Parametrizing Motion Controllers of Humanoid Robots by Evolution

Dietmar Schreiner and Clemens Punzengruber
Vienna University of Technology
Institute of Computer Languages, Compilers and Languages Group

dietmar.schreiner@tuwien.ac.at cpunzengruber@gmail.com

Abstract: Autonomous mobile robots are devices that operate within a highly indeterministic environment, the real world. Even worse, robots are physical devices that are part of the real world and hence are inherently nondeterministic by construction w.r.t. mechanical precision and sensor noise. In consequence, robotic control software has to cope with discrepancies between a robot's specification and its de-facto physical properties as achieved in production. Finding feasible parameters for robust motion controllers is a time consuming and cumbersome work. This paper contributes by demonstrating how to utilize an evolutionary process, a genetic algorithm, to automatically find terrain specific optimized parameter sets for off-the-shelf motion controllers of humanoid robots. Evolution is performed within a physical accurate simulation in order to speed up and automate the process of parameter acquisition, while results are devolved to the real devices that benefit noticeably.

1 Introduction

Due to enormous progress in mechanics and electrical engineering, autonomous mobile robots are no longer science fiction. State-of-the-art robotic devices are already applied to mission critical tasks that could not be solved by humans due to inherent danger or hostile environments (e.g., urban search and rescue, deep sea exploration, inspection of nuclear power plants). Moreover, demand for intelligent autonomous devices even in everyday life has tremendously increased. Robotic vacuum cleaners, lawn mowers, or social robots have reached mass production, while domestic housekeepers, personal assistants, or autonomous transportation vehicles will appear in near future.

However, reaching the market introduces additional burdens to those devices: development cost, time to market, and maintenance cost. In consequence, manufacturers have to develop robots that are affordable, have a rather short time to market, and that do require as little maintenance as possible. Therefore, general purpose motion controllers have to be deployed that are robust in a wide range of environments and can cope with material and production related inexactness. Unfortunately, those controllers clearly underperform in various terrains. This disadvantage could be overcome by providing sets of terrain specific controller configurations for defined environments. However, finding these configurations is again time and resource consuming due to the huge variability in environmental factors.

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

Our over-all intention is to automatically generate complex motion controllers via genetic programming. These complex controllers are programs that operate over primitive actions (the off-the-shelf general purpose controllers)[1]. In order to "breed" near optimal complex controllers, the programs' building blocks also have to be optimized. Hence, this paper provides a solution for automatically (and hence cost efficiently) finding adequate parameters for primitive actions for specific environments. We focused on a walk controller for a humanoid robot in presence of varying floor materials, but our findings can easily be transferred to other types of controllers and robots.

To find optimized parameters, a genetic algorithm as a specialized type of blind search was used, as we did not get a precise specification of the robot's control algorithm from the manufacturer. Hence, we had to treat the controller itself as a black box, which can be fed with certain inputs via a well defined API, and that in return calculates output which is directly fed into the robot's actuators. A physical accurate simulation of the robot and its environment as much as a simulation of the robot's middleware (provided by the manufacturer) were used to evaluate generated parameters for specific floor materials in terms of evolutionary fitness.

The remainder of this paper is structured as follows: Section 2 discusses related work and outlines the context for the work presented in this paper. Section 3 provides a short introduction to the concept of genetic algorithms and how one was used for our specific purpose. Section 4 provides an outline of the robot we used for our experiments, its software architecture and the simulation that came to use for the simulated process of evolution. In Section 5 we describe our experimental setup and provide measurements that prove our approach valid. Finally, Section 6 provides our conclusion, as much as an outlook to our future work.

2 Related Work

Davidor [Dav91] outlines three potential fields of application for genetic algorithms in robotics: (i) the system design (mechanics, electrical control etc.), (ii) programmable hardware parameters for individual systems, and (iii) trajectories and motion programs for individual applications. As our work is related to computer science, systems design is off-topic and hence is not considered. Our over-all work is related to genetic programming [Koz10, Cra85] for complex robotic motion controllers. Therefore, we generally contribute to the third field. However, this paper's contribution deals with optimizing parameters for primitive motion controllers (the building blocks of complex motion programs that come to use in our genetic programming efforts) and hence is part of the second category.

