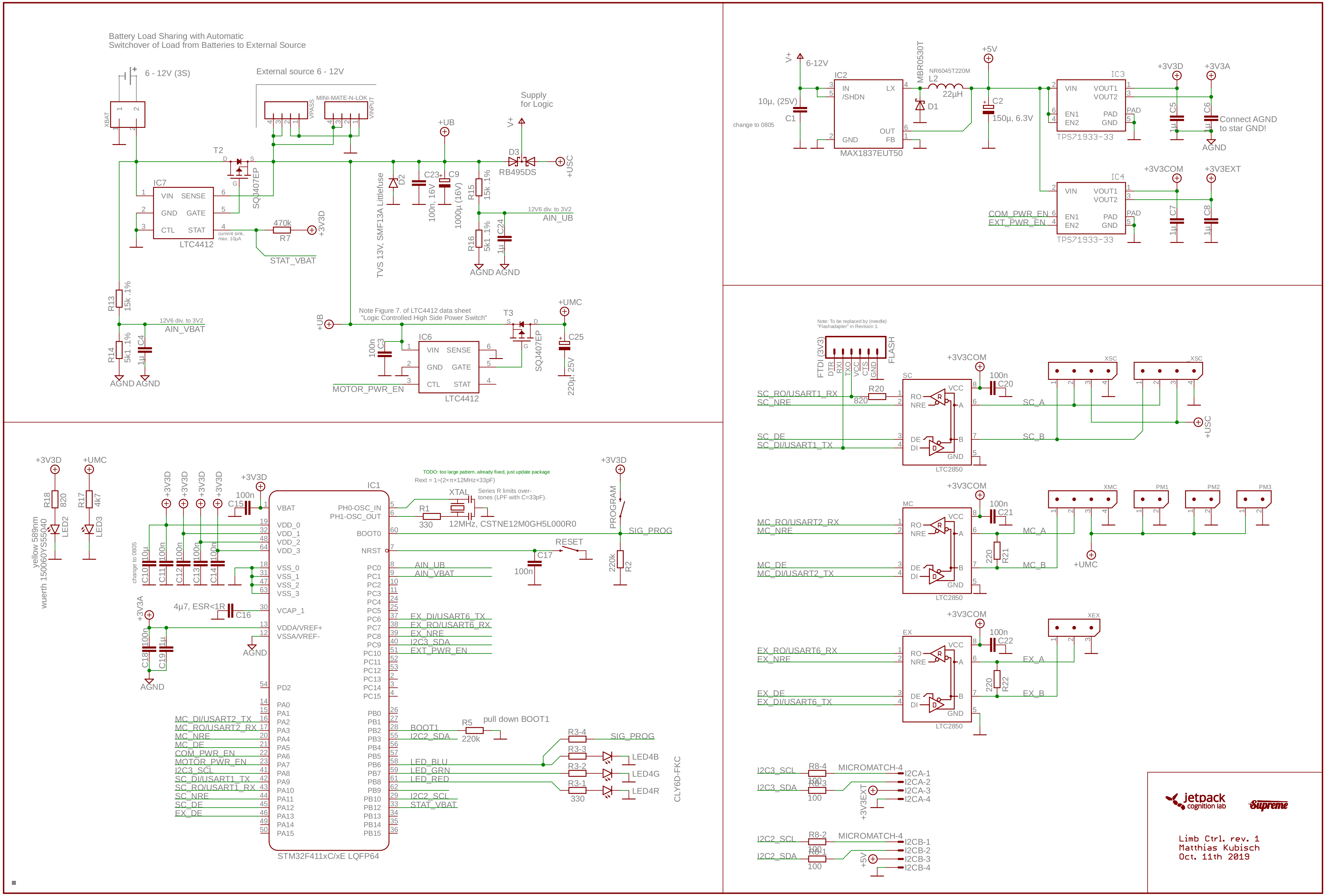
Jetpack Cognition Lab, Inc.
Supreme Machines
Sensorimotor

---A networked motor controller for
distributed actuator systems,
such as legged robots, robotic arms,
or any other kind of PWM-driven loads.

