Automating the Incremental Evolution of Controllers for Physical Robots

Andrés Faíña*,**
Lars Toft Jacobsen**
Sebastian Risi**
IT University of Copenhagen

Keywords
Evolution of physical systems (EPS), evolution in hardware, evolutionary robotics

Abstract Evolutionary robotics is challenged with some key problems that must be solved, or at least mitigated extensively, before it can fulfill some of its promises to deliver highly autonomous and adaptive robots. The *reality gap* and the ability to transfer phenotypes from simulation to reality constitute one such problem. Another lies in the embodiment of the evolutionary processes, which links to the first, but focuses on how evolution can act on real agents and occur independently from simulation, that is, going from being, as Eiben, Kernbach, & Haasdijk [2012, p. 261] put it, "the evolution of things, rather than just the evolution of digital objects.…" The work presented here investigates how fully autonomous evolution of robot controllers can be realized in hardware, using an industrial robot and a marker-based computer vision system. In particular, this article presents an approach to automate the reconfiguration of the test environment and shows that it is possible, for the first time, to incrementally evolve a neural robot controller for different obstacle avoidance tasks with no human intervention. Importantly, the system offers a high level of robustness and precision that could potentially open up the range of problems amenable to embodied evolution.

1 Introduction

In evolutionary robotics (ER; [3, 33]), robot morphologies and control policies are optimized through artificial evolution. Such evolutionary optimizations are often performed inside a computer using simulations, which allows the fast evolution of large populations over many generations; only the best evolved individuals are then replicated in the physical world [10, 26]. Alternatively, optimization can be performed in a real-world physical substrate by evaluating the performance of an actual physical robot in its environment. While artificial evolution in the physical world is much more time-consuming than its simulation-based counterpart, it has some immediate benefits.

One major problem with simulation-based optimization is the aptly named *reality gap* [21]. It remains an open question in evolutionary robotics how to effectively address the problem that reality is notoriously difficult to model in simulations. Inaccurate models of the environment and the robot can lead to controllers or morphologies that underperform in reality, even though they appeared

* Contact author.
** IT University of Copenhagen, Copenhagen, Denmark. E-mail: anfv@itu.dk (A.F.); latj@itu.dk (L.T.J.); sebr@itu.dk (S.R.)

promising in simulation. Effectively, evolution can and will exploit features of the simulation that will either be different or not be present in reality.

While running evolution solely on a physical robot does effectively circumvent the reality gap problem [17, 18, 39, 43, 44], it also presents a number of challenges that go beyond the lower evolutionary speed. While it is straightforward in a simulator to reset a simulated robot back to its starting position or to test its performance in different environments, how to automate this process in the physical world is less clear. Many evolutionary experiments in the physical world still require significant support by human experimenters, such as (1) moving robots by hand to a particular position for evaluation [6, 17], (2) keeping track of objects in the environment to calculate fitness [17], or (3) manually reconfiguring the environment. These limitations have significantly restricted the type of behaviors that can be evolved on physical robots.

To further increase automation and to open up the scope of problems amenable to embodied evolution, this article presents a reality-based evolutionary system in which an industrial robot arm is able to automatically reconfigure and set up the training environment (including placing obstacles and moving the robot back to its starting position) for an obstacle avoidance and navigation task. The system uses computer vision to control the robot arm, manage the test environment, and provide measurements used in fitness calculations during the evaluation of a controller phenotype. Furthermore, the system can perform completely autonomous reality-based optimization of a control policy for a robot: (1) it governs the evolution of the robot's neural network controller and sets up the environment for each evaluation, and (2) it performs fitness calculations and advances to the next generation after completing the evaluation of the entire population. Additional features built into the system include a complex environment-building algorithm, to allow fast and robust operation. A series of experiments was conducted in order to evaluate the system in terms of robustness, precision, features, and ability to autonomously evolve well-performing control policies.

With this new setup in place, it is now possible to run more complex evolutionary experiments, such as *incremental evolution* [15], autonomously in the physical world. While incremental evolution has shown to facilitate the evolution of controllers for simulated [15, 29, 30] and real robots [16], the challenge to automatically reconfigure the domain environment in the physical world has so far hindered the adoption of such an approach for more complex tasks. Because of this limitation, controllers are often evolved incrementally in simulation and only afterwards transferred and fine-tuned in the real world [11]. In this article we show that the developed system is able to fully autonomously perform incremental evolution of increasingly complex obstacle avoidance tasks. Additionally, it is able to reconfigure the environments with a high degree of precision and fault tolerance. The hope is that in the future this system can become an additional tool to explore the evolution of autonomous robots directly in the physical world. In order to accelerate this process, the code and setup to run the experiments described here are available at https://bitbucket.org/afaina/embodiedevolution.

The article begins in the next section by reviewing related prior work in evolutionary robotics with an emphasis on embodied evolution (Section 2). Section 3 then details the hardware design of our approach, followed by its software components (Section 4). The experimental setup used to validate the system is explained in Section 5, followed by the results in Section 6. The article concludes with discussion and future work (Section 7).

2 Background

In evolutionary robotics [3, 33], evolutionary algorithms, a class of population-based metaheuristics, are used to optimize the control policy and/or morphology of an autonomous robot. This is contrary to mainstream robotics, where most aspects of the robot are designed from domain knowledge and where machine-learning algorithms are used to optimize the running system. While mainstream design and optimization techniques can make a robot perform a task fast and with a certain amount of reliability, they do not say anything about whether a different kind of robot actually would be more suited for the given application. Evolution, on the other hand, is not biased when it comes to

design choices or particular approaches (unless we introduce bias ourselves); it simply rewards the best performers, given an appropriate fitness function. Since biological evolution is the only force known to have created fully autonomous *and* adaptive systems, this holds great promises for robotics. Generally speaking, evolving some or all aspects of robots using population-based metaheuristics offers the benefit that few assumptions need to be made about the problem, but it also comes with costs. The developmental process offers no guarantee as to if or when an optimal morphology or controller is found, and often a large number of evaluations are required before the results become useful [3].

2.1 Neuroevolution

Artificial neural networks (ANNs) have long been the favored approach when modeling control policies [12, 33, 42]. This is mainly because of its roots in machine learning, where ANNs are used in decision making and become increasingly proficient at a task through learning strategies. Evolutionary robotics also embraces neural networks for robot controllers. But instead of engineering networks and employing machine learning techniques, like backpropagation, to optimize the network, neural networks are evolved artificially. This approach has been driving research in neuroevolution, because it calls for rather rich and complex representations schemes, in part due to the very large potential state space of neural networks. Among the more prominent work is Stanley et al.'s *neuroevolution of augmenting topologies*, or NEAT [41]. NEAT provides genetic encoding of networks in a linear fashion where markers make it possible to line up evolved features from two genomes during crossover—even after many generations. In combination with speciation, NEAT provides good protection of innovation, which could otherwise be lost in previously suggested representations. A feature that distinguishes NEAT from earlier methods, which often optimized ANNs with fixed topologies [33], is that it evolves connection weights and structure at the same time. NEAT can add both hidden nodes and connections and thus explore a large solution space.

2.2 The Reality Gap

A computer simulation has the advantage that large populations can be evaluated over many generations in a short amount of time. This advantage is what sparked research in evolutionary algorithms in the first place. It is also the reason that simulation is a common starting point in evolutionary robotics—the cost of building and optimizing robots in the real world is high.

However, because both the simulated agent and the environment it interacts with are only abstract models of the physical world, transferring controllers evolved in a simulator into physical robots has its challenges. If the model is inaccurate, this can lead to highly unexpected behavior in the real robot, even though the control policy or other aspect worked well in simulation. Essentially, evolution will exploit attributes of the simulator that are different from or not present at all in reality. This discrepancy between what is optimized in simulation and how it actually performs in reality is referred to as the *reality gap*.

For example, because of idiosyncrasies in robot sensors, they often respond slightly differently when exposed to the same stimulus. To alleviate this difficulty when transferring controllers to the real world, Miglino et al. [27] sampled robot sensors empirically and used the results in a simulator to set the activation levels for the simulated sensors. Controllers evolved through this approach work reasonably well when transferred to the real world [31, 32], but it is difficult to scale the approach to more complex environments. Others, such as Jakobi et al. [21], further point out the importance of taking great care in modeling the simulator in simulation-based optimization. Not only should the simulator model be based on large quantities of empirical data [27], but noise must also be introduced, since transducers in reality are noisy. Because simulation is so appealing, a lot of work has gone into improving the accuracy of simulation-based optimization in attempts to further close the reality gap. Jakobi [20] also introduced the concept of *minimal simulations* as a way to circumvent the reality gap. In this approach only the characteristics of the interaction between robot and environment are modeled

that are critical for the emergence of a desired behavior. Others, such as Lehman et al. [25], have shown that a more robust transfer to the real world can also be achieved by encouraging machines to be more *reactive* to their environmental inputs.

A different approach was introduced by Koos et al. [23]. Called the *transferability* approach, it considers how well a solution transfers from simulation to reality. So this becomes a multi-objective approach where simulators are optimized in parallel with the actual robot. The fitness of the simulators is derived from their ability to transfer to reality, and this measure in turn comes from evaluating the simulated controller in a real robot. More recently, and building on this work, the authors showed that an intelligent trial-and-error algorithm allows physical robots to quickly adjust to damage by creating a map of promising behaviors in a simulation beforehand [8].

Even with improved methods to perform simulation-based optimization, the reality gap remains a critical issue. Either reality-based optimization will directly be a part of the process, or, at a minimum, it will be the last step for any experiment that wants ultimately to produce a real robot.

2.3 Evolution of Physical Machines

In its strictest definition [9], *embodied artificial evolution* requires the embodiment of self-sustained evolution. That is, the controller or the morphology is evolved in place, in a running system, using resources available to the robot as part of its construction or in the environment. This is definitely hard to achieve, but more modest approaches that involve controller evolution in real robots and automation (e.g., by using other robots) has proven quite successful as well.

This kind of embodiment, and its relation to this project, is also shown in the work of Watson et al. [43], Heinerman et al. [17], Prieto et al. [35], and Bredeche et al. [5], where controllers are evolved and optimized in groups of real robots. The use of multiple robots allows parallel evaluation and reproduction among the robots, which speeds up the optimization process and allows the robots to evolve their behavior in the task environment. Because multiple real robots are involved, the approach also offers the opportunity to study the evolution of collective behaviors in the real world. Bredeche et al. [4] and Montanier and Bredeche [28] also showed that robot controllers can be evolved on line and onboard single robots. In their setup each robot maintains a population of controllers inside itself, which are evaluated sequentially. However, none of the aforementioned works include the automatic reconfiguration of the domain environment, and they often still require human intervention.

