On the Tradeoff between Hardware Protection and Optimization Success: A Case Study in Onboard Evolutionary Robotics for Autonomous Parallel Parking

Mostafa Wahby and Heiko Hamann
Department of Computer Science,
University of Paderborn,
Paderborn, Germany
mostafaw@mail.uni-paderborn.de,
heiko.hamann@uni-paderborn.de

January 21, 2015

Abstract

Making the transition from simulation to reality in evolutionary robotics is known to be challenging. What is known as the reality gap, summarizes the set of problems that arises when robot controllers have been evolved in simulation and then are transferred to the real robot. In this paper we study an additional problem that is beyond the reality gap. In simulations, the robot needs no protection against damage, while on the real robot that is essential to stay cost-effective. We investigate how the probability of collisions can be minimized by introducing appropriate penalties to the fitness function. A change to the fitness function, however, changes the evolutionary dynamics and can influence the optimization success negatively. Therefore, we detect a tradeoff between a required hardware protection and a reduced efficiency of the evolutionary optimization process. We study this tradeoff on the basis of a robotics case study in autonomous parallel parking.

1 Introduction

In evolutionary robotics [Nolfi and Floreano, 2000, Bongard, 2013], simulations are a key tool due to limited resources and costly robot experiments. However, if simulations are overused, key challenges of evolutionary robotics might be either overlooked or difficult to resolve [Nelson et al., 2009]. For example, "some subtleties of control are contained within the robot–world interface and

are easily overlooked" [Nelson et al., 2009]. It is also well-known that the transition from simulation to the real robot is challenging which is refereed to by the term 'reality gap' problem [Jakobi et al., 1995, Koos et al., 2013, Bongard, 2013]. Both, pushing evolutionary robotics to higher degrees of autonomy, for example by online, onboard evolution [Eiben et al., 2010, Haasdijk et al., 2014, Haasdijk and Bredeche, 2013, Stradner et al., 2012], and getting closer to solving real-world problems, require to cover the robotics-related challenges of evolutionary robotics completely. Complex interactions between hardware features of the robot and the evolutionary dynamics should be expected. Especially dynamic robot behaviors that rely heavily on physical features, such as friction and inertia, are difficult to simulate and inherently complex [Koos et al., 2013]. However, even tasks that are easily simulated can be problematic when hardware protection is an issue as discussed in the following.

The transition from simulation to the real robot creates problems that go beyond what is summarized as reality gap problem. Not only the optimal controllers differ between simulation and reality, also elements of the evolutionary algorithm might require adaptations. It is crucial to protect the real robot from damages to allow for a rapid and cost-effective synthesis of robot controllers (collision avoidance has been investigated in evolutionary robotics from the very beginning [Floreano and Mondada, 1994]). For example, the robot might fall over, get stuck, or might be exposed to high accelerations due to collisions.

One option is to limit the robot's accelerations to decrease the impact of collisions. However, that way we deliberately exclude potentially efficient solutions to the problem. Another option is to implement a hardware protection layer that is interlaced between the application layer (here the evolutionary algorithm and the currently evaluated controller) and the hardware layer [Levi and Kernbach, 2010]. The hardware protection layer constantly monitors actuator control values and the sensor input, possibly supported by a world model, while trying to classify whether a certain control value is too risky, that is, detecting critical situations. Accordingly the robot would be stopped in dangerous situations. Such an approach might be quite complex and requires a high reliability of sensors and good sensor coverage which might be difficult to achieve especially for simple robots. The sensor configuration of the robot might not allow to prevent collisions at all times. Hence, a more efficient approach could be to solve the problem within the framework of evolutionary computation and adapt the fitness function such that controllers are favored that generally tend to prevent critical situations. This way hardware protection is easier to achieve because most of the controllers in the population at later generations will generally tend to stay away from walls. Hence, we face lower requirements for sensor reliability and sensor coverage because the robot would generally spent less time in dangerous situations. For some tasks, however, the task might actually require the robot to put itself intro critical situations. An example is parking a car, which is a simple but suitable task for our following study (a simulation study of parallel parking using evolutionary algorithms was reported in [Ronchetti and Lanzarini, 2011]). As part of the task the robot has to get close to walls (obstacles) and hence has to risk collisions. When evolving the

controllers on the robot directly, the robot's hardware needs to be protected. This can be done by penalizing collisions or close approaches to walls explicitly in the fitness function. However, then we impose additional constraints which might increase the difficulty to evolve efficient behaviors. More conservative behaviors might be favored (i.e., collision risk minimized) which might be more difficult to evolve. Therefore, we face a tradeoff between preventing damage to the robot and achieving an efficient evolutionary optimization process.