Arakawa and Fukuda [AF96] demonstrate how to generate motion trajectories for a biped forward walk. Like the work described in this paper, their approach mainly contributes to the second field of applications of genetic algorithms, the parametrization of programmable hardware. It is based on a kinematic model of the robot and a given reference

[1] including ϵ (NOP)

trajectory (a spline function) of joint related parameters. The trajectory itself is optimized with a genetic algorithm in terms of energy consumption. In contrast, our approach aims at the optimization of motion trajectories without knowledge of the robot's kinematic model or the mathematical model of reference trajectories and hence is well suited to optimize black boxed off-the-shelf motion controllers.

In [Ger99, ARPGMP$^+$04] genetic algorithms are used for off-line path planning. A full path for the whole robot between two locations within the real world is optimized. Hence, these papers contribute to the third field of application of genetic algorithms. The genome used in both papers encodes geometric information of a path: waypoint coordinates in [Ger99], distances and angles in [ARPGMP$^+$04]. We also aim at optimized paths for a humanoid robot, However, we do not evolve paths and trajectories but control parameters for low level motion controllers (described in this paper) and motion programs made of these low level controllers (out of scope of this paper). Hence, the genomes used in this paper do not encode path and trajectory specific values but controller specific constraints and parameters.

3 Evolutionary Computations

The principle of genetic algorithms was developed in the mid seventies of the last century and is based on the concept of evolution discovered by Charles Darwin. Solutions for a given problem are considered to be individuals that compete within a process of reproduction and survival. Therefore, each solution is encoded into a genome that fully represents the solutions properties. Like in nature, all individuals compete to survive in a process of selection. Only competitive individuals will survive or even reproduce, in order to create even more competitive offsprings. In contrast to evolutionary algorithms, genetic algorithms like introduced in [Hol73, Hol75] utilize genetic operators to recombine genomes during sexual reproduction [Mit98], as much as the concept of generations, to simulate natural evolution in a more realistic way. The quality and hence the probability to survive and to reproduce of an individual is determined by a so called fitness-function. This function provides a quality measure for the solution of the given problem and is one key element of genetic algorithms.

3.1 Genetic Algorithm

Algorithm 1 outlines the basic structure of a genetic algorithm like the one used for our work described within this paper. At Line 2 a randomized start population consisting of s_P individuals is generated. The evolutionary process for one generation is executed between Line 3 and Line 17 and is repeated as long as the quality of evolved solutions (the individuals) is not sufficient and hence lies below the minimal required quality ε, or until the maximum number of generations Ω is exceeded. In Line 6 two individuals are randomly selected from the actual population. The process of selection takes each individ-

ual's fitness into consideration: the better the individual fitness, the higher the probability to be selected. The block from Line 7 to Line 11 denotes the genetic operaton *crossover*: from two parents (f,m), two offsprings (c1,c2) are generated. With respect to the crossover probability p_c the genes of the parents are crossed over, or stay untouched producing two perfect clones. Thereafter, both offsprings are subject to mutation, which is again done with respect to the mutation probability p_m in Line 12 and Line 13. Finally, the next generation is calculated in Line 16, by sorting all individuals (parents and offsprings) in order of their individual fitness, and selecting the best s_P ones while eliminating bad ones.

1: $gc = 0$
2: $Population =$ randomPopulation(s_P)
3: **repeat**
4: $NextGeneration \leftarrow \{\}$, gc++
5: **for** $k = 0$ to $s_P/2$ **do**
6: f, m = selectParents($Population$)
7: **do probably**(p_c)
8: $c1, c2 =$crossover(f,m)
9: **or**
10: $c1, c2 = f, m$
11: **end do**
12: **do probably**(p_m) mutate($c1$)
13: **do probably**(p_m) mutate($c2$)
14: $NextGeneration \leftarrow NextGeneration \cup \{c1, c2\}$
15: **end for**
16: $Population \leftarrow$selectBest($Population \cup NextGeneration, s_P$)
17: **until** $gc \geq \Omega$ **or** bestFitness($Population$) $\geq \varepsilon$

Algorithm 1: Genetic algorithm as used in our approach

3.2 Genetic Operators

The heart of each genetic algorithm are its genetic operators: selection, crossover, and mutation. By carefully selecting proper versions of these operators, quality of solutions and the over-all runtime of the genetic algorithm have to be balanced.

3.2.1 Selection

As implied by its name, the selection operator chooses those individuals of a population that may reproduce in order to build the next generation. In general high quality individuals should be selected to improve the individuals over generations. However, keeping a certain diversity is of great importance to avoid convergence at local optima. Hence, a proper selection operator will favor superior individuals but will not exclude inferior ones from reproduction. In literature, various selection operators have been evaluated and discussed.