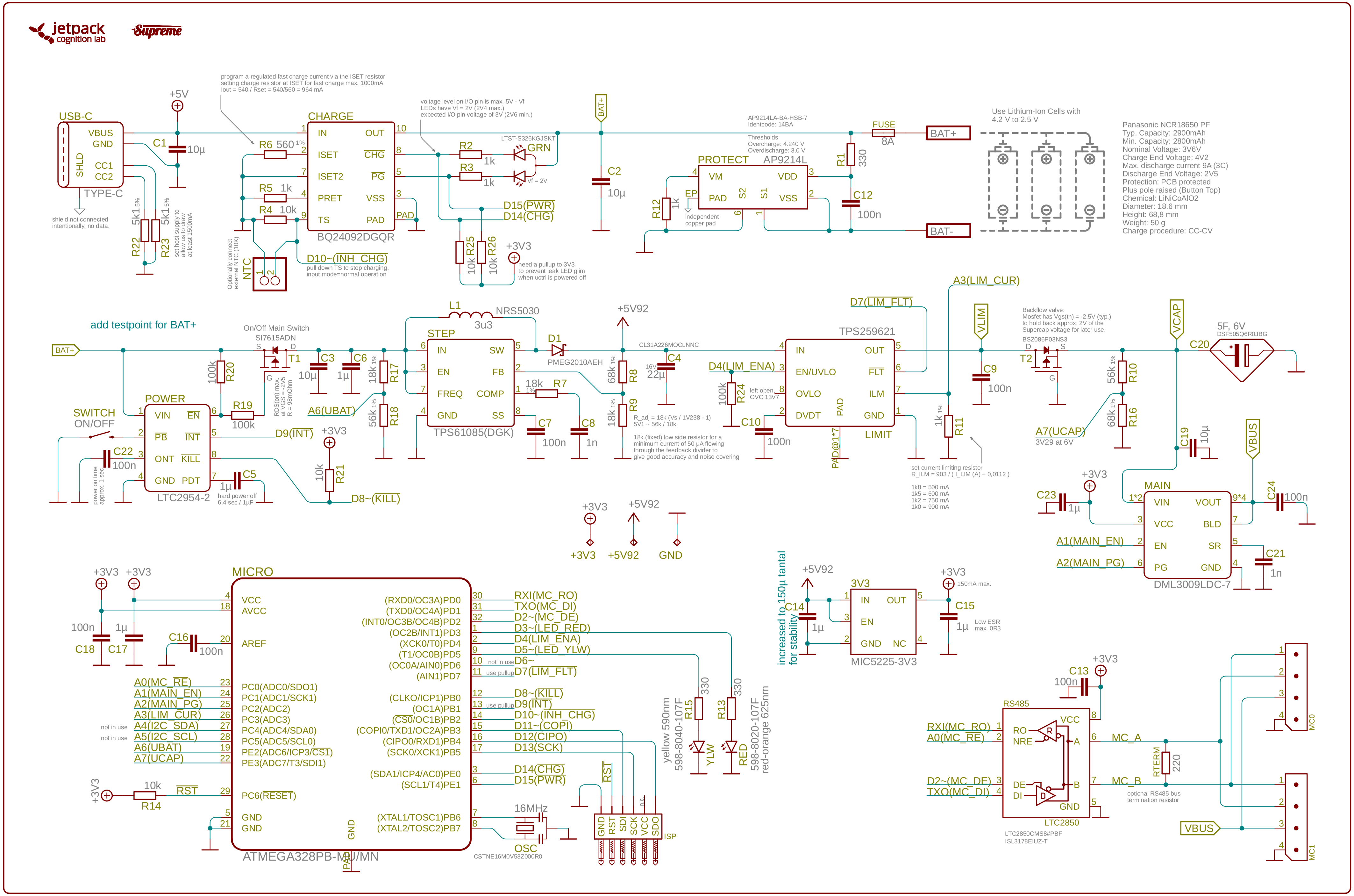
The boards communicate via RS485
and provide rich sensory measurements
such as current, voltage, temperature and
position. An integrated H-Bridge allows for
various control modes.

Sensorimotor, Revison 1.3
Matthias Kubisch
June 28th, 2021
Creative Commons 4.0, CC-BY-SA

Note: Unless otherwise stated, X5R/X7R capacitors with min. 16V should be used and all resistors shall have 1% tolerance.