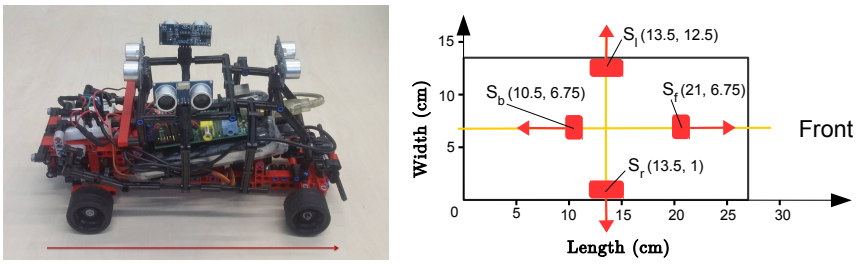
This hardware-protection-vs.-efficiency tradeoff is a general challenge of evolutionary robotics with far-reaching impact. As we seek to increase the autonomy of evolutionary robotic systems, it is getting more important that the robot stays fully functional (i.e., no damages and no deadlock situations) at all times without human interaction. In particular, it is a challenge to the new concept of embodied online onboard evolution [Eiben et al., 2010, Haasdijk et al., 2014, Haasdijk and Bredeche, 2013, Stradner et al., 2012] which follows the idea of a fully autonomous evolutionary process on the robot hardware. In the following, we investigate the above mentioned tradeoff in simulation and report robot experiments of onboard evolution as showcase. We present the robot and the robot arena that were used in the robot experiments as well as the simulation that was used for preliminary tests. In Sec. 3 we give implementation details of the evolutionary algorithm and the designed fitness function, followed by a section reporting the results.

2 Robot and robot arena

In this work, we report robot experiments of embodied onboard evolution in a parallel parking scenario. Our robot called 'LegoBot' (see Fig. 1(a)) was built using the following components: Lego^{TM} bricks, which were used to build the car chassis and allow for a high degree of customization; a DC motor, which drives the back wheels; a servo motor, for the front wheel steering; four HC-SR04 ultrasonic ranging modules (S_r, S_l, S_f and S_b), each sensor is placed on one side of the robot, see Fig. 1(b), they provide distance readings of up to four meters with a measuring angle of 15°; and a Raspberry Pi Linux board which collects the data from the ultrasonic sensors and instructs the motors according to the Artificial Neural Network (ANN) controller.

In addition, a rectangular arena of dimensions 120cm×80cm was built. Two rectangular objects of dimensions 40cm×23.5cm and 20cm×23.5cm are placed in the arena to represent a parking slot. The parallel parking slot length equals twice the length of the robot as seen in Fig. 2(a). By replacing the 20cm×23.5cm rectangular object on the right by another object of dimensions 35cm×23.5cm, we obtain a narrower parking slot which equals about one and a half the length of the robot[1] as shown in Fig. 2(b). Note that the LegoBot has limited agility

[1]For example in North America, parallel parking slots are standardized to a width of about 2.76m and a length of about 6.1m. While the average dimensions of a mid-size car are 4.1m in length and 1.85m in width. Approximately, a parallel parking slot length and width are equal to one and a half of an average mid-size car.



(a) (b)

Figure 1: (a) our car-like robot 'LegoBot' (arrowhead indicates front); (b) its dimensions and sensor setting (left sensor S_l, right sensor S_r, front sensor S_f, and back sensor S_b).

compared to typical cars, because the maximum steering angle of the LegoBot is 31° only. In the following, we use both available parking slot lengths to define parking tasks of different difficulty.

