Revolve: A Versatile Simulator for Online Robot Evolution

Hupkes, Elte; Jelisavcic, Milan; Eiben, A. E.

published in
Applications of Evolutionary Computation
2018

DOI (link to publisher)
10.1007/978-3-319-77538-8_46

document version
Publisher's PDF, also known as Version of record

document license
Article 25fa Dutch Copyright Act

citation for published version (APA)
Hupkes, E., Jelisavcic, M., & Eiben, A. E. (2018). Revolve: A Versatile Simulator for Online Robot Evolution. In K. Sim, & P. Kaufmann (Eds.), Applications of Evolutionary Computation: 21st International Conference, EvoApplications 2018, Parma, Italy, April 4–6, 2018, Proceedings (pp. 687-702). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 10784 LNCS). Springer. https://doi.org/10.1007/978-3-319-77538-8_46

Revolve: A Versatile Simulator for Online Robot Evolution

Elte Hupkes[1,2], Milan Jelisavcic[1](✉), and A. E. Eiben[1]

[1] Vrije Universiteit Amsterdam,
De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands
{m.j.jelisavcic,a.e.eiben}@vu.nl
[2] University of Amsterdam,
Science Park 904, 1098 XH Amsterdam, The Netherlands

Abstract. Developing robotic systems that can evolve in real-time and real-space is a long term objective with technological as well as algorithmic milestones on the road. Technological prerequisites include advanced 3D-printing, automated assembly, and robust sensors and actuators. The necessary evolutionary mechanisms need not wait for these, they can be developed and investigated in simulations. In this paper, we present a system to simulate *online* evolution of *constructible* robots, where (1) the population members (robots) concurrently exist and evolve their morphologies and controllers, (2) all robots can be physically constructed. Experiments with this simulator provide us with insights into differences of using online and offline evolutionary setups.

Keywords: Evolutionary algorithms · Reality gap · Online learning
Offline learning · Modular robots

1 Introduction

The motivation for this paper comes from the vision of the Evolution of Things described by Eiben and Smith [1]. In particular, the interest is in physical robots that "can evolve in real time and real space" [2]. The ultimate system of our interest consists of robots with evolvable bodies and brains that 'live and work' concurrently in the same physical environment. However, the current technology lacks essential components to this end, in particular, the mechanisms that enable that robots to reproduce, i.e., 'have children'. This technology is being developed, but in the meanwhile, simulators can be of great value as they allow us to study various system setups and generate scientific insights as well as know-how regarding the working of physically evolving robotic systems.

Looking into existing work on evolvable morphologies we can note a large variety of approaches. Yet, there are two important limitations shared by most of them. First, evolution is executed in an offline fashion, that is, through a (centralised) evolutionary algorithm that runs the standard EA loop and only calls the simulator to establish the fitness of a new genotype. Such fitness evaluations typically happen in isolation: the phenotype, i.e., the robot, that belongs to the given genotype is placed and evaluated in an environment without other robots present. This is in contrast to our vision with genuinely embodied online evolution, where all population members concurrently operate in the same environment. An important question is whether the differences between online and offline evolution are purely procedural or do the two types of systems exhibit different behaviour. To answer this question experimentally we would need a simulator that can be configured for both types of evolution keeping all other system properties identical.

Second, the robots in most of the existing simulations are not constructible. Certainly, this is not limiting for fundamental studies but the long-term goal of constructing a system of physical robots that can evolve in real time and real space implies a need for a simulator with a hi-fidelity model of real robots with an evolvable morphology.

In summary, current systems and the investigations performed with them suffer from either or both of these limitations and are less realistic that we would prefer. To this end, we present a new simulator with a unique combination of features that supports experimental research that cannot be done with existing systems. In particular,

1. All robots are physically constructible by assembling off-the-shelf components (e.g., servo motors, LED lights) and 3D-printed body parts.
2. Robots have evolvable morphologies and controllers. Both the physical and the mental makeup are encoded by genotypes that can be mutated and recombined. The phenotype space and the corresponding genetic code is based on RoboGen [3].
3. Evolution is carried out in an online, embodied fashion. Robots populate an environment simultaneously and selection and reproduction are determined by the (inter)actions of the robots and the environment. Thus, evolution is induced 'from inside', robots are not just isolated candidate solutions in some traditional genetic algorithm.

An important feature of this simulator is that it can be used in a traditional offline mode as well. Thus, it allows us to answer the research question: what are the main differences between online and offline evolution and how do these affect the dynamics of the evolving robot population? For this purpose, we specify three scenarios, one with offline evolution, one with online evolution, and a combined form and compare the emerging dynamics.

The rest of this paper is organised as follows. In Sect. 2 we briefly review the most relevant existing works. The details of the Revolve simulator are presented in Sect. 3. Section 4 describes the robots, followed by the outlines of the evolutionary system in Sect. 5 and the three scenarios in Sect. 6. The experimental results are presented in Sect. 7.