Limb Ctrl. rev. 1
Matthias Kubisch
Oct. 11th 2019



program a regulated fast charge current via the ISET resistor
setting charge resistor at ISET for fast charge max. 1000mA
Iout = 540 / Rset = 540/560 = 964 mA

voltage level on I/O pin is max. 5V - Vf
LEDs have Vf = 2V (2V4 max.)
expected I/O pin voltage of 3V (2V6 min.)

AP9214LA-BA-HSB-7
Identcode: 14BA

Thresholds
Overcharge: 4.240 V
Overdischarge: 3.0 V

Use Lithium-Ion Cells with
4.2 V to 2.5 V

Panasonic NCR18650 PF
Typ. Capacity: 2900mAh
Min. Capacity: 2800mAh
Nominal Voltage: 3V6V
Charge End Voltage: 4V2
Max. discharge current 9A (3C)
Discharge End Voltage: 2V5
Protection: PCB protected
Plus pole raised (Button Top)
Chemical: LiNiCoAlO2
Diameter: 18.6 mm
Height: 68,8 mm
Weight: 50 g
Charge procedure: CC-CV

shield not connected
intentionally. no data.

set host supply to
allow us to draw
at least 1500mA

Optionally connect
external NTC (10K)

pull down TS to stop charging,
input mode=normal operation

need a pullup to 3V3
to prevent leak LED glim
when uctrl is powered off

independent
copper pad

add testpoint for BAT+

On/Off Main Switch
SI7615ADN

RDS(on) max.
at VGS = -2V5
R = 98mOhm

power on time
approx. 1 sec

hard power off
6.4 sec / 1µF

CL31A226MOCLNNC

R_adj = 18k (Vs / 1V238 - 1)
5V1 ~ 56k / 18k

18k (fixed) low side resistor for a
minimum current of 50 µA flowing
through the feedback divider to
give good accuracy and noise covering

left open
OVC 13V7

set current limiting resistor
R_ILM = 903 / (I_LIM (A) - 0,0112)

1k8 = 500 mA
1k5 = 600 mA
1k2 = 750 mA
1k0 = 900 mA

Backflow valve:
Mosfet has Vgs(th) = -2.5V (typ.)
to hold back approx. 2V of the
Supercap voltage for later use.
BSZ086P03NS3

3V29 at 6V

5F, 6V
DSF505Q6R0JBG

150mA max.

Low ESR
max. 0R3

increased to 150µ tantal
for stability

not in use

use pullup

yellow 590nm
598-8040-107F

red-orange 625nm
598-8020-107F

CSTNE16M0V53Z000R0

optional RS485 bus
termination resistor

LTC2850CMS8#PBF
ISL3178EIUZ-T

Bibliography

Aström, K. J. and Hägelund, T. (1995). *PID-Controllers: Theory, Design, and Tuning.* International Society for Measurement and Control.

Auerbach, J. and Bongard, J. C. (2009). Evolution of Functional Specialization in a Morphologically Homogeneous Robot. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 89–96.

Auerbach, J. and Bongard, J. C. (2010). Dynamic resolution in the co-evolution of morphology and control. In *Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems*, pages 451–458, Cambridge, MA. MIT Press.

Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, 2:53–58.

Bandow, B. and Hartke, B. (2006). Larger water clusters with edges and corners on their way to ice: Structural trends elucidated with an improved parallel evolutionary algorithm. *J. Phys. Chem. A.*, 110 (17):5809—5822.

Banks, M. R., Willoughby, L. M., and Banks, W. A. (2008). Animal-assisted therapy and loneliness in nursing homes: Use of robotic versus living dogs. *Journal of the American Medical Directors Association.*

Bethge, S. (2014). *ABC-Learning: Ein Lernverfahren zur modellfreien Selbstexploration autonomer Roboter.* Institut für Informatik, Humboldt-Universität zu Berlin. Diplomarbeit.

Bharatharaj, J., Huang, L., and Al-Jumaily, A. (2015). Bio-inspired therapeutic pet robots: Review and future direction. In *10th Int. Conf. on Information, Communications and Signal Processing (ICICS).*

Bongard, J. (2011). Morphological change in machines accelerates the evolution of robust behavior. *Proceedings of the National Academy of Sciences*, 108(4):1234–1239.