Early and foundational work in evolutionary robotics in the real world was performed by Nolfi, Floreano, and colleagues, of which their seminal textbook gives a good overview [33]. One of their most complex experiments carried out entirely in the physical world was the evolution of a robot that performs homing navigation [14]. The experiment took ten days and required the authors to introduce obstacles manually. In more recent work, Floreano and Keller [13] performed reality-based optimization using one or more robots in different environments. In this case the environments presented different tasks and hence different controllers. Again, the environments were static for the purpose of accomplishing a specific task.

Moving towards more automated approaches, Brodbeck et al. [6] recently presented a system that allows the automated manufacturing of physical robots. In their setup the phenotypes of evolved morphologies are created by an industrial robot. The genome is a set of instructions on how to assemble a set of only two components—a passive and an active element. The goal is to evolve speed of locomotion. The entire process is not fully autonomous, though; after the creation of a phenotype, it must be manually moved to the testing area, and the task is performed on different surfaces. Likewise the phenotype must be disassembled and the elements returned to the building area. The interesting part is the model-free phenotype development that Brodbeck et al. apply. It makes it possible to evolve morphologies, albeit constrained, that can be built autonomously by another robot. So this is an important step towards autonomous and adaptive systems.

There is a clear incentive to move forward in investigating automation and embodiment. Although simulation in practice cannot be disregarded, autonomous reality-based optimization may

prove useful in cases where transferability is very low, or simply to completely obviate simulate-and-transfer issues. In this article we aim to further increase automation in embodied evolutionary systems, by allowing them to also reconfigure the robot's training environment, as is explained next.

3 Experimental Design

Since evolutionary robotics often is applied in evolving only the controller for a specific platform to perform a certain task, or to make an inquiry into certain evolved behaviors, it makes sense to ask how automation and/or complete experiment autonomy can be achieved—in particular, if it can improve on previous work by establishing an entire optimization ecosystem that not only contains a standardized robotic platform, but also encapsulates automation of the most dominant tasks involved. This could be used to create environmental variations or reconfigurations, aid in collecting transferability metrics, and speed up the evaluation process in long-running experiments by avoiding the need for human intervention to set up the physical test environment prior to each evaluation. Other positive side effects of letting a robot conduct the tedious setup tasks involved in reality-based optimization may be greater precision and accuracy.

To investigate how automation can be applied to reality-based optimization, a test bed for conducting such experiments was developed. This is a complete setup that allows an industrial robot to be used in conjunction with a small evaluation environment and the necessary hardware and software infrastructure to conduct automated evolution and evaluation.

The experimental test bed (Figure 1) is based on an industrial robot (UR5); an environment, or *arena*, in which the optimization of a controller for a small robot is performed; and a computer vision system. The entire arena is well within the reach of the industrial robot. To track objects, a computer vision system is in place overlooking the primary work area of the UR5.

3.1 Control Robot

The Universal Robots UR5 industrial robot arm has six independent joints, has a spherical workspace that extends 850 mm from the base joint, and can carry a payload of up to 5 kg [37]. These attributes naturally pose some restrictions on the design of the experiments in general and on the evaluation area in particular.

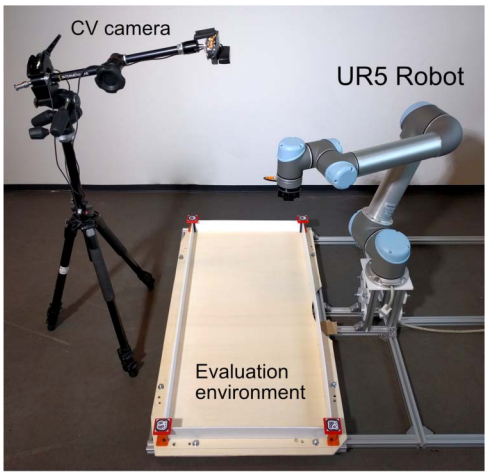


Figure 1. The main working area of the system, including the UR5 robot, the evaluation environment (arena) to contain robot(s) and objects, and the computer vision camera.

The UR5 comes with a control module that powers and controls the six joints. It features real-time control software that abstracts away the low-level hardware interfaces and exposes a high-level interface to movement commands and telemetry. The robot can be controlled or programmed directly from a pendant terminal or programmed and controlled via common communication interfaces like Ethernet and the industry standard MODBUS.

Communication between the UR5 and the main application is based on the TCP/IP socket interface over Ethernet, because it provides a robust and ubiquitous physical connection and because it provides access to a rich application program interface (API) that can be programmed using the proprietary URScript [36] programming language. URScript is a Python-like language that includes a number of modules that encapsulate different functionalities of the UR5: motion, math, internals (telemetry), and interfaces (GPIO, tool, etc.).

In the presented setup, the UR5 performs pick-and-place-like operations, for example, move to an object, pick it up, move to new location in the arena, and place the object back on the floor.

3.2 Robot End Effector

The robot arm end effector, or *tool*, needs to be compatible with the objects it manipulates. Many industrial effectors are grippers that can hold on to objects of varying width or, to some degree, shape. Thus, grippers are great universal tools, albeit expensive and complex. Instead of using a generic gripper, this system employs a custom-designed magnetic end effector with matching (mating) shapes to be placed on top of objects and robots used in the environment.

The end effector that attaches to the UR5 is very simple in design and has no moving mechanical parts. It connects with a matching shape that includes a ferrometallic disk at its center. Both can be seen in Figure 2. The use of a special object to match the tool is a choice to reduce complexity and improve robustness. This separate component must be duplicated and attached to the top of every object that is added to the environment.

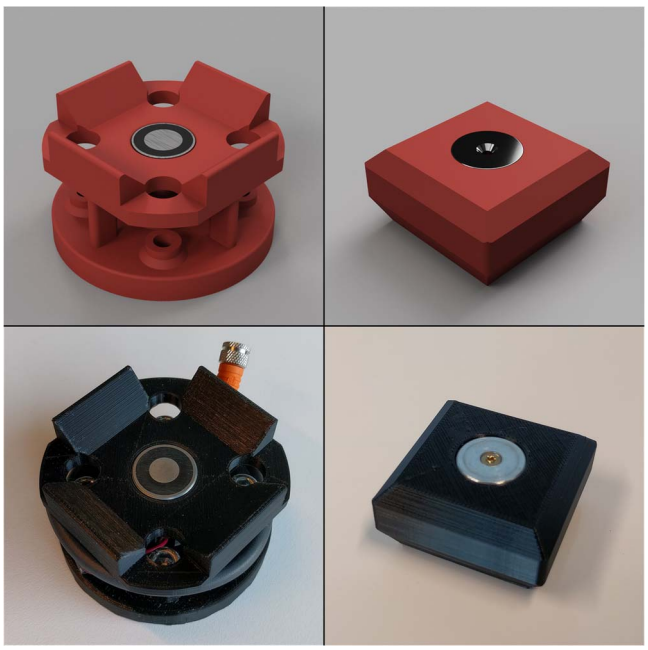


Figure 2. Robot end effector (tool) and connecting object. Top row: CAD renderings of the gripping component attached to the UR5, including the electromagnet in the middle (left) and the matching connector object (right). Bottom row: prototype 3D prints of the same components.

When the effector and object are aligned, the effector can be lowered onto the object and the disk will mate with an electropermanent magnet at the center of the effector. By using an electropermanent magnet, the actuator needs to be energized only when releasing an object—unlike a traditional electromagnet, which must be energized in order to create the magnetic field. This increases safety, primarily for a robot being picked up, since it will not get dropped in case of power or interface failure. The magnet is controlled via the UR5 I/O interface, which can sink the nominal current of the electromagnet, 0.25 A at 24 V, directly.

The electromagnet has a holding force of 45 N, and the end effector has been tested with a payload of 2.5 kg. This payload is the recommended maximum and will be stable even at an angle and under vibrations. The end effector weighs 111.0 g and thus leaves ample space up to the UR5 maximum payload of 5.0 kg.

The center of the tool's contact point is called the *tool center point*, or TCP. In this case the TCP lies at the center of the connecting surface of the electromagnet. It is important to know the exact location of the TCP relative to the center of the tool mounting flange on the UR5. The robot controller will then maintain a transformation matrix that ensures correct positioning according to the TCP configuration.

Tolerances in the end effector and matching shape design allow for both lateral and angular misalignment. Tapered edges at an angle of 45° on both components will center the object being picked up as it closes the gap between the electromagnet and the metal disk. Since the UR5 will lower the tool directly from above onto a given object, the amount of allowable misalignment depends on the frictional forces between the object and the surface. A light object with little friction between the bottom and surface will easily be centered by the tool forcing it towards its center. In this case the lateral misalignment can be quite large: up to 10 mm. A heavy object or an object with a high-friction material in contact with the arena surface cannot be pushed as easily by the tool, however. The worst case allows for approximately 1–2 mm of lateral misalignment.

3.3 Arena

The scope of the experiments in this article is limited to evolving simple behavioral controllers for a small robot working in a confined area, due to the limited reach of the UR5. The arena measures only 100 × 50 cm and is placed directly on the UR5 rig, approximately aligned with the robot base x and y axes. Although its small size makes it easy for the UR5 to pick up objects in the entire area, it severely limits the size of the robot and likewise the size of the objects that, for practical purposes, can be used in the reality-based optimization experiments. As a proof of concept, however, it will suffice for showing that simple task optimizations can be conducted autonomously in a system like this and may illustrate the promise for this approach.

The arena can be seen in Figures 1 and 3. It has walls on each side to keep the robot and environmental objects from falling or being pushed outside the arena during evaluations. Each corner has a fiducial marker to aid the computer vision system. Presently the arena is rectangular; however, it could be arbitrarily shaped, as long as the TCP can cover the entire area. The maximum area could be achieved by laying out the arena as a circular band around the UR5 base, using the recommended minimum and maximum operating radii.

4 Software Architecture

The system application, programmed in Python, contains two major subsystems that handle computer vision processing and UR5 control respectively. A task model encapsulates the jobs the system can perform and therefore also governs the artificial evolution if that is part of a job. All modules expose an interface tailored for easy task generation and specific to the capabilities of the system.