2 Related Work

Current technology limitations confine studies on evolving morphologies to software simulations. Although works based purely on simulation fall outside the present overview, the classic experiments of Sims deserve to be mentioned [4]. This system works through a traditional EA that evaluates virtual creatures one by one in a simulator, assessed by different locomotion skills, such as walking, hopping, swimming, and for the task of fighting over a block in between two organisms.[1] The virtual organisms are modular, consisting of blocks of different sizes which are connected through actuators driven by neural network controllers. A couple of other papers follow a similar approach: they evolve artificial organisms and their control structures in simulation using evolutionary algorithms [5–7]. These creatures are not very realistic, they cannot be directly manufactured in real hardware. The evolutionary systems are not natural either (centralised, offline), but the papers demonstrate the concept of morphology evolution.

The work of Auerbach and Bongard is especially interesting because it casts morphological evolution in a broader context of the body-mind-environment trichotomy [5]. In particular, they study the relation between morphological, neural, and environmental complexity in an evolutionary system. They use a simple and a complex environment and evolve robots that comprise triangular meshes and are driven by neural controllers. Comparing the evolution of morphological complexity in different environments they find that “When no cost was placed on morphological complexity, no significant difference in morphological complexities between the two sets of robots evolved. However, when the robots were evolved in both environments again, and a cost was placed on complexity, robots in the simple environment were simpler than the robots evolved in the complex environment” (quote from [8]).

An interesting cluster of papers gets closer to reality by using simulations for evolving morphologies and constructing the end result. Lipson and Pollak used an EA and a simulator to evolve robotic organisms that consisted of bars and actuators (but no sensors) driven by a neural net for the task of locomoting over an infinite horizontal plane [9].

Perhaps the most interesting work in this cluster of papers is the recently developed RoboGen system [3]. RoboGen works with modular robots encoded by artificial genomes that specify the morphology and the controller of a robot, a simulator that can simulate the behaviour of one single robot in a given environment, and a classic evolutionary algorithm that calls the simulator for each fitness evaluation. The system is used to evolve robots in simulation and the evolved robots can be easily constructed by 3D-printing and manually assembling their components. RoboGen was not meant and is not being used for physical robot evolution, but it could be the starting point for such a system after a number of extensions (e.g., crossover for both morphologies and controllers).

2.1 Reality Gap

A notorious problem in evolutionary robotics is the *reality gap* first mentioned by Jakobi et al. [10], referring to the behavioural differences between simulated sys-

[1] http://www.karlsims.com/evolved-virtual-creatures.html.

tems and their real physical counterparts. While a simplification, rounding and numerical instability lead to differences whenever computer models are involved, this effect is amplified in evolutionary systems. The reason for this is that evolution, as previously noted, will often solve its set challenges with unexpected solutions. While this is generally a favourable property, it also means that the process may eventually 'exploit' whatever modelling errors or instabilities are present, arriving at a solution that is valid only in the context of the simulation. Aside from careful calibration of the behaviour of the simulator, the simplest and therefore most commonly used approach to counter this problem is to add noise to a virtual robot's sensors and actuators. While straightforward, this greatly increases the number of sensory representations of otherwise similar or identical states, slowing down the evolutionary process. More complicated approaches may involve alternating fitness evaluations between simulation and reality [11–14], but this is infeasible when simulating entire artificial ecosystems. Crossing the reality gap is by no means a solved problem and as always one should be cautious drawing definitive conclusions from a model.

3 The Simulator

The main design decision when conceiving Revolve was a choice between either (a) building on top of a dynamics engine directly, (b) modifying the code of an existing research project or (c) using a simulation platform. Out of these (b) and (c) are more viable options because they take away a large part of the bootstrapping process, and in addition, ensure improvements and fixes to the underlying infrastructure regardless of the development of the toolkit. The important factor for making the final decision was the ability of any of the possible options to integrate C++ libraries into its environment. The importance of using these native libraries lays behind the idea to recreate all simulated robots in hardware with Raspberry Pi as a controller.

Investigations were performed with the NASA Tensegrity and RoboGen source codes [3,15], running benchmarks and trying to realise simple artificial ecosystems using the existing code base. There was a particular focus on RoboGen as an attractive candidate for a proof of concept, given that its robot body space is easily constructed using 3D printing, and is subject to an ongoing real-life calibration process. During the setup of simple scenarios, however, it was found that the RoboGen software suite was too much tailored to its serial, offline evolution to be conveniently re-factored to the new use case. In addition, all code would have to be written in the C++ language, which provides high performance at the expense of being verbose and sometimes tedious to develop. While the choice for the RoboGen body space as a proof of concept remained, the decision was made to build the Revolve Toolkit on top of a general purpose simulation platform instead. The decision was made based on the fact that it appeared to be much easier to develop a simulation platform for a specific need on top of a flexible system than to modify any of existing solutions.