Bongard, J. (2015). Using robots to investigate the evolution of adaptive behavior. *Current Opinion in Behavioral Sciences*, 6:168–173.

Braitenberg, V. (1984). *Vehicles: Experiments in synthetic psychology.* MIT Press, Cambridge, MA.

Canudas De Wit, C., Olsson, H., Aström, K. J., and Lischinsky, P. (1995). A new model for control of systems with friction. *IEEE Trans. Automatic Control*, 40(3):419–25.

Clune, J., Mouret, J.-B., and Lipson, H. (2013). The evolutionary origins of modularity. *Proceedings of the Royal Society B: Biological Sciences*, 280.

Der, R., Hesse, F., and Martius, G. (2005). Rocking Stamper and Jumping Snake From a Dynamical System Approach to Artificial Life. *Adaptive Behavior*, 14:116.

Der, R. and Martius, G. (2012). *The Playful Machine - Theoretical Foundation and Practical Realization of Self-Organizing Robots*, volume 15.

Der, R., Steinmetz, U., and Pasemann, F. (1999). Homeokinesis - a new principle to back up evolution with learning.

Doya, K. (1993). Bifurcation of Recurrent Neural Networks in Gradient Descent Learning. *IEEE Transactions on Neural Networks.*

Doya, K. (2002). Metalearning and Neuromodulation. *Neural Networks*, 15.

Dörner, D. (2001). *Bauplan für eine Seele.* Rowohlt, Reinbek bei Hamburg.

Eckstein, M., Mamaev, I., Ditzen, B., and Sailer, U. (2020). Calming effects of touch in human, animal, and robotic interaction—scientific state-of-the-art and technical advances. *Frontiers in Psychiatry.*

Elman, J. L. (1990). Finding Structure in Time. *Cognitive Science*, 14(2):179–211.

Fritzke, B. (1995). A Growing Neural Gas Network Learns Topologies. In *Advances in Neural Information Processing Systems 7*, pages 625–632. MIT Press.

Fritzke, B. (1997). A Self-Organizing Network That Can Follow Non-Stationary Distributions. In *Proc. of ICANN-97, International Conference on Artificial Neural Networks*, pages 613–618. Springer.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press.

Grunwald, M. (2017). *Homo hapticus: Warum wir ohne Tastsinn nicht leben können.*

Guckenheimer, J. and Holmes, P. (1983). *Nonlinear Oscillations, Dynamical Systems and Bifurcations of Vector Fields.* Springer, New York.

Hamming, R. W. (1989). *Digital Filters.* Prentice Hall, 3rd edition.

Hansen, N. and Ostermeier, A. (2001). Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, 9(2):159–195.

Haykin, S. (2008). *Neural Networks and Learning Machines.* Prentice Hall, 3 edition.

Hein, D. (2007). Simloid: Evolution of Biped Walking Using Physical Simulation. Diplomarbeit, Department of Computer Science, Humboldt-Universität zu Berlin.

Hein, D., Hild, M., and Berger, R. (2007). Evolution of biped walking using neural oscillators and physical simulation. In *RoboCup: Proceedings of the International Symposium LNAI, Springer.*

Hild, M. (2007). *Neurodynamische Module zur Bewegungsteuerung autonomer mobiler Roboter.* PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin.

Hild, M. and Kubisch, M. (2011). Self-Exploration of Autonomous Robots Using Attractor-Based Behavior Control and ABC-Learning. In *Proceedings of the 11th Scandinavian Conference on Artificial Intelligence*, Trondheim, Norway.

Hild, M., Siedel, T., Benckendorff, C., Kubisch, M., and Thiele, C. (2011). Myon: Concepts and Design of a Modular Humanoid Robot Which Can Be Reassembled During Runtime. In *Proceedings of the 14th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines*, Paris, France.

Hinton, G. E. (1989). Connectionist learning procedures. *Artificial Intelligence*, 40(1-3):185–234.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, pages 1735–1780.

Hoffmann, L. and Krämer, N. C. (2021). The persuasive power of robot touch. behavioral and evaluative consequences of non-functional touch from a robot. *Plos one.*

Holoborodko, P. (2008). Smooth noise robust differentiators.