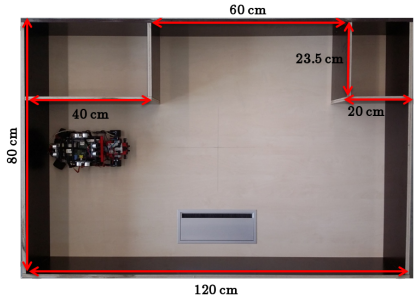
In simulation, we have a 120cm×80cm arena in a simulated environment using Player/Stage[2] [Vaughan, 2008] as shown in Fig. 3. The parking slot has dimensions 60cm×23.5cm. The actual simulated arena and parking slot dimensions match those of the real world parking slot mentioned above, see Fig. 2(a). The red cuboid in Fig. 3 represents a simulated LegoBot and is placed in the arena in a ready to reverse position. The simulated robot settings were configured to match the properties of the robot in reality as follows: simulated robot velocity 20 cm/s, front wheel steering velocity 137.7°/s, steering angles in seven discrete positions: -31°, -21°, -11°, 0°, 11°, 21° and 31°.

3 NEAT and fitness functions

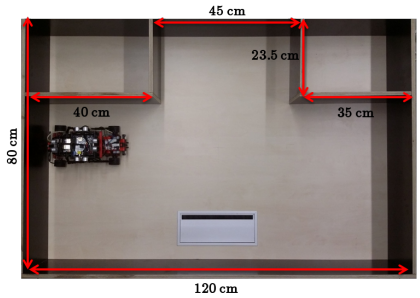
Our implementation is based on MultiNEAT[3] [Chervenski and Ryan] which is a portable software library implementing NEAT that uses the complexification method [Stanley and Miikkulainen, 2002]. MultiNEAT includes many external libraries which allow the user to connect the NEAT functions and methods with other algorithms easily. In this work, the MultiNEAT software library is used in the simulations and it is also run on the robot in our experiments, in order to evolve ANN controllers which are able to perform the parallel parking task efficiently in both simulation and real world environments. NEAT relies on many parameters to define the evolution process. Table 1 specifies the NEAT parameters used in our experiments. These parameters were fine-tuned to the parallel parking task. We evolve ANN with four input neurons, a variable

[2]Player/Stage is a popular open source software for research in robotics and sensor systems, see http://playerstage.sourceforge.net.

[3]It was developed by Peter Chervenski and Shane Ryan around 2008 at NEAT Sciences Ltd.



(a) Parallel parking slot of dimensions 60cm×23.5cm (long slot).



(b) Parallel parking slot of dimensions 45cm×23.5cm (short slot).

Figure 2: Real world parallel parking arena scenarios.

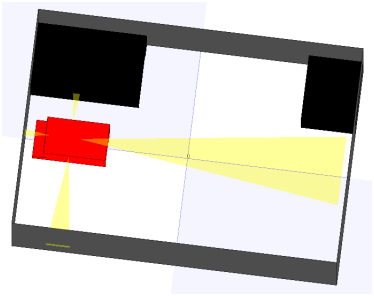


Figure 3: Simulated robot and arena.

Parameter	Value	Parameter	Value
PopulationSize	30	*CrossoverRate*	0.5
DynamicCompatibility	True	*MutateWeightsProb*	0.9
CompatTreshold	2.0	*WeightMutationMaxPower*	5.0
YoungAgeTreshold	15	*YoungAgeFitnessBoost*	1.0
OverallMutationRate	0.5	*WeightReplacementMaxPower*	5.0
OldAgeTreshold	35	*MutateWeightsSevereProb*	0.5
MinSpecies	5	*WeightMutationRate*	0.75
MaxSpecies	25	*MaxWeight*	20
SurvivalRate	0.6	*MutateAddNeuronProb*	0.04
RouletteWheelSelection	False	*MutateAddLinkProb*	0.03
RecurrentProb	0	*MutateRemoveLinkProb*	0.03

Table 1: Used NEAT parameters.

number of neurons on the hidden layer based on NEAT, two output neurons (*DCMotorControl* and *ServoMotorControl*), and an unsigned sigmoid activation function (see Fig. 4). This simple controller approach was chosen because our focus is on the hardware-protection-vs.-efficiency tradeoff.