Out of the considered simulation platforms (Webots[2], MORSE[3], V-REP[4], Gazebo[5]), only Webots was discarded beforehand because it is a commercial and closed-source platform, and constraints in Webots limited the number of individuals that could be simulated, regardless of performance. MORSE appeared to be a suitable candidate but lacked the ability for high-performance C++ integration that Gazebo and V-REP provided, as well as the lack a choice of physics engines. A comparative analysis of the last two remaining platforms was conducted [16], ruling in favour of V-REP by a slight margin. However, a much older version (2.2) of Gazebo was used than was available, even at the time the paper was written. Additionally, the methodology compares CPU usage rather than simulation work performed over time. Considering all points, the bottom line is that V-REP and Gazebo are very similar platforms in terms of features. The eventual choice for Gazebo is motivated by its non-commercial nature, its large online community and the XML format it uses to describe models, which simplifies creating dynamic robot morphologies from external applications. That being said, V-REP would likely also have been very suitable as a platform. While Revolve has been written with Gazebo in mind, large parts are simulator agnostic and could potentially be used for creating a similar platform for use with V-REP. Following paragraphs will describe system parts in details.

Revolve. [6]The **R**obot **Evolve** toolkit is a set of Python and C++ libraries created to aid in setting up simulation experiments involving robots with evolvable bodies and/or minds. It builds on top of Gazebo, complementing this simulator with a set of tools that aim to provide a convenient way to set up such experiments. Revolve's philosophy is to make the development of simulation scenarios as easy as possible while maintaining the performance required to simulate large and complex environments. In general this means that performance critical parts (e.g. robot controllers and parts relating to physics simulation) are written in the C++ language, which is highly performant but can be tedious to write, whereas less performance-focused parts (such as world management and the specification of robots) are written in the slower yet more development-friendly Python language. The bulk of the logic of a simulation setup commonly falls in the latter category, which means the experimenter will be able to implement most things quickly in a convenient language.

Gazebo. An open source, multi-platform robotic simulation package that is available free of charge. It provides both physics simulation and visualisation of rigid body robotic structures and their environments. Abstraction wrappers are provided for several well-established physics simulation engines: The Open Dynamics Engine (ODE), Bullet Physics, SimBody, the Dynamic Animation and Robotics Toolkit (DART), Having these abstractions available means that the

[2] https://www.cyberbotics.com/.
[3] https://www.openrobots.org/wiki/morse.
[4] http://www.v-rep.eu/.
[5] http://gazebosim.org/.
[6] http://www.github.com/ci-group/revolve.

same simulation can, in theory, be run using any of these physics engines by changing a single parameter - the caveat being that subtle differences between these engines often require additional parameter tuning to get a stable simulation.

In order to describe robots and environments, Gazebo uses the Simulation Description Format (SDF)[7], which allows an end user to specify anything from the texture of the terrain to the physical and visual properties of robots in an XML-based format. Because XML can be cumbersome to write for human beings, the `sdf-builder`[8] Python package was developed concurrently with Revolve to provide a thin, structured wrapper over this format that aids with positioning and alignment of geometries, and calculation of their physical properties.

What makes Gazebo particularly useful is the means by which it allows programmatic access to observing and modifying the simulation. It provides two main interfaces to do this and Revolve makes use of both:

- **A messaging API.** Gazebo comes bundled with a publisher/subscriber messaging system, in which any component can subscribe to and/or publish on so-called topics. Many aspects of the system can be controlled using these messages, which are specified in Google's *Protocol Buffers* (Protobuf) format. Because this communication happens over TCP sockets, access to this interface is quite straightforward in most programming languages.
- **The plugin infrastructure.** It is possible to load shared libraries as a plugin for several types of Gazebo components, providing programmatic access to the simulation using Gazebo's C++ API. As an example, one can specify a certain piece of compiled C++ code to be loaded with every robot that is inserted into the world.

Revolve Libraries. At the heart of Revolve lie a set of general purpose tools, which can be roughly separated into Python components and Gazebo C++ plug-in components. A certain layering is present in the provided tools, ranging from anything from closely related to the specification to more practical tools that can be used to quickly implement an actual experiment.

Revolve Angle. Alongside the modules to create a wide variety of experimental setups described in the previous sections, Revolve includes a more opinionated module called `revolve.angle`, implementing a specific subset of all possible experimental setups. Its function is twofold, in that (a) it allows for setting up any experiment matching these setup descriptions rapidly, and (b) it serves as an example of how to use Revolve. It implements the following functionality: (a) a genome including both a robot's body and brain, (b) a conversion from this genome to a usable SDF robot, (c) evolutionary operators functioning on this genome: crossover and mutation, and (d) the entire RoboGen body space is included as Revolve components, though its use is optional and other body parts may just as well be used with the genome.