Some improve the runtime performance of a genetic algorithm, others guarantee higher diversity.

For our approach, we combined two well known operators creating the σ-operator, which provides lower execution times but also establishes a high diversity. The genetic algorithm used for our work utilizes two parent individuals in order to create two offsprings. In order to combine two selection operators, the two parent individuals are selected by distinct operators. Algorithm 2 sketches this process.

```
1: function selectParents(Population) returns f,m
2:     f = tournamentSelection(Population)
3:     m = truncationSelection(Population)
4:     return f,m
5: end function
```

Algorithm 2: Selection operation as used in our approach

The first selection operator that comes to use is called *tournament selection* [GD91]. In contrast to most selection operators tournament selection does not require the calculation of fitness values for the whole population. Instead it randomly chooses a subset of individuals, the so called tournament. To find the winner of the tournament, fitness values for this small subset of individuals has to be calculated only. The process of creating tournaments and finding the winners is repeated as often as parent individuals have to be selected.

The second selection operator is that of *truncation selection*. Operators of this type select a predefined number of parents μ that produce a fixed number of offsprings λ. The so called (μ, λ) selection operator chooses the μ best offsprings as individuals for the next generation, while the actual generation is eliminated. A modified version of this operator that is used in our work is called $(\mu + \lambda)$ selection operator. It chooses the μ individuals for the next generation from the set union of the actual generation and the offsprings.

3.2.2 Crossover

In sexual reproduction multiple (typically two) individuals are used to procreate offsprings. The parents' genomes are recombined in order to "construct" new individuals that unite properties of their parents. As two full genomes–the ones of the parent individuals–have to be combined into one genome per offspring, they are typically spitted into fragments that are randomly combined. Typical crossover operators split genomes in two (*Single-point Crossover*) or multiple halves (*Multi-point Crossover*). Our approach uses a *Parameterized Uniform Crossover* operator [SJ91], which uses separate crossover probabilities for each locus of the genome, and hence allows fine grained control of the reproduction process.

3.2.3 Mutation

In each iteration of the genetic algorithm defined in Algorithm 1 the genetic operation for mutation is applied to each "newborn" individual with a given probability p_m. This mutation operator is used to reduce the risk of getting stuck in local extrema with the performed

blind search by adding randomness to temporary results. Mutation on the one hand has to be used very carefully in terms of frequency and impact. Only a small number of individuals should undergo mutation as genetic algorithms provide good means of diversification by sexual reproduction via the crossover operator. The other important issue on mutation is the impact: altering information, for example flipping a bit, has to be done at semantically defined locations. Like in molecular biology, these locations are called *locus* and cluster all symbols that represent one specific fact. A locus of a byte value for example denotes a sequence of 8 bits within a bit-streamed chromosome. If this locus has to be mutated, only one of those eight bits must be affected by the alteration.

```
1: function mutate(individual)
2:     genome = getGenome(individual)
3:     for locus in genome
4:         do probably(p_l) locus = randomValue()
5:     end for
6: end function
```

Algorithm 3: Mutation operation as used in our approach

Algorithm 3 schematically denotes the mutation operator as implemented for our work. The loop starting at Line 3 iterates over all loci of an individual's genome. Mutation is done in Line 4, where each locus may be affected by a separate probability of mutation p_l that is independent from all other locis' probabilities. If a locus is mutating in accordance to the probability function, it is simply overwritten by a semantically correct but randomized fact. Randomization can be pure random or can be a randomized deviation of the original locus, thus producing changes that are located within the original's neighborhood.

4 Robotic System

The work described in this paper is motivated by our work with standardized "near mass production" devices, the Aldebaran Nao robots[2] [GHB+08]. This product is currently used in academic research and education, and hence has been manufactured in numbers of several hundreds. However, the company aims at mass market, and thus much larger lot, which underlines our considerations of cost and time to market. The following sections provide a summary of the robots' hardware and software architecture.

4.1 Hardware

The Nao is a two-legged robot, which is $58cm$ tall and weighs approximately $4.3kg$. The robot has a total of 21 degrees of freedom, where a degree of freedom denotes a joint that can be changed within one dimension by the robot's actuators. The Nao's actuators

[2] http://www.aldebaran-robotics.com/

are driven by two types of motors: type one operates at a nominal speed of $6330rpm$ by a nominal torque of $12.3mNm$, type two operates at a nominal speed of $8810rpm$ by a nominal torque of $3.84mNm$.