Computer vision runs in an independent thread and occupies its own module, responsible for image acquisition and processing. It also manages object data used to locate things in the environment and features calibration routines to ensure high precision and easy setup.

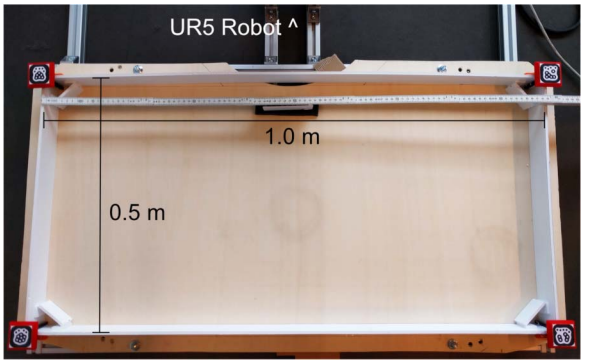


Figure 3. Arena top view. The arena measures 1 × 0.5 m. The UR5 base is located at the top center, right outside the image frame.

Robot control is managed in the UR5 module. Here, the system-specific functionality is implemented in terms of how the UR5 robot can interact with the environment.

4.1 Operational Design

As mentioned, the UR5 tool is designed to grip some special objects only. These objects can in turn be placed onto more unwieldy objects, and the combined objects can then be physically managed by the UR5 with respect to their placement in the arena. This can be either a robot with a connector on the top or an environmental building block. Even using simple, identical connectors for all objects, the system must still be able to accurately locate them and align the end effector with the connectors whenever an object needs to be moved. To minimize requirements for the positioning system, all objects in the environment are assumed to be of the same height and thus located in the same plane.

With all the objects having the same height, the positioning only needs to be expressed in two dimensions. Furthermore, the UR5 movements can be constrained to a few planes above the arena, while the orientation of the tool remains more or less fixed. This type of robot control is very similar to that of pick-and-place operations.

4.2 Computer Vision (CV) System

Object positioning is key to controlling the UR5 and to tracking a robot during evaluation. It goes without saying that positioning must be computerized to obtain the necessary performance and autonomy. A wealth of location technologies exists [24], but the field narrows down quickly on considering the application domain and the constraints imposed by using an industrial robot.

The UR5 has limited range and thus limits the size of the environment considerably. The system must be able to accurately position and orient the end effector in order to pick up objects and reposition them inside the arena. Furthermore, since the connection points of all objects will exist in the same plane, we need only obtain spatial information in that plane. To this end, fixed-camera computer vision is ideal. The environment is stationary and can easily be captured within a single image. Vision-based location can also extract the necessary spatial features, such as position and orientation, if the identifiable objects provide enough visual clues. Other approaches would require extensive instrumentation of the environment or not provide adequate accuracy or feature extraction capabilities, albeit some could provide richer spatial data that would otherwise require more than one camera. The choice is supported by computer vision being a proven technology in industrial pick-and-place automation—essentially what the industrial robot will be doing in this project.

The next problem is solving object identification and feature extraction. The computer vision (CV) system must be able to take a picture of the environment, identify movable objects, and extract the necessary spatial features to allow for accurate positioning of the TCP to perform pick-and-place

Figure 4. reacTIVision fiducials. An example of three small markers.

operations. These requirements, combined with robust object discrimination, are typically achieved using a marker system [22], using fiducial markers, that applies a special geometry tailored for CV approaches. The marker system thus consists of a set of patterns and an algorithm to identify them in an image and extract features that can yield additional spatial information.

4.2.1 Fiducial Tracking

Although devised for multitouch-surface applications, the fiducial tracking used in the reacTIVision system by Kaltenbrunner et al. [2] is a good match for this experimental setup. The fiducial makers have individual IDs, and their location and orientation can be obtained from the marker geometry. It is of course designed for strictly two-dimensional positioning, but this complies with the design decision to keep all objects, or at least their identifiable parts, in one plane and detectable by one camera alone.

The reacTIVision system (fiducial examples shown in Figure 4) uses topology-based identification. This is an approach to marker detection and identification, where a region adjacency graph is computed from a thresholded image. Each marker produces a unique left-heavy depth sequence.[1] By searching the entire adjacency graph for matching subtrees, markers can be detected and identified in a single operation. This makes topology-based identification quite fast, but also limited in data encoding and very sensitive to occlusion.

After marker detection, the location can be obtained as the average centroid (black and white leaves). An orientation vector can be computed using the average centroid and the average centroid of the black leaves.

These features position the reacTIVision system somewhere between simple color markers or the like and marker systems that carry more information and can be used for camera pose estimation (e.g., for augmented reality). This is ideal for our purpose because it fits the current operational design.

4.2.2 CV Implementation

For tighter integration with the other system components, the reacTIVision fiducial tracking has been reimplemented using OpenCV [19] in this project. The original implementation does not expose a programming interface, but instead provides a socket interface that can send marker data to a server. A new implementation in OpenCV provides the possibility of exposing programming interfaces in multiple languages. Because the common language for this project has been Python, the OpenCV Python API is the one being used.

The implementation is very faithful to the original [2] and provides identical behavior. Processing a single frame is performed in the following steps (Figure 5):

1. *Image preprocessing*. The image is undistorted and converted to grayscale.
2. *Thresholding*. A binary image is produced using an adaptive Gaussian thresholding and an erosion followed by a dilation to remove the noise.
3. *Segmentation*. The region adjacency graph is computed.

1 Uniqueness is guaranteed only within the set of generated fiducials. But because this was made for an image containing few other features than the markers, the likelihood of getting false positives is very low. Topological complexity, which results in adequately large subtrees, also increases robustness. However, there are some checks to validate the detected markers: They should have some maximum and minimum dimensions, and their leaf nodes should be placed inside the external contour of the marker. With these checks, we avoid false positives.

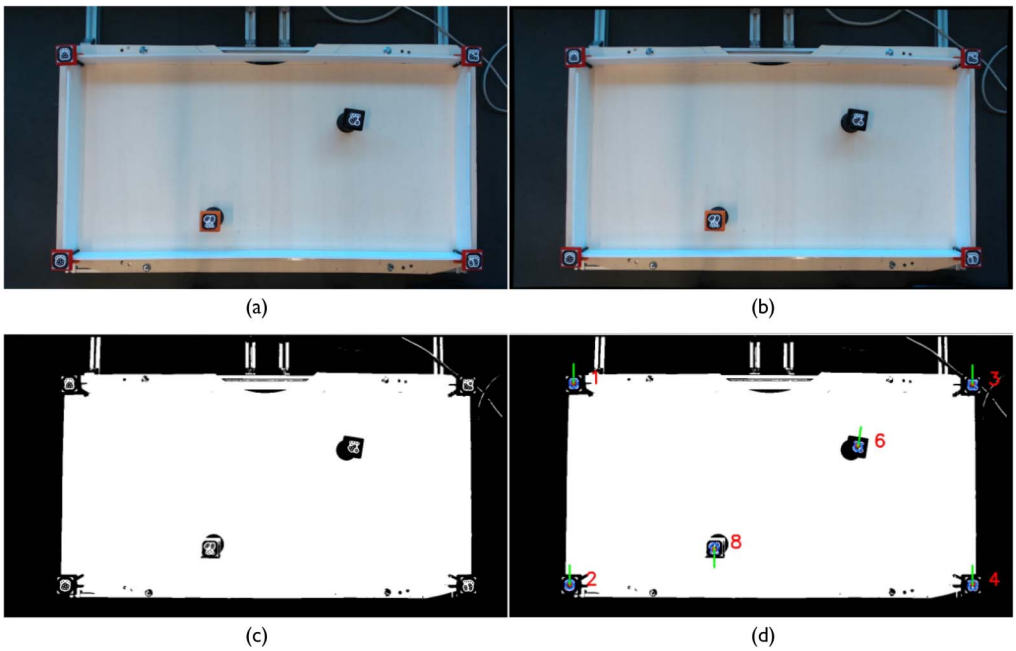


Figure 5. CV image processing: (a) the raw image, (b) the undistorted image, and (c) the undistorted image after thresholding. (d) Fiducials have been identified and their ID, location, and orientation (green line) have been overlaid on the thresholding image.

4. *Recognition*. By searching the adjacency graph for specific subtrees, fiducials can be recognized and their location and orientation computed.

Images are captured using a regular HD webcam at 1280 × 720 pixels. This resolution is chosen somewhat empirically as striking a balance between performance, precision, and recognition robustness. Higher resolution becomes slower to process but will provide locations with higher precision. Lower resolution may not capture enough details to make out all the distinct areas in each marker.

This computer vision system is immune to sudden light changes, as the camera automatically controls the white balance and the gain, and it works with natural light or artificial light. The only problem arises with light reflections caused by direct sunlight or strong lamps directly pointed towards the arena, but these situations can be easily avoided.

4.2.3 Vision Calibration and Transformation

The system operates on several coordinate systems, with the two most important being the vision coordinate system (pixel coordinates) and the UR5 base coordinate system.

The UR5 base coordinate system has its origin at the center of the robot base. This makes it a good reference for locating objects relative to the robot's position, especially in that the arena is bolted to the robot platform. The base x axis is parallel to the base and cuts through the middle of the arena. The y axis is also parallel to the base and is nearly parallel to the long sides of the arena. The z axis runs perpendicular to the base. See Figure 6 for an illustration.

The vision system operates solely on pixels, so a homography transformation is used to turn pixel coordinates into base coordinates that can be used to direct the UR5. The transformation matrix is obtained during a calibration procedure where the robot arm is placed on all four corner markers of the arena, and the four points reported through telemetry are stored. The vision system then grabs a frame and locates all four corner markers in the image. With the two sets of points from each

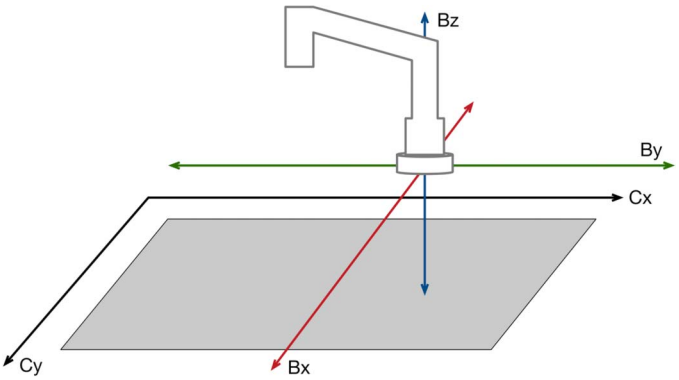


Figure 6. Camera and UR5 base coordinate systems. The *Cx* and *Cy* axes represent the camera perspective, and the *Bx*, *By*, and *Bz* axes constitute the coordinate system relative to the UR5 base.

coordinate system a transformation matrix can now be calculated by calling the findhomography function in OpenCV. The transformation matrix can now easily be stored for later use or recalculated using only the image; the camera is the only thing not attached to the rest of the structure and hence is more susceptible to shift, while the corner markers are practically fixed in their position relative to the base. In fact, the system recalculates the transformation matrix every generation to compensate the small displacements of the arena caused by the accelerations and decelerations of the UR5.