[7] http://sdformat.org/.
[8] https://github.com/ci-group/sdf-builder.

Gazebo plugins. In order to actually control a robot in simulation, Gazebo has to be told what sensor values to read, what joints to control, etc. While it is possible in principle to provide most of these functionalities through the messaging API, when it comes to controlling a robot the code is closely related to the simulation, runs often and is, therefore, more apt to be considered as a high-performance aspect to be written in C++. Revolve supplies a base *robot controller* plugin to deal with this aspect of the simulation setup. When the SDF contents of a robot are produced for simulation, a reference to this plugin is included alongside information about its sensors, actuators and brain. Gazebo supports many types of sensors, all of which are accessed in a different fashion. Revolve wraps around a number of often used sensors and unifies them in a generic interface passed to the robot controller. The same holds for actuators, which control the joints of robots in the simulation. Rather than having to specify the forces that operate on these joints, Revolve allows setting either a position or velocity target which, combined with a predefined maximum torque, resembles the interface of a real-world servo motor.

In addition to the robot controller, Revolve also includes a *world controller* plugin, which should be included with each loaded simulation world. While using this plugin is not strictly necessary, it includes some convenient functionality to insert robots into the world, keep track of their position and remove them. Overall, Revolve system is a simulation toolkit designed specifically for the purpose to study embodied co-evolution, specifically described in [2]. However, having in mind that it is based on highly flexible Gazebo simulator, it can be also used for a research related to specific parts of the system, e.g.individual learning, group learning, island model evolution.

4 The Robots

The robot design in our system is based on the RoboGen framework[9]. RoboGen robots are designed to be evolvable and easily manufacturable from a pre-defined set of 3D printable components (some of which are parametrisable) coupled with off the shelf electronic elements. The following paragraphs describe the possible components in detail.

The Phenotype of Morphologies. The robots used in this work are built out of seven possible component types, which can connect to each other at specified attachment slots.

1. Each RoboGen robot contains at a minimum a **core component** that houses a battery and a microcontroller, with an onboard 6-dimensional inertial measurement unit (IMU) composed of an accelerometer and a gyroscope. This component has four attachment slots: apart from the top and bottom faces, it is possible to attach other components to every side. It is slightly larger than other components so it could contain any needed electronics if the robot would be recreated in real-space.

[9] http://robogen.org.

2. The **fixed brick** has the smaller dimensions than the core component, and it does not contain any electronics or sensors. In addition, other components can be attached to its four faces.
3. The **parametric bar joint** is a connection element with parametrised length and connection angle. It has two attachment slots, one at either end.
4. The **active hinge** is a simple hinge joint powered by a servo motor. Each active hinge adds one actuated degree of freedom to the robot: this DOF is controlled by a single value per time step, which defines a desired angular position between −45 and 45 degrees. Like the parametric bar joint, it has two attachment slots.
5. The **passive hinge** is similar to the active hinge, but as the name suggests the passive hinge is not powered by a servo but rather can move freely. It, therefore, adds one un-actuated DOF to the robot. It also has two attachment slots.
6. The **touch sensor**, which contains two binary inputs. Each input represents whether or not one half of the sensor is in contact with another object. Like the light sensor, the touch sensor has a single attachment slot.

Each of these components is defined by two-part model: a detailed mesh suitable for visualisation and 3D-printing and a set of geometric primitives that define the components' mass distribution and a contact surface. As described above, each model also defines the number and placement of possible attachment slots, as well as the **inputs** (sensors) and **outputs** (motors) contained within it. Each input i is defined as a single numerical value. If a sensor outputs more than one value (as is this case with the IMU and the touch sensor) then this results in multiple defined inputs. Similarly, each output o is defined by a single value.

The Phenotype of Controllers. Each robot is controlled by a neural network that receives inputs from the robot's sensors and provides output to the robot's actuators. In this way, there is a one-to-one correspondence between a morphology's inputs and its controller's input neurons as well as between a morphology's outputs and its controller's output neurons. The neural network can also contain hidden units.

The neurons in the hidden and output layers of a robot's neural network may have one of three activation functions. The first two–linear and sigmoid–are common neural network activations whose parameter set consists of bias and gain values. The third possible type is an oscillator neuron, whose value depends not on its input values but rather is a sinusoid depending only on the current time. The three parameters for this neuron type are the oscillator's period, phase offset and amplitude. This neuron type was added to accommodate the needs of simplifying the experiment while still producing proper locomotion patterns.

The Genotype of Morphologies. Robots are genetically encoded by a tree-based representation where each node represents one building block of the robot and edges between nodes represent physical connections between pieces. Each node contains information about the type of the component it represents, its name, orientation, possible parametric values, and its colour. The colour parameter is included to allow handily tracking of which body parts originate from which parent. Each edge also defines which of the parent node's available attachment slots the child will attach to.