Holoborodko, P. (2009). One-sided differentiators.

Kamioka, H., Okada, S., Tsutani, K., Park, H., Okuizumi, H., Handa, S., Oshio, T., Park, S.-J., Kitayuguchi, J., Abe, T., Honda, T., and Mutoh, Y. (2014). Effectiveness of animal-assisted therapy: A systematic review of randomized controlled trials. *Complementary Therapies in Med.*

Kanamori, M., Suzuki, M., Oshiro, H., Tanaka, M., Inoguchi, T., Takasugi, H., Saito, Y., and Yokoyama, T. (2003). Pilot study on improvement of quality of life among elderly using a pet-type robot. In *Proc. IEEE Int. Symposium on Comp. Intelligence in Robotics and Automation.*

Kaplan, F. and Oudeyer, P.-Y. (2007). In Search of the Neural Circuits of Intrinsic Motivation. *Frontiers in Neuroscience.*

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR.*

Klyubin, A. S., Polani, D., and Nehaniv, C. L. (2008). Keep your options open: an information-based driving principle for sensorimotor systems.

Koenig, N. and Howard, A. (2004). Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154.

Komosinski, M. (2000). *The World of Framsticks: Simulation, Evolution, Interaction*, pages 214–224. Springer, Berlin.

Krzywicki, D., Stypka, J., Anielski, P., Faber, L., Turek, W., Byrski, A., and Kisiel-Dorohinicki, M. (2014). Generation-free Agent-based Evolutionary Computing. *Procedia Computer Science*, 29 C:1068–1077.

Krzywicki, D., Turek, W., Byrski, A., and Kisiel-Dorohinicki, M. (2015). Massively-concurrent agent-based evolutionary computing. *Journal of Computational Science.*

Kubisch, M. (2008). *Modellierung und Simulation nichtlinearer Motoreigenschaften.* Institut für Informatik, Humboldt-Universität zu Berlin, Berlin. Studienarbeit.

Kubisch, M. (2017). A growing multi-expert structure for open-ended unsupervised learning of sensory state spaces. In *Proceedings of the 7th Joint IEEE International Conference on Development and Learning and on Epigenetic Robotics*, Lisbon, Portugal.

Kubisch, M., Benckendorff, C., and Hild, M. (2011a). Balance Recovery of a Humanoid Robot Using Cognitive Sensorimotor Loops (CSLs). In *Proceedings of the 14th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines*, Paris, France.

Kubisch, M. and Berthold, O. (2022). flatcat – playful robots that respond to touch. In *Proceedings of the 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI 2022), Workshop on Robot Curiosity*, Online (Originally Sapporo, Hokkaido, Japan).

Kubisch, M., Hild, M., and Höfer, S. (2010). Proposal of an Intrinsically Motivated System for Exploration of Sensorimotor State Spaces. In *Proceedings of the 10th International Conference on Epigenetic Robotics*, Örenäs Slott, Sweden.

Kubisch, M., Werner, B., and Hild, M. (2011b). Using Co-Existing Attractors of a Sensorimotor Loop for the Motion Control of a Humanoid Robot. In *Proceedings of the International Conference on Neural Computation Theory and Applications (NCTA).*

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK. Springer-Verlag.

Martius, G., Fiedler, K., and Herrmann, J. M. (2008). Structure from Behavior in Autonomous Agents. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 858–862.

Maujean, A., Pepping, C. A., and Kendall, E. (2015). A systematic review of randomized controlled trials of animal-assisted therapy on psychosocial outcomes. *Anthrozoös.*

Meier, J. (2015). Entwicklung und Implementierung einer interaktiven Verhaltenssteuerung fur den humanoiden Roboter Myon. Masterarbeit, Freie Universität Berlin, Institut für Informatik.

Nolfi, S. and Floreano, D. (2000). *Evolutionary Robotics*. MIT Press, Cambridge.

Olsson, H. (1996). *Control Systems with Friction*. Department of Automatic Control, Lund Institute of Technology.

Olsson, H., Aström, K. J., Gäfvert, M., Wit, C. C. D., and Lischinsky, P. (1998). Friction models and friction compensation. *Eur. J. Control*, page 176.