The motherboard contains a $500MHz$ x86 AMD Geode CPU, $256MB$ SDRAM, and $2GB$ flash memory, and is operated by a custom based distribution of Open Embedded Linux (32 bit x86 ELF).

4.2 Software

As the Nao robot is delivered with a proprietary middleware based framework for service oriented applications, namely NaoQi, the software implemented for our experiments was designed w.r.t. the service oriented computing paradigm and hence is based on SOAP [NY07]. NaoQi provides a service container for application services as much as abstraction of the robot's operating system, the firmware, and the device drivers. In addition, NaoQi includes a set of behavioral services, like motion primitives. The walk controller and its parametrization targeted by our work here is one of those black-box low level motion primitives supplied by Aldebaran. The robotic application itself–artificial intelligence, computer vision, and complex motions–is implemented as modules issuing services at the application layer

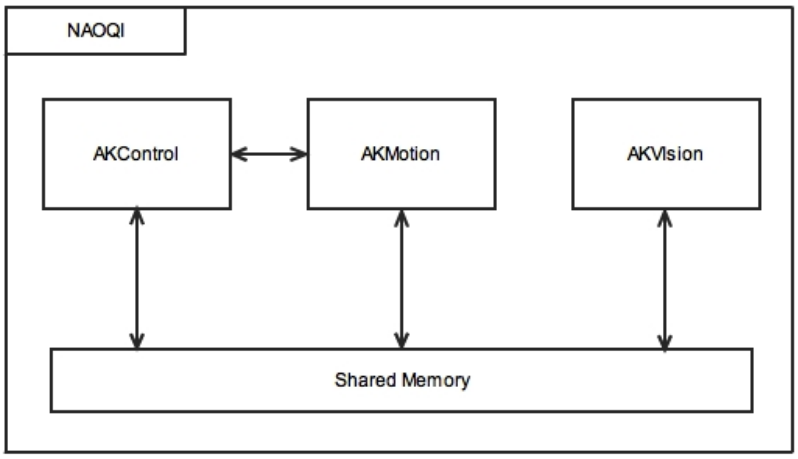

Figure 1: Software architecture

Figure 1 depicts an overview of this system: The application modules are embedded into a NaoQi container. Thus, calls to low level functionality as much as service requests are issued via the NaoQi SOAP interface. Following main application modules are relevant for the experiments described within this paper:

***AKControl*:** This module's main responsibility is the AI, as much as motion planning and scheduling.

***AKMotion*:** The motion module executes and monitors motion primitives in order to perform complex motions. High-level motion commands like *walk to position* are split down to primitive commands that are then issued to the NaoQi middleware in order to feed low level motion controllers. The parameter sets evolved by the genetic algorithm proposed in this paper are injected here.

***AKVision*:** To monitor motion efforts sensor feedback is required by the motion module. The vision module provides this information and stores it into shared memory for further use.

5 The Experiment

As described in the introduction of this paper, the off-the-shelf motion controller parametrization is not competitive for most floor types due to its universality. Consequently, the process of parametrization has to be done for each device on each considerable ground type. In absence of a feasible automated solution for this problem, in a first try this work was manually done and took several months to get a stable but competitive walk for 5 robots on a small set of floor types. To overcome this resource consuming and dull task, the solution to evolve parameter sets for the motion controller via a genetic algorithm was finally developed.

5.1 Setup

In a genetic algorithm, a population has to be evaluated. Hence, the fitness of all individual has to be calculated for each generation. For our experiment precision and speed of a parametrized walk have to be measured in order to asset its fitness. As early experiments showed, measuring one individual's fitness takes up to 20 seconds of runtime and about 30–40 seconds setup time for the robot, and several measures have to be done for one evaluation to calculate a satisfying averaged value. This leads to an overall time of approximately 3 minutes required per individual of one generation. It is obvious that this number is too large for manually conducted experiments with adequate population sizes and a sufficient number of generations. Therefore, the evolutionary process has been carried out within a physically accurate simulation, introducing a noticeable speed up in evaluating the individuals' fitness in absence of a human operator.

The developed software framework hence integrates (i) the simulator and a vendor supplied binary of the robots kernel for adequate simulation, (ii) an existing framework for genetic algorithms, (iii) and the robotic application software described in Section 4.2.