At run time the transformation matrix is passed to marker objects that initially only exist in pixel space. When the real-world coordinates are requested, they can be calculated as the dot product of the transformation matrix and the position vector.

4.3 Automation Control

Besides the CV implementation, the bulk of the software that makes up the system is concerned with orchestrating the UR5 movements based on information provided by the vision module. The UR5 module uses the urx Python library [38], which wraps all the URScript commands and handles socket communication over TCP/IP. The urx module will format and send commands to the robot, but it also provides a telemetry parser for real-time feedback.

By having a low-level command module in place, the UR5 system module becomes an interface for composite commands that describe high-level operations the robot must be able to perform in order to complete certain tasks. As it turns out, only a small set of atomic operations is needed to build all other operations per the operational design described in Section 4.1:

1. *Home*. Positions the robot arm in a fixed, known pose. This can be used to ensure it does not interfere with the camera, but it also provides a good starting point for pose transformations.
2. *Move to coordinates*. Moves the TCP to a coordinate and sets the orientation vector so that it points directly downward and is ready to pick up an object.
3. *Pick up object*. From its current pose, the TCP will be lowered to connect with an object and return to its starting pose again. The robot uses its ability to detect the exerted force and stop automatically.
4. *Place object*. Identical to the pickup operation, but once lowered, the electropermanent magnet will be energized and release the payload before returning to its starting pose.

The last three operations are then composed into a single pick-and-place operation, a function that moves the TCP to an object, picks it up, moves to a target location, and places it again. Organizing the objects in the arena is simply accomplished by a sequence of pick-and-place operations. However, performing any given sequence in a reliable and robust manner is not a simple task.

4.4 Building Environments

Despite the somewhat simple setup and agreed limitations on how environments can be designed within the arena, plenty of things can still go wrong. Automating reality-based optimization would not be very useful if it required constant oversight and regular intervention to recover from failure. To make the automation as robust and reliable as possible, the sequencing of pick-and-place operations has been encapsulated by a function called Build.

Build accepts a list of marker objects and a list of desired positions for the markers as arguments. The marker objects represent identified fiducials and hold their ID, location, and orientation—the latter two for both camera and UR5 base coordinates. The desired positions are UR5 base coordinates only and may or may not include a desired orientation. As an optimization feature, relaxation parameters for both location and orientation can be passed as arguments. The relaxation parameters are thresholds, and should a marker already be at its desired position within these thresholds, there is no need to perform a physical pick-and-place operation.[2]

When called, Build will use a custom algorithm to generate a pick-and-place sequence that is *conflict-free*. Because objects may be shuffled around the arena by a small robot or by hand when placing them the first time, their initial position can potentially conflict with the desired target position of another object. This means that the sequence ordering is important, to avoid an attempt to place an object on top of another. The algorithm that governs the sequence ordering works like this:

1. All target positions are checked against all marker positions. A conflict is defined as another object being within a predefined radius of the target for a given marker. In this step a conflict table will be built containing marker IDs associated with a list of markers that are in conflict, that is, positioned near the desired target.
2. If no conflicts were found in the first step, all objects can be moved in any order, and the sequence will just follow the list of markers provided as argument.
3. If the conflict table is nonempty, conflict resolution begins. Initially all markers with a clear target (no conflicts) will be queued for immediate pick-and-place (Figure 7, step 1). The conflict table is then iteratively processed in an attempt to further reduce the conflicting markers. The algorithm repeatedly adds markers to the queue whenever the conflict list is empty and removes them from the lists of the others (Figure 7, step 2). This approach solves most problems with objects standing in the way of others. In an additional step the system checks for any cyclical conflicts or deadlocks (e.g., object A is on B's desired target and vice versa).
4. If a deadlock has been detected, a new operation queue is created. A vacant position in the arena is established, and this will become the target for an intermediary move for the first marker still in conflict. Its originally desired target is deferred and queued. This process continues until all deadlocks have been resolved by intermediary moves to vacant positions (Figure 7, steps 3–6).

2 The occurrence of objects that either haven't moved at all or only shifted ever so slightly after an evaluation is quite frequent. It could be the evaluated robot itself that hasn't moved, or objects that haven't been touched.

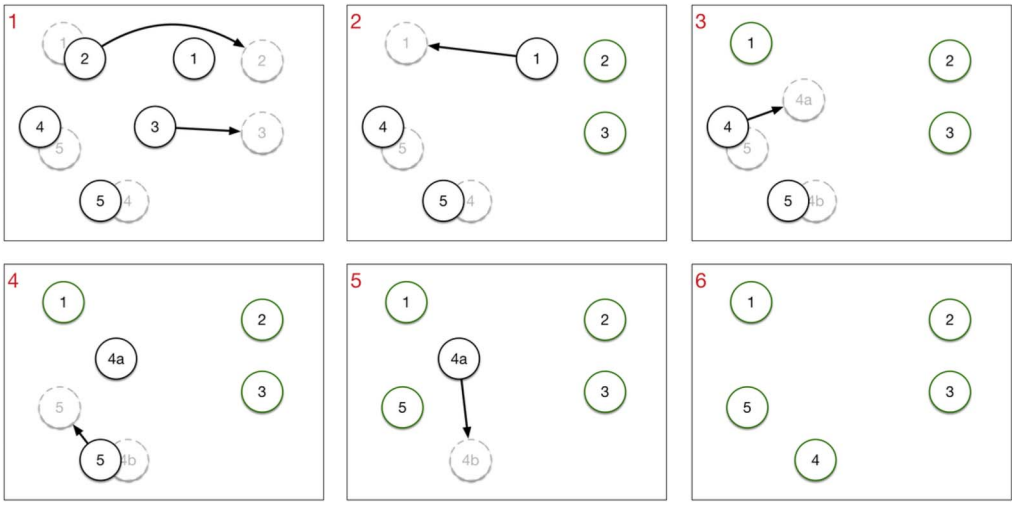


Figure 7. Environment building algorithm. The full black or green numbered circles are current positions, and the gray dashed circles are the desired target for that object. Steps 1–6 depict how the movement sequence is generated by first moving objects that have no positional conflicts, then moving the objects that have conflicts that were resolved after the first step, and finally resolving the more complex conflicts.

5. Finally, the sequence is executed using the pick-and-place function in the order established by the operation queue. If the deferred-moves queue is populated from deadlock resolution, this sequence will be executed immediately after.

Vacant positions are found by planar subdivision of the image, using the fiducial centers as vertices. The plane is divided into non-overlapping triangle regions using Delaunay's algorithm. All the regions within the four corner markers will then represent unoccupied space, and given a large enough space, an object can be placed within it. For the simple objects described later in Section 5.2 it suffices to check if the largest inscribed circle is larger than the diameter of the object.

5 Experimental Setup

The physical evolution system presented in this article is evaluated on a simple task of navigation and obstacle avoidance. These experiments test the ability of the system to run an embodied evolutionary experiment that requires environmental reconfiguration, in a completely automated fashion. This section first describes the mobile robot that navigates the arena, and then the objects that serve as the obstacles for the robot to avoid.

The neuroevolution system is based on NEAT [41] using the MultiNEAT [7] implementation—a C++ NEAT-HyperNEAT library with Python bindings. The system is essentially agnostic of the EA or neuroevolutionary implementation used, as long as it can be interfaced within the Python framework.

5.1 Mobile Robot

The evolving neural networks control a customized version of a Pololu Zumo 32U4 robot [34], a tracked robot whose small size (10 × 10 cm) makes it suitable for use in the small arena. It features a number of onboard sensors such as a 9-DoF IMU (accelerometer, gyro, and magnetometer), line-following sensor (down-facing array of IR reflectance sensors), side and front IR proximity sensors, and quadrature encoders on the motor shafts.

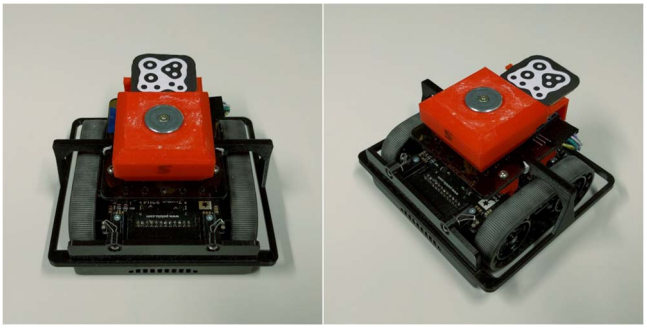


Figure 8. The Zumo 32U4 robot. It is a small tracked robot ca. 100 × 100 mm. It has encoders on the motor shafts, and an array of sensors that includes proximity, reflectance (for line following), and a 9-DoF IMU. The 360° bumper is 3D printed for this project and helps it avoid getting caught on walls or other objects. On the top plate, a red connector is mounted, which also houses a Bluetooth module for wireless communication.

An additional Bluetooth module is attached to the onboard microcontroller for wireless control of the robot and access to sensor data. The module is an RN42-XV Bluetooth module from Sparkfun [40] attached to an Adafruit Xbee adapter [1] for convenient pin access and power management, including logic level conversion. The RN-42 module is connected to the Zumo main board via the UART and power headers exposed on the top pin header for an auxiliary display. The robot is battery powered, and the system does not support autonomous charging for the time being. Notice in Figure 8 that the fiducials are offset to the back of the connector, while they are put directly over the connector for the objects. Because of the increased weight of the robot, placing the paper directly on the robot's connector would reduce the magnetic field to a level too weak to hold the robot. The total weight of the Zumo including batteries and all attached parts is 340.1 g.