The sensor input is provided by the four ultrasonic sensors mounted on the LegoBot. The output *ServoMotorControl* controls the angles of the front steering wheels. The primary output of the network is limited to $[0, 1]$ and then mapped to seven intervals. Each interval is associated with a steering angle (as mentioned above, seven possible steering positions between -31° and +31°). The output *DCMotorControl* controls the movement state of the back wheels (i.e., drive axle). The primary output is also limited to $[0, 1]$ but then mapped to three intervals. Each interval is associated with a movement state: forward, backward, and stopped.

We have put an effort into the design of the fitness function to minimize the number of required evaluations and allow for an evolutionary approach onboard the robot. The evaluation of a controller begins with placing the robot at the starting position manually (human interaction). Then the currently evaluated controller steers the robot for a period of time (see Table 2 for the used parameters). During the evaluation the robot acquires sensor data to calculate an accumulated fitness autonomously. The LegoBot requires 0.06 seconds to receive the readings from all four sensors (based on the technical specification of the used ultrasonic sensors). Consequently, the time frame is divided into a number of discrete steps with an interval of 0.06 seconds each.

We consider the following behavioral features for the design of the fitness function: reaching the parking slot, positioning the robot at the center of the parking slot (balancing), avoiding to hit the walls, continuous movement, and the required time. The first aspect (reaching the parking slot) is the main objective that considers how close the robot is positioned to the parking slot. The second aspect (balancing) penalizes controllers that position the robot not cor-

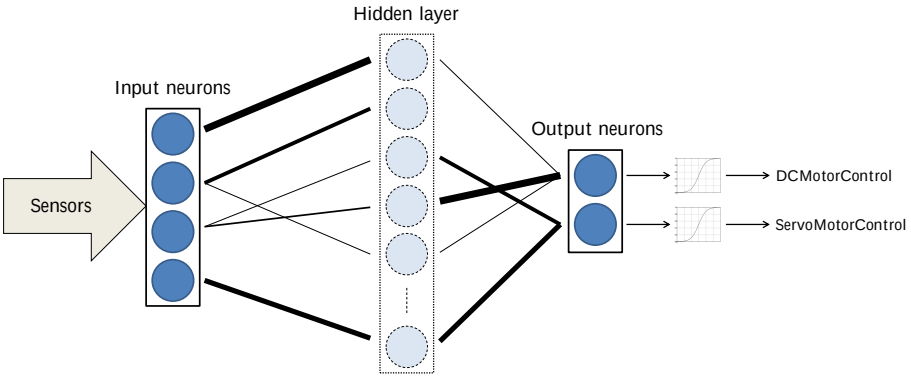


Figure 4: An example of an ANN with four input neurons and two output neurons. The connection weights are represented by the thickness of the connection lines.

Parameter	simulation long slot	robot long slot	robot short slot	robot forward parking
crashing penalty P_{crash}	varied	0.4	0.4	1
stopping penalty P_{stop}	0.2	0.2	0.2	0.2
alignment penalty threshold θ_{align}	0.07	0.07	0.07	0.07
alignment penalty weight w_{align}	0.1	0.15	0.15	0.1
balancing penalty threshold θ_{bal}	0.6	0.6	0.45	0.45
balancing penalty weight w_{bal}	0.5	0.5	0.5	1.0
sensor weight front w_{f}	2.0	2.0	2.0	2.0
sensor weight right w_{r}	2.0	2.0	2.0	2.0
sensor weight back w_{b}	2.0	2.0	4.0	4.0
transposition penalty P_{trans}	0	0	0.1	0.15
transposition threshold θ_{trans}	0	0	1	5
sensor error threshold θ_{sensor}	–	–	0.4	0.4
max. error E_{max}	6.0	6.0	6.0	7.0
evaluation time steps T	75	75	58	66
total evaluation time	4.5s	4.5s	3.5s	4.0s

Table 2: Used fitness function parameters for each experiment: simulation long slot (simulation experiment with 60cm parking slot), robot long slot (robot experiment with 60cm parking slot), robot short slot (robot experiment with 45cm parking slot), and robot forward parking short slot (robot experiment with 45cm parking slot and robot enters parking slot forwards).