ISBN 978-3-88579-286-4

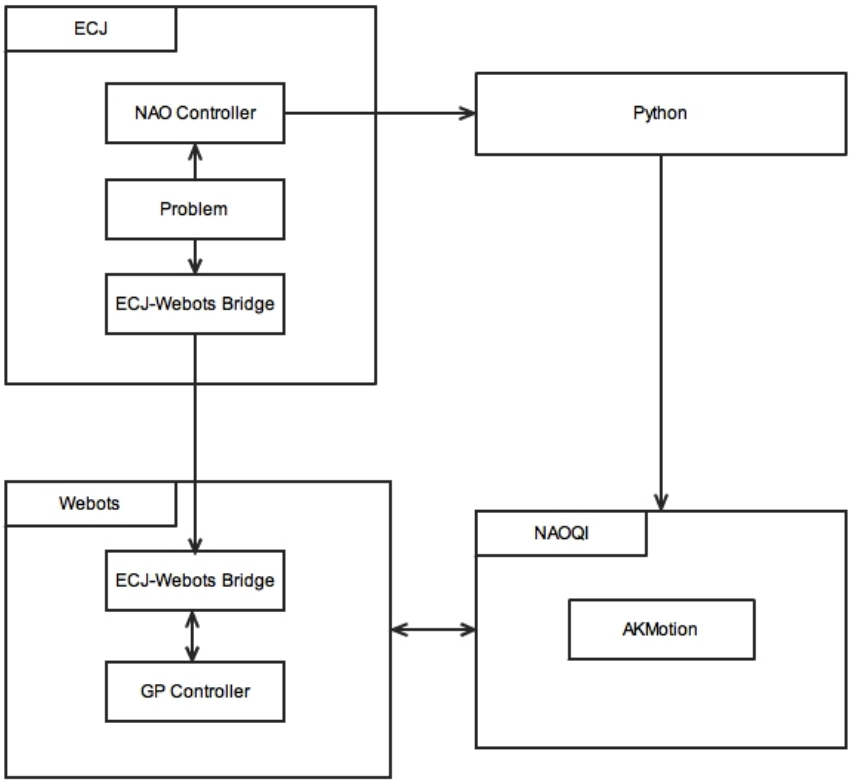

Figure 2: Software architecture for evolutionary simulation

For the robotic simulation, the Cyberbotics Webots simulator[3] was chosen [LT05, Mic98]. The simulator is based on the open source physics engine ODE[4] that is rather robust, customizable, and accurate in terms of precision and runtime performance. In addition, Aldebaran provides correct models of the Nao robot and its controllers for Webots. For the genetic algorithm, the ECJ framework[5] was used, as it is well known and accepted in academia.

Figure 2 depicts the architecture of our simulation framework: An external NaoQi core (a robot kernel cross compiled for the computer hosting the simulation) is connected to the Webots simulator via the Webots interface and the Nao models provided by Aldebaran. In that way, a physically accurate simulation of the Nao robot can be achieved. The ECJ framework is connected to the simulator via a bridge component that implements bidirectional communication over network sockets. The *problem* that has to be solved by the

[3] http://www.cyberbotics.com/products/webots/
[4] http://www.ode.org/
[5] http://www.cs.gmu.edu/~eclab/projects/ecj/

genetic algorithm in addition is connected to the NaoQi core via a python bridge, and is able to feed parameter sets directly into the robot's (simulated) motion module.

As the NaoQi middleware provides a set of motion primitives that are related to the robot's walking ability, not only a straight walk was parametrized but the full set of available controllers: (i) *Walk Forward*, (ii) *Walk Backward*, (iii) *Walk Left*, (iv) *Walk Right*, (v) *Turn Left*, and (vi) *Turn Right*.

5.2 Settings

Following sections specify the parameters we used for the robotic simulation and for the genetic algorithm.

5.2.1 Robotic Simulation

The Webots simulator Version 6.2.4 was used at its default settings in conjunction with the original Nao models from Aldebaran as much as the original NaoQi 1.2 core.

As simulated results have to be used on real robots and the real world, the physical friction coefficients for floor materials had to be determined for the simulation. For our experiment these values have manually been acquired and compared to the ones used by the simulator. Results showed that the coefficients used were sufficient for the desired precision (deviations of less than a millimeter are negligible due to tolerances within the robot's hardware).

5.2.2 Genetic Algorithm

For the genetic algorithm, a genome was developed that contains 35 loci, each representing one of the 35 parameters relevant for the walk controller. Properties of interest for the genome denoted in Table 1 were for example step length, step height, magnitude of arm movements, and damping coefficients in terms of stiffness of relevant joints[6].