5.2 Environmental Objects

The experiments reported in this article are all performed with only one type of environmental object, a small circular column about 45 mm in diameter with a fiducial connector on the top (Figure 9). It is light enough to be pushed around by the Zumo but dense enough to trigger the proximity sensors. Hence the objects can be used for both collection and collision avoidance tasks. A sturdy base ensures they are not knocked over by the Zumo on impact.

Figure 9. Environment objects. Items to be placed in the environment for the robot to interact with. They are light enough to be pushed, yet stable, and will not fall over easily. The tool connector with the fiducial is mounted on top.

5.3 Experiments

A set of four experiments are devised to evaluate the performance and feasibility of the system:

General performance evaluation of the automation control systems. This test will only make use of a single environment object. The UR5 will move the object around to random positions and register positional and rotational errors.

Building environments. A series of tests designed to challenge the robustness of the system's building algorithm.

Evolution of basic navigation skills. The goal in this simple navigation task is to evolve a controller that steers the mobile robot towards a given target. The main purpose of the robot arm is to automatically reset the mobile robot back to its starting position after the evaluation is over.

Incremental evolution of obstacle avoidance skills. This task is similar to the navigation task, but it includes objects in the environment that are placed by the robot arm and must be avoided by the mobile robot. We also compare a non-incremental evolutionary setup versus an incremental one in which the robot arm reconfigures the environment after a few generations to make it more complex.

The tests evaluate different aspects of the system, but combined they evaluate all the individual components. The first two tests are functional, and aim to establish that the system operates as intended per design. The other two tasks are application tests, which evaluate how the system manages to perform actual embodied evolution experiments. Each test and the results are described in detail in the next section.

6 Results

A video of the system in action can be found at https://youtu.be/7kzw_cvqTIw.

6.1 General Performance Evaluation of the Automation Control Systems

The UR5 can be repeatedly positioned with ±0.1-mm precision [37]. However, other factors can affect the precision and accuracy of the system, such as the resolution of the camera image (1,280 × 720 pixels). In the image the pixel distance between arena corner markers in the top row is 1,012. The real distance is 0.992 m. A pixel then represents 0.992 m/1012 = 0.00098 m, or just about 1 mm. Another source of error is the transformation matrix. If the base coordinates of the corner markers, for instance, are not accurate, this will affect the transformation matrix. The transformation matrix must also yield good results over a large area. Camera lens distortion is however not necessarily linear in the field of view, and even the camera matrix and distortion coefficients used to undistort the image may contain errors.

Eliminating all these errors is impossible; instead we focus on determining whether the described setup is adequate for all intended purposes. The system should be able to repeatedly position objects with a precision close to that of the camera, which in this case translates to ±1 mm. The mechanical design should then deal with the practical aspects of the achievable precision.

To measure the repeatable precision across the entire arena, a task has been developed to move an object (Figure 9) to random positions through random angles (rotation around the object z axis). After each placement the computer vision system takes a picture and calculates the absolute difference between the CV and the TCP coordinate, including the object's angle.

The test ran 854 pick-and-place operations. Figure 10 is a visualization of all the visited positions. The circle diameter is proportional to the lateral displacement sum (positional error),

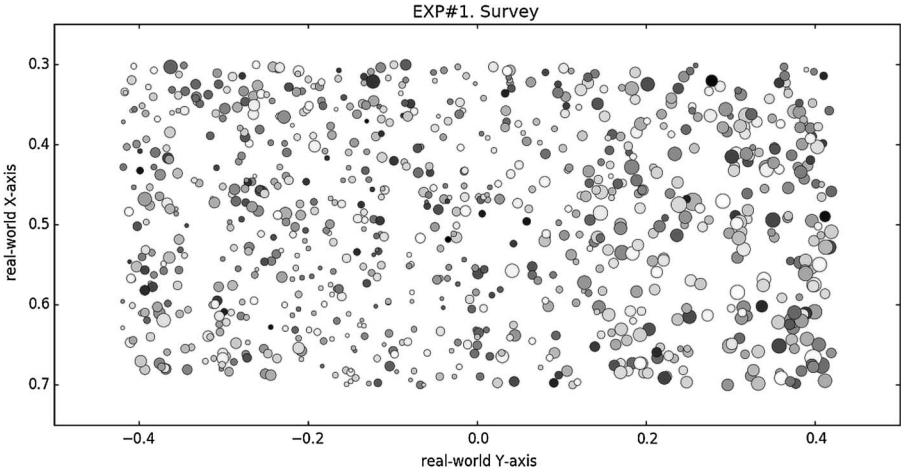


Figure 10. Positioning plot. The same object was repeatedly moved to a random position in the arena. The circle diameter is proportional to the positional error (err_x + err_y). Shading indicates the orientation error, darker being smaller.

$|x_b - x_c| + |y_b - y_c|$, while the shading indicates the orientation error, darker being smaller. The main purpose of this plot is to indicate coverage and to see if errors have a tendency to cluster (i.e., be greater or less in certain regions). The plot shows that coverage from random positions is quite good and that there clearly is a tendency to have larger errors around the edges, especially in the right side. Angular errors on the other hand seems to be more or less evenly distributed, which is consistent with observations made during the test.

Table 1 summarizes the results from the positional and angular error measurements. The mean error for both the x and y axes is $\leq$2 mm (s.d. = 1 mm). The orientation, denoted by the angle θ, has a mean of 2.9°. The actual precision of these measurements is determined by the camera resolution, and at worst 1 mm can be achieved. Overall this is quite acceptable in view of the many possible error sources. It is worth noting, though, that the maximum positional error is between 5.1 and 5.5 mm. But this is still within the range that can be mitigated by the mechanical design of the end effector. The orientation with a maximum of 10.34° is prone to become a bigger problem, because it works against the centering mechanism in the effector and thus must not be overlooked. However, at no point during the test did the UR5 fail to pick up the object. Whether the precision is acceptable for a given application would be a matter of requirements, but is definitely satisfactory for robust operation within the scope of the experiments described in this article.

During the random pick-and-place task, time and image errors were reported every 25 operations (Table 2). On average a pick-and-place operation takes 29 s, and this is a worst-case scenario because the UR5 starts and ends in its home position. Chained pick-and-place operations will be faster on average. If the quality of the captured image is not sufficient to detect the markers accurately, or at

Table 1. Positional and angular error summary. Data is obtained from 854 points. The error is the absolute difference between the coordinate the UR5 reports and the one computed from the vision transformation.

Axis	Max	Mean	Std. Dev.
x (mm)	5	2	1
y (mm)	5	1	1
θ (°)	10.3	2.9	2.1

Table 2. Additional performance data from the system evaluation.

Mean time/set	Mean time/op	No detection	Recapture
712 s	29 s	39	4.5%

all, the system will retry until it sees all the markers, and this happened 39 times during the test, yielding a recapture percentage of 4.5%. The CV system runs separately in another thread, which processes the camera images and provides the markers to the main thread. Therefore, the performance of the system is good enough to track the Zumo robot. However, a stop-and-go approach, where the robot can only perform a single discreet motion and then stops, has been applied to make the results more reproducible.

6.2 Building Environments

A fundamental feature of the system is its ability to manage the placement of multiple objects in the arena. In this case the system needs to keep track of all objects and control for objects being placed in the same location or interfering with each other. The Build algorithm (Section 4.4) is evaluated on rearranging objects (four environment objects and one Zumo robot) from a given start configuration into a desired final configuration (Figure 11). The system is tested on increasingly complex starting scenarios, in which (1) objects do not occupy any of the target positions, (2) some target positions are occupied, and (3) target positions are occupied and include deadlock situations.

The evaluation shows that the system is able to successfully deal with all three test scenarios. As Figure 12 demonstrates, the rearranged order is highly dependent on conflicts and deadlocks. The average time to complete building a configuration was 127 s for 25 runs. Carrying out a series of such building evaluations without errors is not to say the system will handle all possible situations gracefully. During the evolutionary experiments described in the next sections, problems were sometimes observed when the objects were right next to each other. In some instances an object would touch the one next to it slightly when picked up, due to the protruding edges of the connector. If this happens, the image just used to locate all objects will be invalidated, and the system must back off and assess the scene once more to obtain object positions accurately. Additionally, a few errors occur due to the system trying to exceed the UR5 joint limits. But that is rare, and if the tasks are programmed properly, they can be restarted with minimal loss of work.

6.3 Evolution of Basic Navigation Skills

The goal of the first embodied evolution experiment is to demonstrate and validate the system's capabilities. In this simple navigation task, the goal of the mobile robot is to drive from point *A* to *B* (whose locations are fixed during the experiment) in an arena without any obstacles.

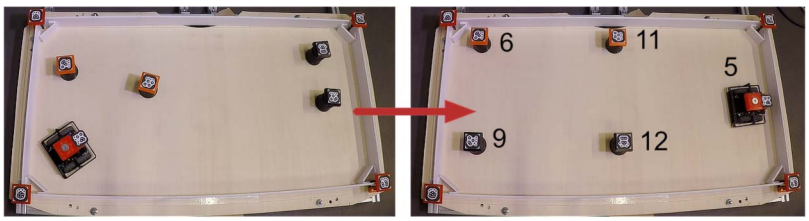


Figure 11. Building feature evaluation. The system must reorganize the objects and arrange them from an initial state (left) to a target state (right). While the initial object placement is different for the three test cases, the target state is always the same. The numbers are the object IDs.

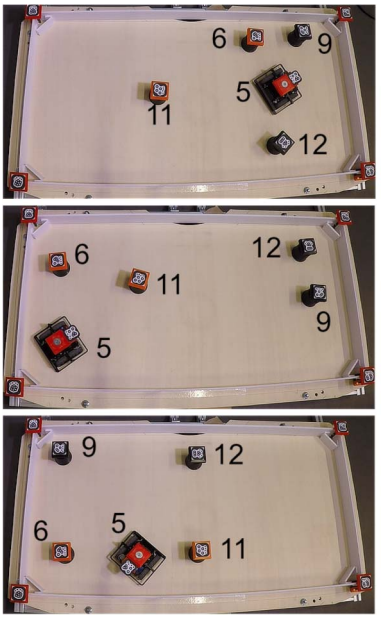


Figure 12. Movement ordering for various problems. The images show three different test cases and the order of movements, by object ID, chosen by the Build algorithm to solve the problem. From all three different starting configurations, the Build algorithm is able to rearrange the objects into the target configuration shown in Figure 11.