rectly at the center of the parking slot. The third aspect is to avoid hitting any walls during the parking process. The fourth aspect is to keep the robot moving continuously during the simulation run. Therefore, occasional temporary stops

during the evaluation are penalized even if the respective controller parks the robot successfully in the end. The fifth aspect is minimizing the number of transpositions (i.e., transposition maneuvers in terms of changing the robot's direction) performed by the robot during the parking process, therefore, unnecessary transpositions are penalized. The robot is considered to have performed a transposition, if it changes its steering angle of the front wheels or the direction of rotation of its back wheels. The last aspect is to enforce a reasonable time frame, that is, the controller should be able to perform an efficient and successful parking process within short time. We define the fitness function F to evaluate all these behavioral aspects via the deviation, error E, from a theoretical best behavior

$$E = \left(\sum_{t=1}^{T} P_{\text{align}}(t) + P_{\text{stop}}(t) \right) + w_f S_f(T) + w_r S_r(T) + w_b S_b(T) + P_{\text{bal}} + P_{\text{crash}}, \quad (1)$$

$$F = (E_{\max} - E)^2, \quad (2)$$

for the number of evaluation time steps T. Features P_{align} and P_{stop} are accumulated over time, whereas P_{align} depends on the sensor reading of the current time step t. In contrast, the three sensor readings $S(T)$ only take the sensor reading of the last time step T. In total, eq. 1 introduces five different types of penalties. At time step t, if the distance between the robot and the obstacles on the right side of the robot exceeds a certain limit (θ_{align}), the controller receives the alignment penalty $P_{\text{align}}(t) = S_r(t) w_{\text{align}}$. The idea is to minimize the distance between the robot and the wall on its right side, which favors the behavior of staying parallel and close to the wall. The controller receives a stopping penalty $P_{\text{stop}}(t)$ at time steps t for which the robot stops its motor (otherwise $P_{\text{stop}}(t) = 0$). If the robot hits any obstacle at any time step i, the controller is considered to be unsuccessful. Therefore, it receives a crashing penalty P_{crash} and the evaluation is immediately stopped. Hence, the controller cannot collect additional, accumulated penalties: $\forall t > i, P_{\text{align}}(t) + P_{\text{stop}}(t) = 0$. The robot is stopped at the end of each evaluation. Then the sum of the front and back sensor is tested $S_f + S_b < \theta_{\text{bal}}$, which indicates whether the robot is placed correctly within the parking slot. Therefore, the balancing penalty $P_{\text{bal}} = |S_b(T) - 0.04 - S_f(T)| w_{\text{bal}}$ is added to the total error with the aim of balancing the robot in the parking slot. The robot is perfectly balanced for $|S_b(T) - S_f(T)| = 0.04$ because the sensors are positioned asymmetrically on the robot, see Fig. 1(b). Moreover, the final weighted sensor readings $w_f S_f(T) + w_r S_r(T) + w_b S_b(T)$ are added to the total error value E. To specify the optimization problem as a maximization problem, we define fitness F as the squared difference between a theoretical maximal error $E_{\max}$ and the calculated error value E. Table 2 specifies the fitness function parameters that were used in each experiment. In the following we report results of four experiments. For the two last experiments (robot short slot and robot forward parking, see Sec. 4), the transposition penalty is introduced.

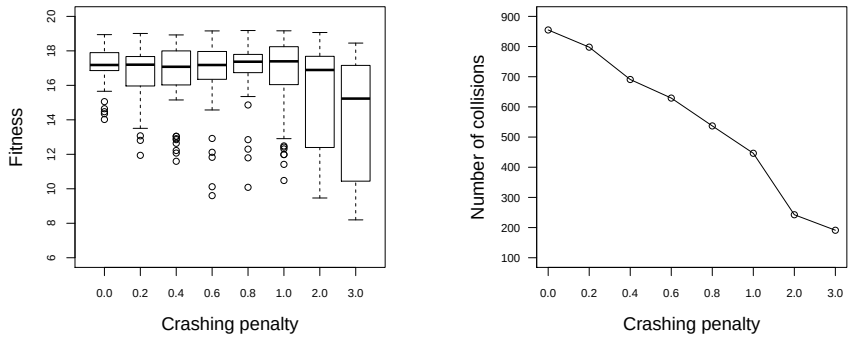
If the total number of transpositions m (i.e., moving back and forth) exceeds a certain limit $m > \theta_{\text{trans}}$ the controller receives a transposition penalty (P_{trans}) at each additional transposition.