The settings for the genetic algorithm that were used to produce the results provided in Section 5.3 are denoted in Table 2. In accordance to our observations, major improvements are achieved within the first 20 generations, therefore the maximum number of rounds (*max*) for the genetic algorithm was limited to 20. Evolution beyond that point did not bring any noticeable changes for the individuals fitness. For the process of mutation new values are generated within a given neighborhood (ϕ) around the original value instead of using pure random values in order to keep noise low.

The fitness of each evolved parameter set is evaluated at a given ground material by letting a robot exactly walk a predefined distance. The time taken and the final position (the exact distance traveled) of this walk are then used to calculate the corresponding fitness

[6]By controlling the stiffness of every single joint, the overall elasticity and flexibility of the robot can be controlled.

ISBN 978-3-88579-286-4

Locus	Parameter	min. Value	max. Value
0	MaxStepLength	0.001	0.09
1	MaxStepHeight	0.001	0.08
2	MaxSideStepLength	0.001	0.06
3	MaxTurnAngle	0.001	1.0
4	zmpOffsetX	0.001	0.05
5	zmpOffsetY	-0.05	0.05
6	pLHipRollBacklashCompensator	0	15
7	pRHipRollBacklashCompensator	-15	0
8	pHipHeight	0.001	0.9
9	pTorsoYOrientation	-20	20
10	pShoulderMedian	1.5	1.9
11	pShoulderAmplitude	0.1	0.5
12	pElbowAmplitude	1.5	1.9
13	pElbowMedian	0.1	0.5
14	pArmsEnable	0.0	1.0
15	pLipYawPitchStiffness	0.0	1.0
16	RHipYawPitchStiffness	0.0	1.0
17	LKneePitchStiffness	0.0	1.0
18	RKneePitchStiffness	0.0	1.0
19	LHippRollStiffness	0.0	1.0
20	RHipRollStiffness	0.0	1.0
21	LHipPitchStiffness	0.0	1.0
22	RHipPitchStiffness	0.0	1.0
23	LAnkleRollStiffness	0.0	1.0
24	RAnkleRollStiffness	0.0	1.0
25	LAnklePitchStiffness	0.0	1.0
26	RAnklePitchStiffness	0.0	1.0
27	LShoulderRollStiffness	0.0	1.0
28	RShoulderRollStiffness	0.0	1.0
29	LShoulderPitch Stiffness	0.0	1.0
30	RShoulderPitchStiffness	0.0	1.0
31	LElbowYawStiffness	0.0	1.0
32	RElbowYawStiffness	0.0	1.0
33	LElbowRollStiffness	0.0	1.0
34	RElbowRollStiffness	0.0	1.0
35	SampleRate	18	27

Table 1: Genome for the parameter set

Parameter		Value
λ	Number of offsprings	150
μ	Number of parents	15
Ω	Number of generations (max)	20
p_c	Crossover probability	100%
p_m	Mutation probability (discrete uniform distribution)	20%
ϕ	Deviation of mutation	0.1

Table 2: Parameters for the genetic algorithm

value in accordance to Formula 1 where m is the number of consecutive tries with the same parameter set (for the experiment we defined $m = 3$), n is the actual try, Δd_n is the deviation from the distance to travel (the difference of the distance that was requested and the way that was de facto walked), and t_n is the time that try n took to reach the final position. For our experiments the robots were programmed to cover a distance of exactly $75cm$.

$$Fitness = \begin{cases} \sum_{n=1}^{m} \Delta d_n + \frac{t_n}{1000}, & \text{if } t_n < t_{TIMEOUT} \\ \top, & \text{if } t_n \geq t_{TIMEOUT} \end{cases} \quad (1)$$

5.3 Results

Our approach was able to evolve robust and performant parameter sets for all six general purpose motion controllers supplied by the robot's manufacturer. Using the evolved parameter sets, the robots have gained an average speedup of 12.1%. At the same time precision increased by 43.1%. This value is a relative value denoting the improvement of precision compared to the precision of the original controller. The best improvement was found for the *Walk Forward* parameter set, where by an overall walked distance of $75cm$ our parameters dislocated the robot for $0.55cm$ while the original parameters lead to a displacement of $1.04cm$.

Table 3 shows the runtimes of the genetic algorithm for the six controllers. As the backward walk is the slowest of all walks (due to mechanical issues) the genetic algorithm for this specific walk took the most runtime, while the walk forward was the fastest one.