The neural networks controlling the robot are evolved by NEAT [41], and have one input, a bias, and one output (Figure 13). The MultiNEAT parameters can be found in the Appendix. The ANN receives as input the angle between the Zumo robot's heading and that of a virtual target in the arena (in the image coordinate system), mapped to [0.0, 1.0]. The single output is mapped to three actuation commands that can be sent to the Zumo robot: A value in the range [0.0, 0.45] will make the robot turn left; [0.45, 0.55] move forward; [0.55, 1.0] turn right. The motors run in opposite directions at the same speed when the robot is turning, and in the same direction at equal speed when the robot is driving forward. The speed and time for each actuation command are detailed in the Appendix.

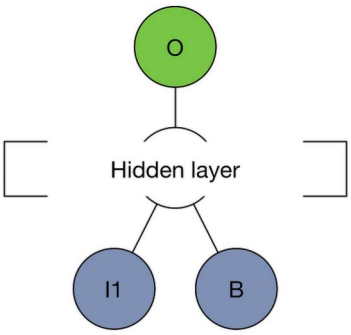


Figure 13. Neural network representation for the basic navigation task. Inputs are angle to target (I1) and bias (B). The single output is mapped to movement commands.

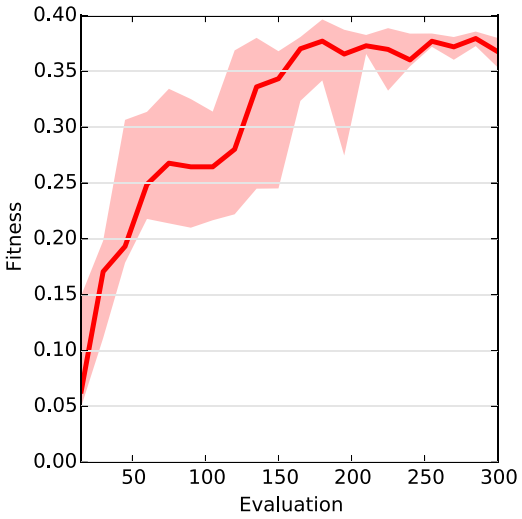


Figure 14. Fitness over generations in the navigation task. The graph shows the median and the 25th and 75th percentiles of six independent evolutionary runs. With the exception of a few dips, fitness generally increases over generations.

A robot is evaluated for a maximum duration of 35 steps, which take around 250 ms each. The CV system tracks the robot's position and angle. After running for N steps, the fitness is calculated as the accumulated remaining distance d_s to the target divided by the total number of steps:

$$F = \frac{1}{N}\sum_{s=1}^{N}(1 - d_s)^2.$$

At the beginning of each evaluation, the UR5 places the robot at the center of the arena. The mobile robot is then allowed to move around and is automatically moved back to the starting position by the UR5 once the evaluation is over. To encourage the evolution of general navigation skills, each robot is evaluated on its ability to approach two different targets, which are at fixed positions. The final fitness is the average of the two runs. All the experiments were run with a population size of 15 individuals for 20 generations.

Figure 14 shows the median of the best genome in each generation for six independent runs. On average, each evaluation takes 25 s (50 s for the two targets). Thus, a run of the experiment takes around 4 h 20 min to complete in total. Fitness is generally improving over generations, but some noticeable dips in fitness are also visible. These results are likely due to the fact that the controller of the best robot from the previous generation is not robust yet and the noise in the marker orientation can generate different behaviors for the same controller. However, the neural networks found in the last generations are more robust, and this noise does not cause a drop in fitness.

Figure 15 shows the paths taken by the champion network in each generation. Lines are colored light gray to black in order of generations, early to late. The plot only shows the path taken in the second of the two evaluation rounds. Over time, more and more behaviors converge on a path towards the goal.

To further validate the result, the best network evolved after 20 generations was tried against five new targets using the same initial position. Figure 16 shows the robot's paths for each of the five targets. The results demonstrate that the robot evolved general navigation skills. Whereas similar embodied evolutionary experiments have been conducted in the past, also using much simpler mechanisms to manage the environment [33], the results reported here do display a completely unsupervised embodied evolution setup made possible by a standard industrial robot arm and computer vision algorithm.

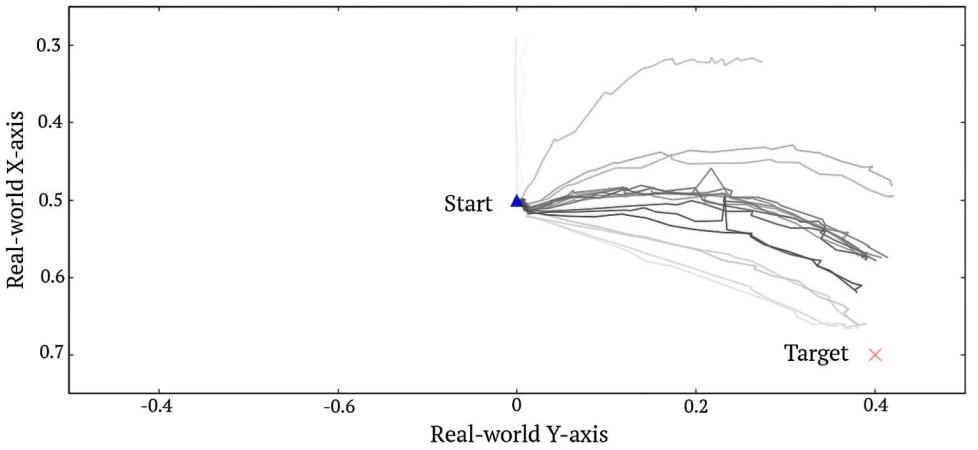


Figure 15. Robot paths from the first evolution experiment. Plots of the second evaluation paths taken by the best genome in each of the 20 generations of the first run. Generations are shaded light to dark in increasing order.

6.4 Incremental Evolution of Obstacle Avoidance Skills

Some of the more interesting prospects arise from the system's ability to work with more complex environments and alter them automatically during optimization. In the second experiment, we compare an incremental versus a non-incremental evolution approach that critically depends on the system's ability to reconfigure the environment. The domain is similar to that of the navigation task (the mobile robot has to approach a given target), but now the robot must also avoid obstacles that are placed in the direct path from its initial position to the target. In the incremental evolution setup, the robot is first evaluated for three generations in two variations of a simple environment (Figure 17a, b), before being evaluated in two variations of a more complex environment (Figure 17c, d). In the non-incremental version, robots are directly evaluated in the more complex environment.

The robot arm is now responsible for (1) moving the robot to its starting position at the beginning of each evaluation and (2) setting up the different obstacle layouts for each of the evaluations. In this setup the CV system is used to calculate the relative angle to the target, calculate the distance from the robot to the goal, and detect if the robot collided with any of the obstacles.

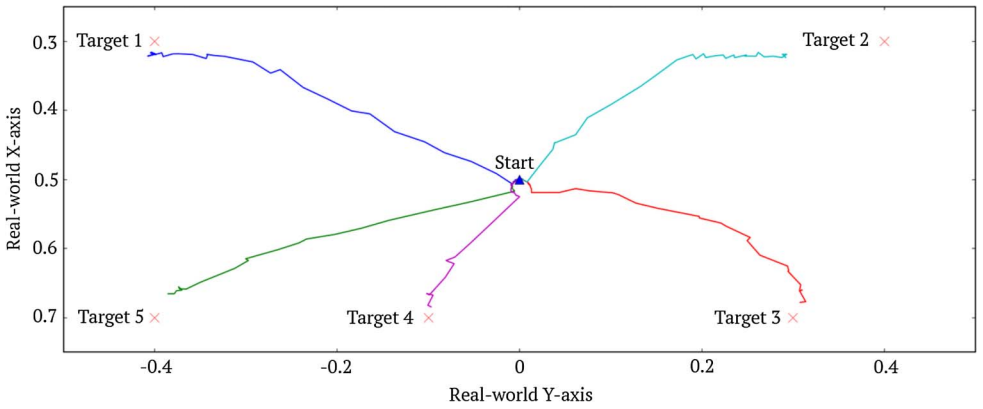
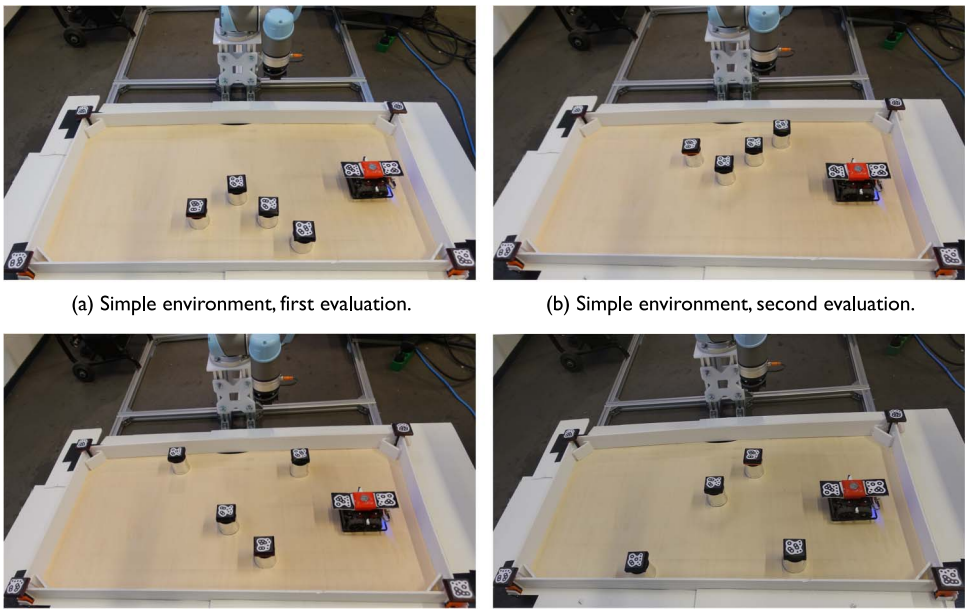


Figure 16. Generalization test. The best network discovered after 20 generations is tried against five new targets. The crosses mark the targets. The starting position is at the center. The results demonstrate that evolution discovered a general navigation strategy.



(a) Simple environment, first evaluation. (b) Simple environment, second evaluation.

(c) Complex environment, first evaluation. (d) Complex environment, second evaluation.