Finally, we have to take care of sensor errors. The robot is evaluating its performance autonomously onboard, which means that the fitness function is sensitive to sensor errors. The ultrasonic sensors work best when facing walls perpendicular to the sensors. Highly inaccurate readings (typically largely overestimated) can occur when the sensors face a smooth surfaces at flat angles because then the reflection of the sound waves is disadvantageous. In our setup that is a relatively frequent situation and during the evaluation process some distance measurements might be wrong. To prevent the assignment of wrong fitness due to sensor errors, we implement the following simple error correction method. A current sensor reading is dropped and replaced by its predecessor if the difference between them exceeds the threshold θ_{sensor}.

4 Experiments

We report results of four experiments: simulation long slot (simulation experiment with 60cm parking slot, forward drive turned off), robot long slot (robot experiment with 60cm parking slot, forward drive turned off), robot short slot (robot experiment with 45cm parking slot, forward drive turned on), and robot short slot forward parking (robot experiment with 45cm parking slot, forward drive turned on, and robot enters parking slot forwards).

Our first result is from simulations (simulation long slot). With this experiment we investigate the above mentioned tradeoff between hardware protection and optimization success. The crashing penalty P_{crash} is the feature in the fitness function that influences the tendency to approach walls closely in the evolved behaviors. A high crashing penalty P_{crash} favors the emergence of conservative behaviors that stay far away from walls if possible. A low crashing penalty favors the emergence of risk-taking behaviors that might approach walls closely. Our hypothesis is that setting the crashing penalty to high values is effective in minimizing the number of collisions but it also decreases the efficiency of the optimization process. In contrast, setting the crashing penalty to low values has a minor effect on the optimization success but is also ineffective in limiting the number of collisions. Hence, we face a tradeoff which needs to be balanced between preserving the robot's intactness while enabling an efficient evolutionary search. We test eight different crashing penalties $P_{\text{crash}} \in \{0, 0.2, 0.4, 0.6, 0.8, 1, 2, 3\}$. For each penalty value, 50 evolutionary runs of 60 generations each with a population size of 30 were done. In total that are 400 evolutionary runs, which required about 37.5 days of computation time. The results are shown in Fig. 5. There is a clear trend of decreasing fitness with increasing crashing penalty in Fig. 5(a) which shows the fitness of the best controllers of the last generation from each evolutionary run. The total number of collisions per evolutionary run averaged over 50 runs and depending on the crashing penalty is shown in Fig. 5(b). Clearly, an increasing crashing



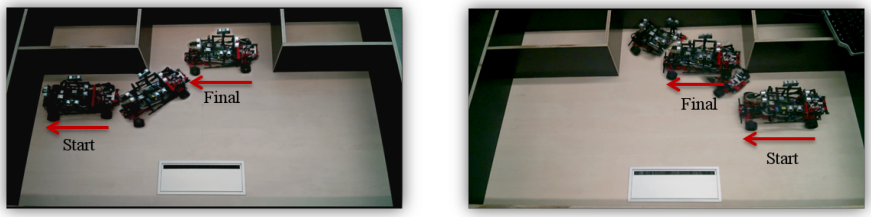
(a) Fitness of best evolved controllers for different crashing penalties.

(b) Average number of collisions for different crashing penalties.

Figure 5: Simulation results for the analysis of the tradeoff between hardware protection and optimization success (varied crashing penalty P_{crash}, simulation long slot).

penalty effectively limits the number of collisions. However, crashing penalties of high values ($P_{\text{crash}} > 1$) decrease the fitness of the best evolved controllers, that is, the evolved behaviors are of reduced quality. Crashing penalties of low values ($P_{\text{crash}} \leq 1$) allow to evolve high quality behaviors while decreasing the number of collisions linearly with increasing penalty. However, a number of over 400 collisions is possibly still unsatisfying (depending on the robustness of the robot hardware). This finding supports our hypothesis. In the following robot experiments, the crashing penalty P_{crash} was set to intermediate values (see Table 2).