Finally, Figure 3 and Figure 4 depict the results for the *Walk Forward* controller. Measurements for the other controllers look similar and are omitted here due to the limited space within this paper: Figure 4 shows the overall development of the population (denoted as *Total Moving*) and the development of stable solutions (denoted as *Total Stable*). Individuals that did not move the robot have been considered to be dead by construction and have been removed before "birth". Individuals that did not cause the robot to fall have been considered to be stable. As one can see, insufficient solutions have been removed

ISBN 978-3-88579-286-4

Motion Controller	Runtime	Motion Controller	Runtime
Walk Forward	32 h 36 min	Walk Backward	45 h 31 min
Walk Left	39 h 52 min	Walk Right	37 h 23 min
Turn Left	34 h 09 min	Turn Right	33 h 03 min

Table 3: Execution times of the genetic algorithm

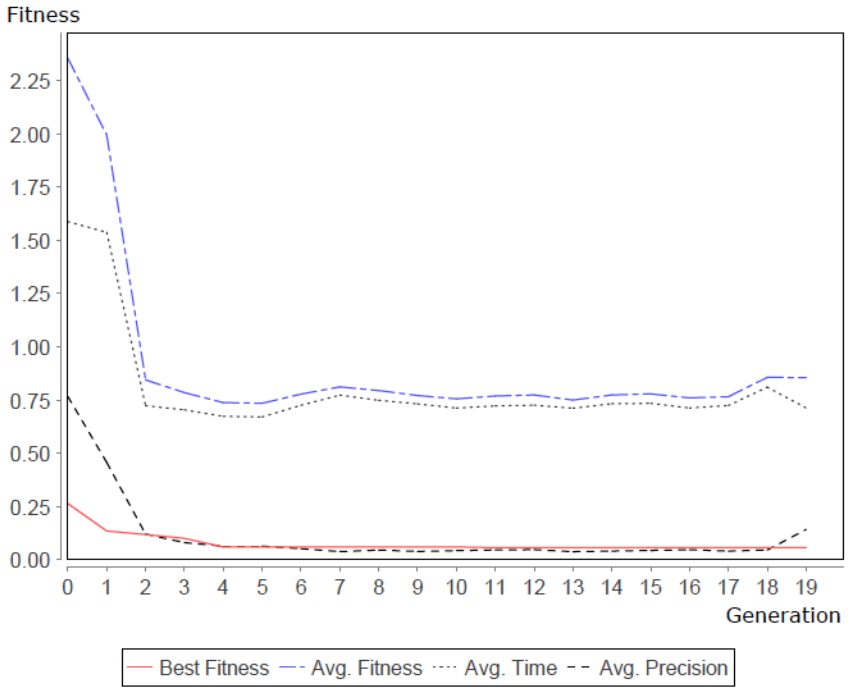

Figure 3: Evolution of *Walk Forward*

from the population's gene pool rather fast: in the first generation, only 3 individuals have been stable, while after generation 4 the number of stable individuals was between 78 and 129. Another effect observed was the fluctuation of stable individuals, which varied from generation to generation, but stayed on a rather high level after generation 5.

The corresponding fitness value for the best individual of a generation as much as the average fitness of the whole generation is depicted in Figure 3. The diagram also shows the average time and the average precision[7], which are used to calculate the fitness value. This shows the interaction of speed and precision: At generation 18 for example, speed is increased (time taken is reduced) while precision is reduced.

[7]The y-axis for time and precision is not labeled as these values are only depicted to visualize dependencies. Time range is $[0, 20]$ seconds, precision range is $[0, 20]$ cm.