Construction of a robot from this representation begins with the root node, defined to always represent the requisite core component. The robot body is then constructed by traversing the tree edges and attaching the components represented by child nodes to the current component at the specified slot positions and orientations.

The Genotype of Controllers. The tree representation describing a robot's morphology also describes its controller. Based on a unique identifier and a type of each module, input and output neurons are defined. Subsequently, neurons in a hidden layer are added and connections between layers. The detailed description of how genotypes of morphologies and controllers are recombined are described in Sect. 5.2.

5 Evolutionary System

5.1 System Architecture: The Triangle of Life

The proverbial Cycle of Life revolves around birth and so does a system of self-reproducing robots. To capture the relevant components of such a robotic life cycle we need a loop that does not run from birth to death, but from conception (being conceived) to conception (conceiving one or more children). A conceptual framework for such an ecosystem in which physical robots actually reproduce was proposed in [2].

This framework, called the Triangle of Life, represents an overall system architecture with three main components or stages. This system is generic, the only significant assumption we maintain is the genotype-phenotype dichotomy. That is, we presume that the robotic organisms as observed 'in the wild' are the phenotypes encoded by their genotypes. As part of this assumption, we postulate that reproduction takes place at the genotypic level. This means that the evolutionary operator's mutation and crossover are applied to the genotypes (to the code) and not to the phenotypes (to the robotic organisms). The first stage in the ToL is the creation of a new robotic organism in a so-called Production Center [19]. This stage starts with a new piece of genetic code that is created by mutating or recombining existing pieces of code (of the robot parents) and ends with the delivery of a new robot. The second stage takes place in a Training Center; it starts when the morphogenesis of a new robot organism is completed and ends when this organism acquires the skills necessary for living in the given world and becomes capable of conceiving offspring. The third stage in the Triangle is the period of maturity. It starts when the organism in question becomes fertile and leads to a new Triangle when this organism conceives a child, i.e., produces a new genome through recombination and/or mutation.

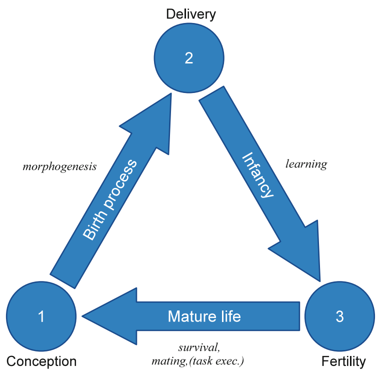


Fig. 1. Robotic life cycle captured as a triangle after [2]. The pivotal moments that span the triangle are: (1) Conception: A new genome is activated, construction of new robot starts. (2) Delivery: Construction of the new robot is completed. (3) Fertility: The robot becomes ready to conceive offspring.

Figure 1 exhibits the stages of the Triangle of Life. Similarly to the general EA scheme that does not specify the representation of candidate solutions, the ToL does not make assumptions regarding the makeup of the robots.

5.2 Evolutionary Operators

Two parent robots can produce offspring through several reproduction operators on their genotype trees. This section discusses these operations in the order in which they are applied to create a child robot c from parents a and b. In the first step, a node a_c from a is randomly chosen to be the crossover point. A random node b_c from b is chosen to replace this node, with the condition that doing so would not violate the restrictions as given in Table 1. If no such node is available, evolution fails at this point and no offspring are produced. If such a node is found, c_1 is created by duplicating a and replacing the subtree specified by a_c with the subtree b_c. With probability $p_{swap\_subtree}$, a random node s_1 is chosen from c_3. Another random node s_2 is chosen provided it has no ancestral relationship with s_1 (i.e. it is not a parent or child of this node). If no such node is available the step is again skipped, otherwise s_1 and s_2 are swapped in c_3 to produce c_4. Again in order to keep robot complexity roughly the same, a new part is added with a probability proportional to the number of parts that are expected to have been removed by subtree removal, minus the number of parts expected to have been added by subtree duplication. The new part is randomly generated by both hidden neurons, neural connections and all parameters, and attached to a random free slot on the tree to produce the final robot c. Again, this step is skipped if adding a part would violate restrictions.

In general, selection operators in an EA do not depend on the given representation and reproduction operators. Hence, the experimenter is free to use any standard mechanism without application-specific adjustments. For online evolution, this is slightly different. The main difference is the lack of synchronisation between birth and death events. In an offline evolutionary process, these are synchronised and the population size is typically kept constant. In online evolution birth and death events are triggered independently by (local) circumstances and the populations can grow or shrink depending on the actual numbers. The specific mechanism we use here is discussed in the next section.

6 Experimental Setup

The purpose of the experiments is to compare online and offline evolution in a ToL-based system of robots with evolvable morphologies. By design, we will perform pure evolutionary experiments without learning in the Infancy stage. Thus, the controllers of the robots will not change during their lifetime and the fertility test in node 3 of Fig. 1 is void: all individuals become mature/fertile 15 s after birth, which is 3 s insertion time for when the robot is dropped into the arena followed by 12 s evaluation time.