Figure 17. Incremental evolution setup. Shown are the two setups (a, b) and (c, d) used for the incremental evolution setup. The goal of the robot is to navigate from its start location on the right to the target location on the far left without colliding with any of the obstacles. In the non-incremental setup the agent is directly evolved in two variations of a complex environment (c, d). In the incremental setup the robot is first evolved in the simpler environment (a, b) for three generations, before being evaluated in the harder environment (c, d).

The neural networks for this task have three inputs (excluding bias) and a single output (Figure 18). The three inputs are the relative angle to the target and two proximity indications based on the front proximity sensor readings while IR light is emitted from the left and right sides of the robot. Similarly to the first evolutionary experiment, the single output is mapped to three movement commands: turn left, turn right, and move forward. If the robot collides with an obstacle, the evaluation is stopped. The fitness function rewards getting close to the target, and it is calculated as the average fitness for the two configurations:

$$F = \frac{1}{2}\sum_{c=1}^{2}(1 - d_c).$$

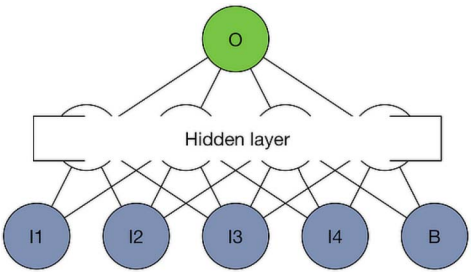


Figure 18. Neural network representation for the obstacle avoidance task. Inputs are angle to target (I1), left, front, and right proximity sensors (I2, I3, and I4), and bias (B). The single output is mapped to movement commands.

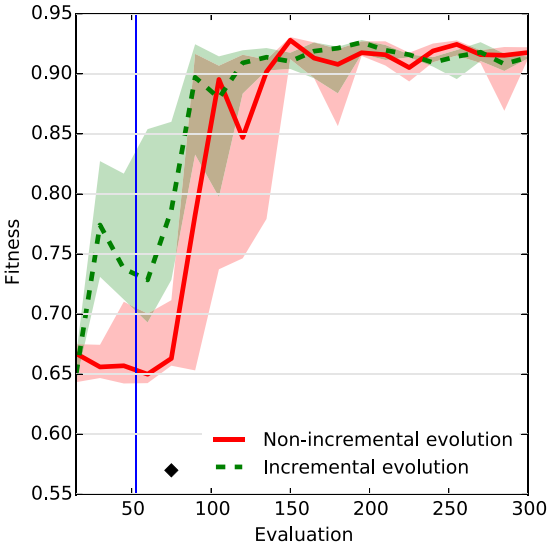


Figure 19. Fitness over generations for the obstacle avoidance task. The graphs show the median and the 25th and 75th percentiles. All results are based on six independent evolutionary runs. In the incremental approach the fitness initially increases faster (significantly different in generation 4), after which the two approaches reach about the same fitness level. The switch from simple to complex environment for the incremental approach is indicated by the vertical line.

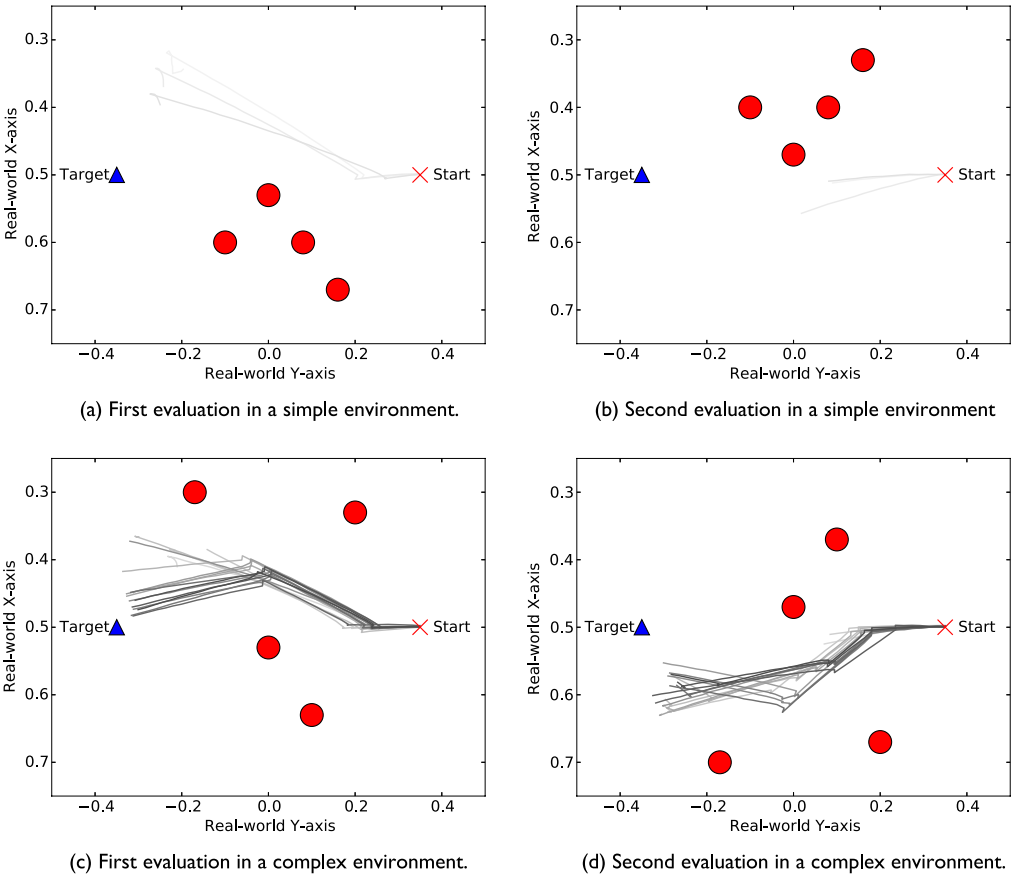
The NEAT settings and movement parameters for the obstacle avoidance tasks are detailed in the Appendix. Experiments ran for a total of twenty generations, which took approximately 6.5 h in total for each evolutionary run. Six independent evolutionary runs were performed.

Figure 19 shows fitness over generations. For both the incremental and non-incremental approach, fitness generally improves. The results show that the incremental setup evolves high-performance controllers slightly faster, with a significant difference in generation 4 ($p < 0.05$; two-tailed Mann-Whitney U test). While the advantage is lost as evolution continues, the setup in this article does demonstrate, for the first time, that fully automated incremental evolution is possible in the real world. The paths taken by the best robot from each generation of one evolutionary run are shown in Figure 20. Robots evolved the ability to navigate to the goal without colliding with any obstacles.

7 Discussion and Future Work

The presented system is capable of performing fully automated reality-based optimization of a neural network control policy using a single robot in a controlled environment. A simple controller can be evolved incrementally and without human intervention within a reasonable time frame. Additionally the system yields predictable behavior in managing the environment. Although a successful outcome is highly dependent on the experimental design and the system limitations, it still provides a robust experimental platform and demonstrates a novel approach to embodied artificial evolution that overcomes simulate-and-transfer problems.

A significant part of the work went into the construction of a physical framework and a software process framework to manage reality-based optimization and robot control automation. Creating robust robotic automation for a dynamic environment is a difficult task in itself, and the system design will necessarily pose restrictions on the problem domain one wants to explore with the system. The end effector and matching connector constituted an attempt to increase generalization and allow any kind of object to be added as long as it has a connector mounted on the top and is within the payload limitation. On the other hand, the system was confined to work on a flat surface, and



(a) First evaluation in a simple environment.

(b) Second evaluation in a simple environment

(c) First evaluation in a complex environment.

(d) Second evaluation in a complex environment.

Figure 20. Incremental evolution champions. Shown are the paths of the best robots found in each of the 20 generations for one evolutionary run. Over generations, robots evolve the ability to approach the target for the two different obstacle layouts. Note that in the simple environment only three paths are shown, because the robot is only evolved for three generations in the simple setup.

this is further enforced by the connectors being a hindrance to stacking. Ultimately experiments are limited by and must be designed around the limiting factors of the physical framework.

One insight from the performed experiments is that automating environment-building and evaluation setups are comparable in complexity to automatic phenotype creation. There are many things that can go wrong, as the automation part operates in the very same volatile environment that is so hard to model correctly in simulation. This can potentially lead to conflicts between system abilities and the desire to impose as few constraints on evolution as possible. In that respect there is no difference from the problems facing automatic phenotype creation, but it could be considered an added limitation if the two approaches were to be combined.

The results in this article show that a control policy can be evolved incrementally and autonomously with no transferability problems. However, it also shows that a priori knowledge about the appropriate evolutionary parameters is needed to successfully conduct reality-based optimization and achieve a good result. These parameters can be obtained empirically by using the system in a series of *preliminary attempts*. This approach may work well for very simple tasks where an indication of progress can be observed early on. For more complex tasks this may not be feasible, and simulation might be used to generate sensible parameters and starting conditions.

If a simulation is used to generate a starting point for reality-based optimization, we are facing the reality gap once again. But since the aim of the simulation is not to evolve a well-performing

controller in reality but to provide approximations of the system parameters, it could be a feasible approach to cutting the total time of the experiments. And time is likely the most significant downside of a method that completely leaves out simulation-based optimization. Seeding the population could be another time-reducing approach to evolving more complex behaviors. The seeding genomes could encode control policies for subtasks that have been evolved previously, using either reality-based or simulation-based optimization. Another promising direction could be to build on the intelligent trial-and-error approach introduced by Cully et al. [8], which could limit the number of evaluations that have to be performed in the real world.

It would be interesting to explore how the presented approach or a similar system could work together with a manufacturing system like the one presented by Brodbeck et al. [6]. Combining the ability to evolve morphologies with the ability to reconfigure the environment could be a step towards more complete *assisted* embodied artificial evolution. Another potential future direction for this system is automating optimization of the robot-in-the-loop type, where a transferability function is used to evaluate the simulator based on real robot performance.

In summary, we have presented a system that employs a robot control architecture capable of achieving a robust autonomous test environment and robot management, requiring no human intervention. The system features a generic design that allows flexibility in the choice of objects and robots used in the arena, and could be the starting framework for more complex embodied evolution experiments in the future. It is important to note that the system is not limited to evolutionary algorithms; in the future it could also allow robot controllers to be optimized by reinforcement learning completely in the real world.

Acknowledgments

This project was partially funded by the European Union's Horizon 2020 research and innovation program under FET grant agreement 640959 (Flora Robotica project). Computation and simulation for the work described in this article were supported by the DeIC National HPC Centre, SDU.