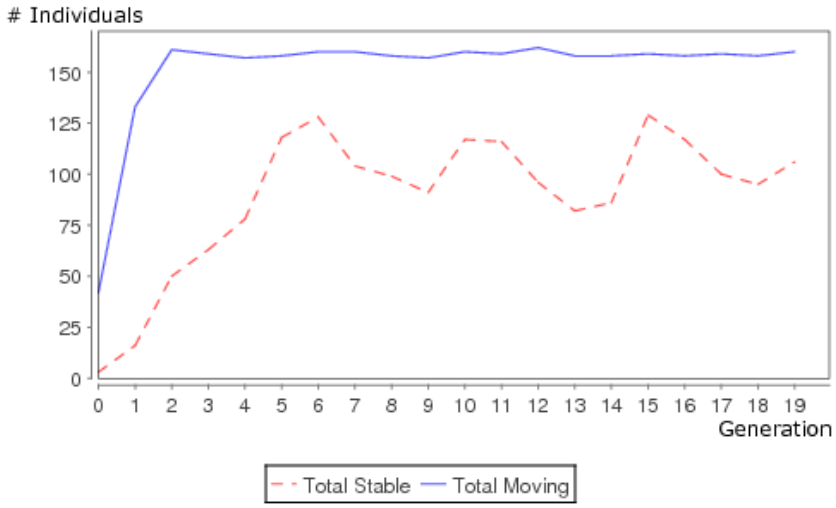

Figure 4: Development of population for *Walk Forward*

6 Conclusion

This paper demonstrates how to automatically calculate improved parameter sets for black boxed, off-the-shelf motion controllers of humanoid robots by a genetic algorithm. The evolved parameter sets outperform custom parameters supplied by the manufacturer in terms of motion speed and precision, and hence can replace the original parameter set for each covered floor material.

The evolutionary process is shifted to a physically accurate robotic simulation and thus can be run automatically without human interaction. Consequently, a rather huge material library can be calculated with reasonable effort. In addition, our experiments show that results from a proper simulation of the Aldebaran Nao robot can be used on real robots without a noticeable loss of precision and performance. The average precision of a simulated robot has been increased by 43.1% while an average speedup of 12.1% was gained. At a real robot precision deviated from the simulation results and was increased by 19.2% only, while the gained speedup was nearly the same.

Measurements prove that excellent results for parametrizing the robot's custom motion controllers already converge after 20 rounds (generations) of the genetic algorithm. However, it is also observable that good enough results can be achieved even after 5 generations. This fact can be useful, when generating huge material libraries.

Our future work on evolutionary processes for humanoid robots aims at genetic programming of motion controllers [Koz10, Cra85]. For this approach we not only try to evolve parameter sets for motion primitives, but also full programs for complex composed motions. First results are promising, and show great potential for future improvements.

References

[AF96] T. Arakawa and T. Fukuda. Natural motion trajectory generation of biped locomotion robot using genetic algorithm through energy optimization. In *Systems, Man, and Cybernetics, 1996., IEEE International Conference on*, volume 2, pages 1495 –1500, oct 1996.

[ARPGMP+04] V. Ayala-Ramirez, A. Perez-Garcia, F.J. Montecillo-Puente, R.E. Sanchez-Yanez, and E. Martinez-Labrada. Path planning using genetic algorithms for mini-robotic tasks. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 4, pages 3746 – 3750, oct. 2004.

[Cra85] Nichael Lynn Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187. Lawrence Erlbaum Associates, 1985.

[Dav91] Yuval Davidor. *Genetic Algorithms and Robotics: Genetic Algorithms and Robotics - A Heuristic Strategy for Optimism*. World Scientific Pub Co, 1991.

[GD91] David E. Goldberg and Kalyanmoy Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In *Proceedings of the First Workshop on Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[Ger99] M. Gerke. Genetic path planning for mobile robots. In *American Control Conference, 1999. Proceedings of the 1999*, volume 4, pages 2424–2429, 1999.

[GHB+08] David Gouaillier, Vincent Hugel, Pierre Blazevic, Chris Kilner, Jérôme Monceaux, Pascal Lafourcade, Brice Marnier, Julien Serre, and Bruno Maisonnier. The NAO humanoid: a combination of performance and affordability. *CoRR*, abs/0807.3223, 2008.

[Hol73] John H. Holland. Genetic Algorithms and the Optimal Allocation of Trials. *SIAM J. Comput.*, 2(2):88–105, 1973.

[Hol75] John H. Holland. *Adaption in natural and artificial systems*. The University of Michigan Press, Ann Arbor, 1975.

[Koz10] John R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):251–284, 2010.

[LT05] Vajta Laszlo and Juhasz Tamas. The Role of 3D Simulation in the Advanced Robotic Design, Test and Control. In Vedran Kordic, Aleksandar Lazinica, and Munir Merdan, editors, *Cutting edge robotics*. pro Literatur, 2005. ISBN: 3-86611-038-3.

[Mic98] Olivier Michel. Webots: Symbiosis Between Virtual and Real Mobile Robots. In Jean-Claude Heudin, editor, *Virtual Worlds*, volume 1434 of *Lecture Notes in Computer Science*, pages 254–263. Springer, 1998.

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, third printing edition, 1998. ISBN: 9780262631853.

[NY07] Mitra Nilo and Lafon Yves, editors. *SOAP Version 1.2*. W3C, 2007.

[SJ91] William M. Spears and Kenneth A. De Jong. On the Virtues of Parameterised Uniform Crossover. In *Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, USA, July 1991*, pages 230–236. Morgan Kaufmann, 1991.