All experiments share a set of values for previously specified parameters, which are specified in Table 1 and each run is repeated 30 times.

Table 1. Parameter values shared across all experiments

Parameter	Description	Value
$\lvert\mathcal{R}\rvert_{\text{max}}$	Maximum number of nodes	30
$\lvert\mathcal{R}\rvert_{\text{min}}$	Minimum number of nodes	3
o_{max}	Maximum number of outputs	10
i_{max}	Maximum number of inputs	10
h_{max}	Maximum number of hidden neurons	10
μ_{parts}	Mean of randomly generated parts $\mathcal{N}(\mu_{\text{parts}}, \sigma^2_{parts})$	12
σ_{parts}	Standard deviation of randomly generated parts	5
$p_{\text{remove subtree}}$	Probability of removing subtree	0.05
$p_{\text{duplicate subtree}}$	Probability of duplicating subtree	0.1
$p_{\text{swap subtree}}$	Probability of swaping subtree	0.05
$p_{\text{remove hidden neuron}}$	Probability of removing hidden neuron	0.05
$p_{\text{remove neural connection}}$	Probability of removing neural connection	0.05

We consider three different experimental scenarios. The first two are offline scenarios in which individuals are evaluated in isolation and a population consists of distinguishable generations. What differentiates these two scenarios is the parent selection method: in the first scenario 15 new individuals are produced before further selection takes place, whereas in the second scenario selection happens after each newly born robot. This method of parent selection is more akin to the online scenario in which robots coexist in the environment and are continuously evaluated and selected.

The fitness function of a robot ρ is the same in all of these scenarios and reads

$$f(\rho) = v + 5s, \tag{1}$$

where v is the length of the path the robot has travelled over the last 12 s and s is the straight distance that the robot has covered the point where it was 12 s ago and the point where it is now (i.e. the length of a straight line between that point and the current point).

The following table summarises the three simulation scenarios.

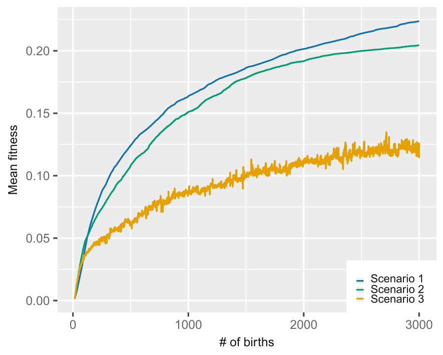
	Scenario 1	Scenario 2	Scenario 3
Scenario type	Offline		Online
Environment	Infinite flat plane		
Evaluation	One robot at a time for 12 s		All active robots simultaneously and continuously in 12 s time frame
Population size	Constant at 15 robots per generation		8 to 30 robots[a]
Selection scheme	$(15 + 15)$[b]	$(15 + 1)$[c]	A new robot every 15 s
Parent selection	4-tournament selection		
Survivor selection	Select the 15 fittest individuals		Robots with a fitness greater than a 70% of the population mean.[d]
Birth location	On the ground at the origin		Random position within a radius of 2 m from the origin
Stop criterion	After 3000 births[e]		

[a]See ‘Survivor selection’.
[b]Each generation of 15 robots produces 15 children before moving on to survivor selection.
[c]Each generation of 15 robots produces 1 child before moving on to survivor selection.
[d]A minimum of 8 robots is maintained to ensure variation and prevent extinction. If the population reaches 30 individuals without any individuals matching the death criterion, the 70% least fit robots in the population are killed regardless of their fitness to prevent a simulation stall.
[e]In scenario 1, 200 generations of 15 individuals.

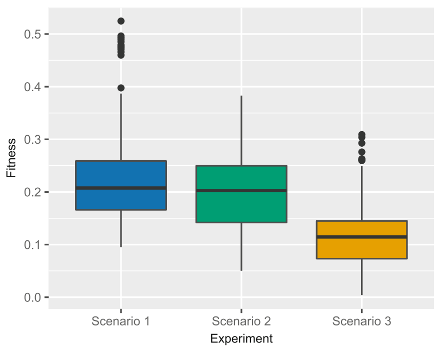
Preceding the runs with these scenarios we have conducted two baseline experiments, disabling reproduction and selection, respectively. The first one makes use of the fitness selection to determine which individuals survive while disabling reproduction, thereby showing the speed at which a population would increase its fitness if a selection is made out of an increasing random population. The second baseline experiment, on the other hand, uses completely random survivor selection, while enabling reproduction. Looking at the obtained fitness values in these experiments serves as a simple sanity check to confirm that evolution is really working, cf. Fig. 2(b).

7 Experimental Results

The fitness values of a final population are defined by the last generation of robots in the offline experiments and all alive, mature robots in the online experiment. These results are shown in Fig. 2b.



a Fitness progressions.