Oudeyer, P.-Y. and Kaplan, F. (2008). How Can We Define Intrinsic Motivation? In *International Conference on Epigenetic Robotics*.

Oudeyer, P.-Y., Kaplan, F., and Hafner, V. V. (2007). Intrinsic Motivation Systems for Autonomous Mental Development. *IEEE Transactions on Evolutionary Computation*, 11.

Pasemann, F., Hild, M., and Zahedi, K. (2003). SO(2)-Networks as Neural Oscillators. In *Proceedings of the International Work Conference on Artificial and Natural Neural Networks (IWANN)*, pages 144–151.

Polani, D., Martinetz, T., and Kim, J. (2001). An Information-Theoretic Approach for the Quantification of Relevance. In *Advances in Artificial Life*, pages 704–713. Springer Berlin Heidelberg.

Prisacaru, A. (2020). Gretchen – A Humanoid Robot for Research and Education. Bachelorarbeit, Humboldt-Universität zu Berlin.

Rechenberg, I. (1994). *Evolutionsstrategie — Optimieren wie in der Natur*. Springer Berlin Heidelberg.

Rempis, C. W. and Pasemann, F. (2012). An interactively constrained neuro-evolution approach for behavior control of complex robots. In Chiong, R., Weise, T., and Michalewicz, Z., editors, *Variants of Evolutionary Algorithms for Real-World Applications*, pages 305–341. Springer.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: Foundations, pages 318–362. MIT Press.

Schmidhuber, J. (2006). Developmental Robotics, Optimal Artificial Curiosity, Creativity, Music, and the Fine Arts. *Connection Science*, 18(2):173–187.

Schmidhuber, J. (2010). Formal Theory of Creativity, Fun, and Intrinsic Motivation (1990–2010). *IEEE Trans. on Auton. Ment. Dev.*, 2(3):230–247.

Schmitt, O. H. (1938). A thermionic trigger. *Journal of Scientific Instruments*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*.

Selesnick, I. W. (2006). Narrowband Lowpass Digital Differentiator Design. *Electrical and Computer Engineering, Polytechnic University.*

Sezgin, H. (2014). *Artgerecht ist nur die Freiheit: Eine Ethik für Tiere oder Warum wir umdenken müssen.*

Sims, K. (1994). Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 15–22, New York, NY, USA. ACM.

Solomon, J. H., Locascio, M. A., and Hartmann, M. J. (2012). Linear reactive control for efficient 2d and 3d walking over rugged terrain. *Adaptive Behavior.*

Solomon, J. H., Wisse, M., and Hartmann, M. J. (2010). Fully Interconnected, Linear Control for Limit Cycle Walking. *Adaptive Behavior*, 18/6.

Stribeck, R. (1902). Die wesentlichen Eigenschaften der Gleit-und Rollenlager. *Verein Deutscher Ingenieure*, 46:38ff. 1341–1348.

Strogatz, S. (1994). *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering.* Sarat Book House.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction.* MIT Press, second edition.

Tesauro, G. (1995). TD-Gammon: A Self-Teaching Backgammon Program. In *Murray A.F. (eds) Applications of Neural Networks.* Springer, Boston.

Thierens, D. (2002). Adaptive mutation rate control schemes in genetic algorithms. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 1, pages 980–985.

Thompson, J. M. T. and Stewart, H. B. (1986). *Nonlinear Dynamics and Chaos.* Wiley, Chichester U.K.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE.

Toussaint, M. (2006). A Sensorimotor Map: Modulating Lateral Interactions for Anticipation and Planning. *Neural Computation*, 18:1132–1155.

van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605.

Wada, K. and Shibata, T. (2007). Living with seal robots–its sociopsychological and physiological influences on the elderly at a care house. *IEEE Transactions on Robotics*.

Werner, B. (2008). Sensomotorische Erzeugung eines Gangmusters für humanoide Roboter. Studienarbeit.

Werner, B. (2013). Entwicklung eines adaptiven sensomotorischen Algorithmus zur dynamischen Bewegungssteuerung autonomer Roboter. Diplomarbeit, Humboldt-Universität zu Berlin, Institut für Informatik.

Wescott, T. (2014). Controlling motors in the presence of friction and backlash. Technical report.