Three robot experiments with different configurations were performed as onboard evolution on the robot to show the effectivity of our approach. This required running the evolutionary process directly on the Linux board on the LegoBot (based on MultiNEAT, robot determines fitness autonomously based on sensor measurements, human interaction for replacing the robot between evaluations). For each of the first two experiments (robot long slot and robot short slot), 10 independent evolutionary runs were done, each with five generations and a population size of 30 (i.e., in total 1500 evaluations). Each evolutionary run required about 19 minutes (for a quick experimenter replacing the robot to the starting position with an average of only three seconds). In total, about three hours were required to accomplish each of these two experiments. The robot was initially placed at the left side of the arena, in a ready to reverse starting position. The setup for the robot-short-slot experiment is seen in Fig. 6(a). In the third robot experiment (robot short slot forward parking), the task difficulty is increased further. The robot's initial position is changed



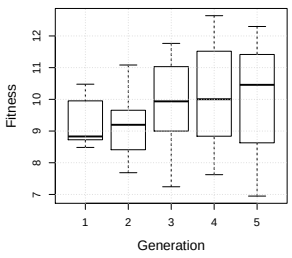
(a) robot experiment, short slot. (b) robot, short slot, forward parking.

Figure 6: Illustration of the evolved behaviors for the onboard evolution robot experiments with short parking slot, (a) backward and (b) forward parking (arrowhead indicates robot's front).

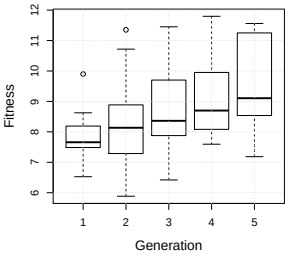
to the right side of the arena as seen in Fig. 6(b). Hence, the robot has to use the forward drive to get into the parking slot. In this setting, the parking slot is too short to allow for forward parallel parking with only one forward movement. The robot needs to switch to reverse and drive backwards while steering to reach a balanced final position. As a result, the parallel parking task of this last experiment is the most challenging. Again 10 evolutionary runs were done but with 10 generations each. Each evolutionary run required about 35 minutes which totals to about six hours for all runs. A video is available online[4] that shows a complete evolutionary run.

The best fitness over generations for the robot long slot experiment is shown in Fig. 7(a). The evolutionary approach is effective as indicated by an increase of fitness. However, the optimization process has not yet converged as indicated by a missing saturation effect. Given the high cost of these experiments we allocated most of our resources to the last and most complex experiment. The best fitness over generations for the robot short slot experiment is shown in Fig. 7(b). Again, the evolutionary approach is effective as indicated by an increase of fitness but the experiment was stopped early to save resources. The parking behavior of the evolved best controllers is similar for both of these experiments. For the robot short slot experiment, it is illustrated in Fig. 6(a). From the starting position, the robot starts to reverse into the parking slot with an appropriate steering angle. As soon as the robot starts entering the parking slot, the steering angle is reversed to the other direction ending up in the final position. A video is available online[5] that shows the whole parking process. The best fitness over generations for the third robot experiment, the robot short slot forward parking experiment, is shown in Figure 7(c). An increase in fitness is clearly seen as well as a saturation effect during the last three generations. The best evolved parking behavior is shown in Fig. 6(b). From the starting position the robot uses the forward drive with an appropriate steering angle to place

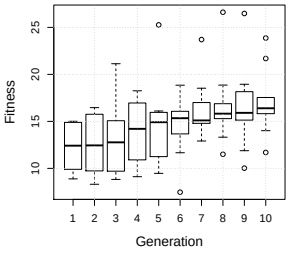
[4] http://youtu.be/L4mnuVJmepk
[5] http://youtu.be/9pz_rezn_3Q



(a) robot long slot



(b) robot short slot



(c) robot short slot forward

Figure 7: Onboard evolution robot experiments, fitness of the best controllers per generation of the 10 evolutionary runs for (a) long parking slot, backward; (b) short parking slot, backward; (c) short parking slot, forward parking.

itself inside the parking slot facing the front wall. Next, it changes its steering angle to face the corner of the parking slot, giving itself enough room to steer back and to place itself in the final position (video online[6]).