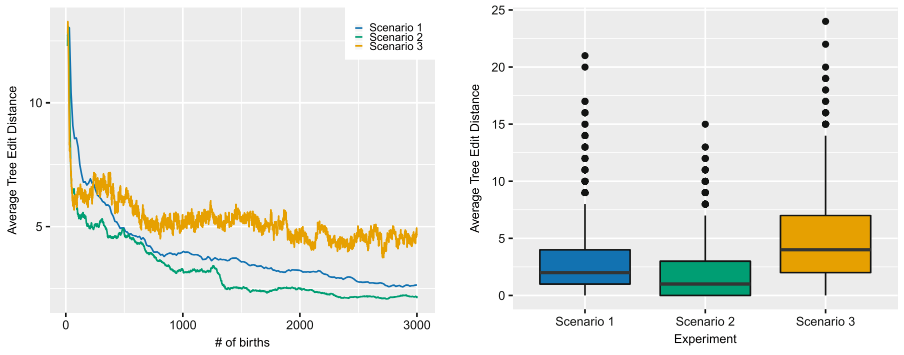
b Fitness of the final populations.

Fig. 2. Fitness values observed in each scenario aggregated over all 30 runs. The plots in Fig. 2a use the number of born individuals on a time scale because it is uniform across all scenarios. Error bars are omitted for clarity. Note that the experimental setups guarantee a monotonically increasing function for the offline experiments, whereas the non-constant fitness values in the online experiment lead to fluctuations.

The three experiments with both selection and reproduction show a significantly higher fitness than the baseline experiments, indicating that indeed we have evolution at work in the system. Between offline scenarios 1 and 2, only a slight difference in fitness is observed, with a more noticeable difference in fitness variation. Scenario 3 shows a significantly lower fitness. We hypothesise that this is caused by the fundamentally different ways of evaluating robots. In scenarios 1 and 2 each (newborn) robot is evaluated once, while in scenario 3 robots undergo continuous evaluation. This reduces the possible artefacts of getting lucky during the first and only evaluation. In other words, scenarios 1 and 2 allow for survival base on an optimistic estimate of a robots fitness, while scenario 3 works with more realistic fitness values.

Further to analysis, we are also interested in diversity to provide an adequate picture. To provide an adequate picture, a heuristic measure is applied to quantify the genetic diversity within robot populations at each time point. This measure applies a Tree Edit Distance (TED) algorithm as described by Zhang and Sasha [17] to the genetic trees of pairs of robots that are part of the same population. The algorithm is applied to the following cost rules:

- Removing a node, adding a node or changing a node to a different type has a cost of 1.
- Attaching a node to a different parent slot has a cost of 1.



a. Diversity progression over number of births. Error bars are omitted for clarity.

b. Diversity of the final populations.

Fig. 3. Genetic diversity in robot populations using Tree Edit Distance averaged over all runs.

Note that differences between neural network contributions between nodes are ignored in this measure because they are harder to quantify. The outcome of the algorithm is included for both the final populations cf. Fig. 3b and as a progression during the experiments cf. Fig. 3a. This shows an initial rapid decline of diversity in all scenarios, possibly as a result of 'bad genes' being eliminated. Diversity decline then slows down, although it decreases faster in scenario 2 than in scenario 1. This makes sense as the populations that scenario 2 uses for reproduction are very similar for each birth, which is expected to decrease variation. The same can be said about scenario 3, but the same effect cannot be observed there, meaning something is keeping diversity relatively high here.

8 Discussion and Conclusions

As shown in this paper, Revolve has proven its merit a a research tool, but there are of course several possible improvements. First, the scenarios of Sect. 6 could be extended by making the controllers evolvable and/or adding learning capabilities to the robots. This will facilitate research into Lamarckian evolution, where we already have nice initial results [21]. Adding a dedicated Production Center, as in our previous work [18], is also a logical extension. In the meanwhile, adding obstacles and different types of ground surfaces will make the system more realistic. Hereby the software and hardware-based system development will be more aligned [19,20].

Something to be wary of in this context is the 'bootstrapping problem', a term used to describe the failure of a system to evolve into interesting dynamics simply because there are no dynamics, to begin with. Robot learning could be developed and integrated into Revolve as a potential solution to this problem. This would enable robots to make more rapid and efficient use of any sensors they have, which is expected to have an impact on the influence of robot interactions and the environment in which they operate. Varying environmental properties is also an interesting line of research, in conjunction with for instance evaluating the robustness and adaptability of robots and robot populations.

The significance of this study is twofold. First, we presented a simulator for studying online evolution of realistic robot morphologies and controllers. Second, using this simulator we established that online and offline evolution are indeed different. This implies that the current practice of relying on offline evolution within evolutionary robotics has inherent limitations. To develop evolutionary mechanisms towards the long term goal of real-world robot evolution we recommend to use simulators that can handle online evolutionary systems.