References

1. Adafruit (2015). Xbee adapter v1.1. https://www.adafruit.com/products/126.
2. Bencina, R., Kaltenbrunner, M., & Jorda, S. (2005). Improved topological fiducial tracking in the reactivision system. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition—Workshops, 2005. CVPR Workshops* (pp. 99). Piscataway, NJ: IEEE.
3. Bongard, J. (2013). Evolutionary robotics. *Communications of the ACM*, *56*(8), 74–83.
4. Bredeche, N., Haasdijk, E., & Eiben, A. E. (2010). On-line, on-board evolution of robot controllers. In P. Collet, N. Monmarché, P. Legrand, M. Schoenauer, & E. Lutton (Eds.), *Artifical evolution: 9th International Conference, Evolution Artificielle, EA, 2009, Strasbourg, France, October 26–28, 2009. Revised selected papers* (pp. 110–121). Berlin, Heidelberg: Springer.
5. Bredeche, N., Montanier, J.-M., Liu, W., & Winfield, A. F. (2012). Environment-driven distributed evolutionary adaptation in a population of autonomous robotic agents. *Mathematical and Computer Modelling of Dynamical Systems*, *18*(1), 101–129.
6. Brodbeck, L., Hauser, S., & Iida, F. (2015). Morphological evolution of physical robots through model-free phenotype development. *PloS One*, *10*(6), e0128444.
7. Chervenski, P. (2015). MultiNEAT. https://github.com/peter-ch/MultiNEAT.
8. Cully, A., Clune, J., Tarapore, D., & Mouret, J.-B. (2015). Robots that can adapt like animals. *Nature*, *521*(7553), 503–507.
9. Eiben, A., Kernbach, S., & Haasdijk, E. (2012). Embodied artificial evolution. *Evolutionary Intelligence*, *5*(4), 261–272.
10. Faíña, A., Bellas, F., López-Peña, F., & Duro, R. J. (2013). Edhmor: Evolutionary designer of heterogeneous modular robots. *Engineering Applications of Artificial Intelligence*, *26*(10), 2408–2423.
11. Filliat, D., Kodjabachian, J., Meyer, J.-A. (1999). Incremental evolution of neural controllers for navigation in a 6-legged robot. In M. Sugisaka & H. Tanaka (Eds.), *Proceedings of the Fourth International Symposium on Artificial Life and Robots* (pp. 753–760). Oita, Japan: Oita University Press.

12. Floreano, D., Dürr, P., & Mattiussi, C. (2008). Neuroevolution: From architectures to learning. *Evolutionary Intelligence*, *1*(1), 47–62.

13. Floreano, D., & Keller, L. (2010). Evolution of adaptive behaviour in robots by means of Darwinian selection. *PLoS Biology*, *8*(1), e1000292.

14. Floreano, D., & Mondada, F. (1996). Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *26*(3), 396–407.

15. Gomez, F., & Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, *5*(3–4), 317–342.

16. Harvey, I., Husbands, P., & Cliff, D. (1994). *Seeing the light: Artificial evolution, real vision*. School of Cognitive and Computing Sciences, University of Sussex Falmer.

17. Heinerman, J., Zonta, A., Haasdijk, E., & Eiben, A. (2016). On-line evolution of foraging behaviour in a population of real robots. In G. Squillero, & P. Burelli (Eds.), *Applications of evolutionary computation, 19th European Conference, EvoApplications 2016* (pp. 198–212). Berlin, Heidelberg: Springer.

18. Hornby, G., Fujita, M., Takamura, S., Yamamoto, T., & Hanagata, O. (1999). Autonomous evolution of gaits with the Sony quadruped robot. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, & R. E. Smith (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, vol. 2 (pp. 1297–1304). San Francisco: Morgan Kaufmann.

19. Itseez (2016). OpenCV—open source computer vision library. http://opencv.org/

20. Jakobi, N. (1998). *Minimal simulations for evolutionary robotics*. Ph.D. thesis, University of Sussex.

21. Jakobi, N., Husbands, P., & Harvey, I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. In F. Morán, A. Moreno, J. J. Merelo, & P. Chacón (Eds.), *Advances in artificial life, Third European Conference on Artificial Life* (pp. 704–720). Berlin, Heidelberg: Springer.

22. Kohler, J., Pagani, A., & Stricker, D. (2011). Detection and identification techniques for markers used in computer vision. *Visualization of Large and Unstructured Data Sets—Applications in Geospatial Planning, Modeling and Engineering*, *19*, 36–44.

23. Koos, S., Mouret, J.-B., & Doncieux, S. (2013). The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Transactions on Evolutionary Computation*, *17*(1), 122–145.

24. LaMarca, A., & De Lara, E. (2008). Location systems: An introduction to the technology behind location awareness. *Synthesis Lectures on Mobile and Pervasive Computing, 3*(1), 1–122.

25. Lehman, J., Risi, S., D'Ambrosio, D., & Stanley, K. O. (2013). Encouraging reactivity to create robust machines. *Adaptive Behavior*, *21*(6), 484–500.

26. Lipson, H., & Pollack, J. B. (2000). Automatic design and manufacture of robotic lifeforms. *Nature*, *406*(6799), 974–978.

27. Miglino, O., Lund, H. H., & Nolfi, S. (1995). Evolving mobile robots in simulated and real environments. *Artificial Life*, *2*(4), 417–434.

28. Montanier, J.-M., & Bredeche, N. (2011). Embedded evolutionary robotics: The (1+1)-restart-online adaptation algorithm. In S. Doncieux, N. Bredeche, & J.-B. Mouret (Eds.), *New horizons in evolutionary robotics* (pp. 155–169). Berlin, Heidelberg: Springer.

29. Mouret, J.-B., & Doncieux, S. (2008). Incremental evolution of animats' behaviors as a multi-objective optimization. In M. Asada, J. C. T. Hallam, J. A. Meyer, & J. Tani (Eds.), *International Conference on Simulation of Adaptive Behavior* (pp. 210–219). Berlin, Heidelberg: Springer.

30. Mouret, J.-B., Doncieux, S., & Meyer, J.-A. (2006). Incremental evolution of target-following neuro-controllers for flapping-wing animats. In S. Nolfi, G. Baldassarre, R. Calabretta, J. C. T. Hallam, D. Marocco, J.-A. Meyer, O. Miglino & D. Parisi (Eds.), *From Animals to Animats 9, 9th International Conference on Simulation of Adaptive Behavior* (pp. 606–618). Berlin, Heidelberg: Springer.

31. Nolfi, S. (1996). Adaptation as a more powerful tool than decomposition and integration. In P. K. Simpson (Ed.), *Proceedings of the Workshop on Evolutionary Computing and Machine Learning, 13th International Conference on Machine Learning*, vol. 1 (pp. 141–146). Piscataway, NJ: IEEE.

32. Nolfi, S. (1997). Evolving non-trivial behaviors on real robots: A garbage collecting robot. *Robotics and Autonomous Systems*, *22*(3), 187–198.

33. Nolfi, S., & Floreano, D. (2000). *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. Cambridge, MA: MIT press.

34. Pololu (2016). Pololu zumo 32u4 robot. https://www.pololu.com/category/170/zumo-32u4-robot.

35. Prieto, A., Becerra, J., Bellas, F., & Duro, R. J. (2010). Open-ended evolution as a means to self-organize heterogeneous multi-robot systems in real time. *Robotics and Autonomous Systems*, *58*(12), 1282–1291.

36. Robots, U. (2015). *The URScript programming language*. Universal Robots A/S, version 3.2 ed.

37. Robots, U. (2015). *User manual, UR5/CB3*. Universal Robots A/S, version 3.1 ed.

38. Roulet-Dubonnet, O. (2015). python-urx. https://github.com/oroulet/python-urx. PYPI urx 0.93.

39. Shen, H., Yosinski, J., Kormushev, P., Caldwell, D. G., & Lipson, H. (2012). Learning fast quadruped robot gaits with the RL power spline parameterization. *Cybernetics and Information Technologies*, *12*(3), 66–75.

40. Sparkfun (2015). RN42-XV Bluetooth module—PCB antenna. https://www.sparkfun.com/products/11601.

41. Stanley, K. O., & Miikkulainen, R. (1996). Efficient reinforcement learning through evolving neural network topologies. *Network (Phenotype)*, *1*(2), 3.

42. Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99–127.

43. Watson, R. A., Ficici, S., & Pollack, J. B. (1999). Embodied evolution: Embodying an evolutionary algorithm in a population of robots. In *Proceedings of the 1999 Congress on Evolutionary Computation, 1999. CEC 99*, vol. 1. Washington, DC: IEEE.

44. Zykov, V., Bongard, J., & Lipson, H. (2004). Evolving dynamic gaits on a physical robot. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, & A. Tyrrell (Eds.), *Proceedings of Genetic and Evolutionary Computation Conference, late breaking paper, GECCO*, vol. 4. New York: Springer.

Appendix: Experimental Parameters

Tables 3 and 4 show the NEAT parameters for the navigation and obstacle task, which were found to work best on this task through prior experimentation. Movement speed of the mobile robot was slightly reduced from the navigation (Table 5) to the obstacle avoidance task (Table 6) to facilitate navigating around obstacles.

Table 3. MultiNEAT parameters for the navigation task.

MultiNEAT parameter	Value
PopulationSize	15
MinSpecies	0
MaxSpecies	2
YoungAgeThreshold	2
OldAgeThreshold	5
OverAllMutationRate	0.8

Table 4. MultiNEAT parameters for the obstacle avoidance task.

MultiNEAT parameter	Value
PopulationSize	15
MinSpecies	1
MaxSpecies	3
SpeciesDropOffAge	3
YoungAgeThreshold	2
OldAgeThreshold	5
OverAllMutationRate	0.8

Table 5. Move command parameters for the navigation task.

Movement parameter	Value
Forward speed	33% (130)
Forward duration	180 ms
Turn Speed	25% (100)
Turn duration	100 ms

Table 6. Move command parameters for the obstacle avoidance task.

Movement parameter	Value
Forward speed	130 (32.5% of maximum speed)
Forward duration	120 ms
Turn Speed	100 (25% of maximum speed)
Turn duration	180 ms
IR frequency	300 ms
IR power levels	(2, 4, 6, 8, 10, 13, 18, 26, 32, 38)

Note. The interested reader is referred to the Zumo Pololu Zumo 32U4 Robot User's Guide for more details about these parameters [34].