Finally, we determine the number of collisions per evolutionary run averaged over all runs in each robot experiment. For the robot long slot experiment we get in average 81 collisions, for robot short slot an average of 88 collisions, and for robot short slot forward parking we have 144 collisions in average. For all three experiments about half of the evaluations were ended with a collision and the robot's motor was turned off for protection. This ratio is considered to be good because we influence the robot behavior only via the fitness function. The effect of the crashing penalty is established slowly over several generations starting from a population of random controllers that typically generate many collisions.

5 Conclusion

We have reported the concept of the tradeoff between hardware protection and the optimization success in onboard evolution. In robot experiments there is a requirement to minimize the number of events that expose the robot to high accelerations. This is particularly relevant for addressing the autonomy challenge of evolutionary robotics, that is, we want to create autonomous evolutionary processes that form highly adaptive systems [Eiben et al., 2010, Haasdijk et al., 2014, Haasdijk and Bredeche, 2013, Stradner et al., 2012]. We have to prevent damage to the robot but at the same time we do not want to overly constrain explorations of the search space by the evolutionary algorithm. Awareness of this tradeoff helps to define an appropriate fitness function. In our simulation experiments we have shown the impact of different crashing penalties and in our onboard evolution robot experiments we have shown the effectivity of our approach.

In future work we will compare the approach of introducing changes to the fitness function to minimize the number of collisions with the implementation of a hardware protection layer. Furthermore, we plan to investigate the impact of this study on the concept of embodied online onboard evolution.

References

Josh C. Bongard. Evolutionary robotics. *Communications of the ACM*, 56(8):74–83, 2013.

Peter Chervenski and Shane Ryan. MultiNEAT, project website;. URL http://www.multineat.com/. http://www.multineat.com/.

Ágoston Endre Eiben, Evert Haasdijk, and Nicolas Bredeche. Embodied, on-line, on-board evolution for autonomous robotics. In Paul Levi and Serge Kernbach, editors, *Symbiotic Multi-Robot Organisms: Reliability, Adaptability, Evolution*, volume 7 of *Cognitive Systems Monographs*, pages 362–384. Springer, 2010.

[6] http://youtu.be/n-KSrIwp87k

Dario Floreano and Francesco Mondada. Automatic creation of an autonomous agent: genetic evolution of a neural-network driven robot. In *Proceedings of the 3rd Int. Conf. on Simulation of Adaptive Behavior (SAB'94)*, pages 421–430. MIT Press, 1994.

Evert Haasdijk and Nicolas Bredeche. Controlling task distribution in MONEE. In Pietro Liò, Orazio Miglino, Giuseppe Nicosia, Stefano Nolfi, and Mario Pavone, editors, *Advances In Artificial Life (ECAL 2013)*, pages 671–678, 2013.

Evert Haasdijk, Nicolas Bredeche, and Ágoston Endre Eiben. Combining environment-driven adaptation and task-driven optimisation in evolutionary robotics. *PLoS ONE*, 9 (6):e98466, 06 2014. doi: 10.1371/journal.pone.0098466. URL http://dx.doi.org/10.1371%2Fjournal.pone.0098466.

Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Proceedings of the Third European Conference on Advances in Artificial Life*, volume 929 of *LNCS*, pages 704–720. Springer, 1995.

Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. The transferability approach: Crossing the reality gap in evolutionary robotics. *Evolutionary Computation, IEEE Transactions on*, 17(1):122–145, 2013.

Paul Levi and Serge Kernbach, editors. *Symbiotic Multi-Robot Organisms: Reliability, Adaptability, Evolution*. Springer, February 2010.

Andrew L. Nelson, Gregory J. Barlow, and Lefteris Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57:345–370, 2009.

Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press, 2000.

Franco Ronchetti and Laura Cristina Lanzarini. Automatic vehicle parking using an evolution-obtained neural controller. In *Presentado en el XII Workshop Agentes y Sistemas Inteligentes (WASI)*, pages 71–80, 2011.

Kenneth Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Jürgen Stradner, Heiko Hamann, Payam Zahadat, Thomas Schmickl, and Karl Crailsheim. On-line, on-board evolution of reaction-diffusion control for self-adaptation. In Christoph Adami, David M. Bryson, Charles Ofria, and Robert T. Pennock, editors, *Alife XIII*, pages 597–598. MIT Press, 2012.

Richard Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2(2-4): 189–208, 2008.