Acknowledgement. The choice of modular robots used for this research is based on the design of Josh Aurebach's RoboGen project—a flexible and scalable modular robot design.

References

1. Eiben, A., Smith, J.: From evolutionary computation to the evolution of things. Nature **521**(7553), 476–482 (2015)
2. Eiben, A., Bredeche, N., Hoogendoorn, M., Stradner, J., Timmis, J., Tyrrell, A., Winfield, A.: The triangle of life: evolving robots in real-time and real-space. In: Liò, P., Miglino, O., Nicosia, G., Nolfi, S., Pavone, M. (eds.) Advances in Artificial Life, ECAL 2013, pp. 1056–1063. MIT Press, Cambridge (2013)
3. Auerbach, J., Aydin, D., Maesani, A., Kornatowski, P., Cieslewski, T., Heitz, G., Fernando, P., Loshchilov, I., Daler, L., Floreano, D.: RoboGen: robot generation through artificial evolution. In: Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems, pp. 136–137. The MIT Press, New York, July 2014. http://mitpress.mit.edu/sites/default/files/titles/content/alife14/978-0-262-32621-6-ch022.pdf
4. Sims, K.: Evolving 3D morphology and behavior by competition. Artif. Life **1**(4), 353–372 (1994)
5. Auerbach, J.E., Bongard, J.C.: Environmental influence on the evolution of morphological complexity in machines. PLoS Comput. Biol. **10**(1), e1003399 (2014)
6. Bongard, J.C., Pfeifer, R.: Evolving complete agents using artificial ontogeny. In: Hara, F., Pfeifer, R. (eds.) Morpho-functional Machines: The New Species, pp. 237–258. Springer, Tokyo (2003)

7. Komosinski, M.: The Framsticks system: versatile simulator of 3D agents and their evolution. Kybernetes **32**(1/2), 156–173 (2003)
8. Bongard, J., Lipson, H.: Evolved machines shed light on robustness and resilience. Proc. IEEE **102**(5), 899–914 (2014)
9. Lipson, H., Pollack, J.B.: Automatic Design and manufacture of robotic lifeforms. Nature **406**(6799), 974–978 (2000)
10. Jakobi, N., Husbands, P., Harvey, I.: Noise and the reality gap: the use of simulation in evolutionary robotics. In: Morán, F., Moreno, A., Merelo, J.J., Chacón, P. (eds.) ECAL 1995. LNCS, vol. 929, pp. 704–720. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59496-5_337
11. Koos, S., Mouret, J.B., Doncieux, S.: The transferability approach: crossing the reality gap in evolutionary robotics. IEEE Trans. Evol. Comput. **17**(1), 122–145 (2013)
12. Bongard, J., Zykov, V., Lipson, H.: Resilient machines through continuous self-modeling. Science **314**(5802), 1118–1121 (2006)
13. Eiben, A.E., Smith, J.E.: Evolutionary Robotics. Introduction to Evolutionary Computing. In: NCS, pp. 245–258. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-44874-8_17
14. Cully, A., Clune, J., Tarapore, D., Mouret, J.B.: Robots that can adapt like animals. Nature **521**(7553), 503–507 (2015). http://www.nature.com/articles/nature14422
15. Caluwaerts, K., Despraz, J., Işçen, A., Sabelhaus, A.P., Bruce, J., Schrauwen, B., SunSpiral, V.: Design and control of compliant tensegrity robots through simulation and hardware validation. J. Roy. Soc. Interface **11**(98), 20140520 (2014)
16. Nogueira, L.: Comparative analysis between Gazebo and V-REP robotic simulators. Seminario Interno de Cognicao Artificial-SICA **2014**, 5 (2014)
17. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. **18**(6), 1245–1262 (1989)
18. Weel, B., Crosato, E., Heinerman, J., Haasdijk, E., Eiben, A.E.: A robotic ecosystem with evolvable minds and bodies. In: 2014 IEEE International Conference on Evolvable Systems, pp. 165–172 (2014)
19. Jelisavcic, M., De Carlo, M., Hupkes, E., Eustratiadis, P., Orlowski, J., Haasdijk, E., Auerbach, J.E., Eiben, A.E.: Real-world evolution of robot morphologies: a proof of concept. Artif. Life **23**(2), 206–235 (2017). pMID: 28513201
20. Jelisavcic, M., De Carlo, M., Haasdijk, E., Eiben, A.E.: Improving RL power for on-line evolution of gaits in modular robots. In: 2016 IEEE Symposium Series on Computational Intelligence (SSCI), pp. 1–8. IEEE (2016)
21. Jelisavcic, M., Kiesel, R., Glette, K., Haasdijk, E., Eiben, A.E.: Analysis of Lamarckian evolution in morphologically evolving robots. In: Proceedings of the European Conference on Artificial Life 2017, ECAL 2017, pp. 214–221. MIT Press, September 2017